U.S. Department of Transportation
Federal Aviation Administration

Technical Center

Atlantic City Int'l Airport
New Jersey 08405

May 4, 1992

Dear Colleague,

Enclosed is additional material for the Digital Systems Validation Handbook - Volume II (DOT/FAA/CT-88/10). These additions consist of a copy of Chapter 17 - Software Quality Metrics, and a revised Table of Contents, List of Authors, Glossary, and Acronyms and Abbreviations. Please add these materials to your Handbook.

If you have any questions or comments concerning the material, or if you would like to receive a copy of the Handbook, please feel free to contact me at the Technical Center.

Sincerely,

Pete Saraceni

Pete Saraceni
FAA Technical Center
ACD-230, Building 201
Atlantic City International Airport, NJ 08405

Phone: (FTS) 482-5577
(609) 484-5577

Enclosures: Chapter 17 - Software Quality Metrics
Table of Contents (revised)
List of Authors (revised)
Glossary (revised)
Acronyms and Abbreviations (revised)

DIGITAL SYSTEMS VALIDATION HANDBOOK - VOLUME II

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF AUTHORS

Clifton A. Clarke
Boeing Commercial Airplane Company
P. O. Box 3707
Seattle, WA 98124

William W. Cooley
Science & Engineering Associates, Inc.
701 Dexter Avenue
Seattle, WA 98109

Hardy P. Curd
Computer Resource Management, Inc.
950 Herndon Parkway, Suite 360
Herndon, VA 22070

Donald Eldredge
Battelle Columbus Division
505 King Avenue
Columbus, OH 43201

Donald A. Elwell
Computer Resource Management, Inc.
950 Herndon Parkway, Suite 360
Herndon, VA 22070

Robert E. Evans
FAA Technical Center
Atlantic City International Airport, NJ 08405

Lee H. Harrison
Computer Resource Management, Inc.
950 Herndon Parkway, Suite 360
Herndon, VA 22070

Myron J. Hecht
SoHaR, Inc.
8500 Wilshire Boulevard
Beverly Hills, CA 90211

John H. Hensyl
Computer Resource Management, Inc.
950 Herndon Parkway, Suite 360
Herndon, VA 22070

LIST OF AUTHORS
(Continued)

William E. Larsen
FAA Field Office
P. O. Box 25
Moffett Field, CA 94035

Susan Mangold
Battelle Columbus Division
505 King Avenue
Columbus, OH 43201

Roger McConnell
CK Consultants
5473 A Clouds Rest
Mariposa, CA 95338

R. L. McDowall
Computer Resource Management, Inc.
950 Herndon Parkway, Suite 360
Herndon, VA 22070

John G. McGough, Consultant
150 Walnut Street
Ridgewood, NJ 07450

Barbara G. Melander
Science & Engineering Associates, Inc.
701 Dexter Avenue
Seattle, WA 98109

Lloyd N. Popish, Consultant
525 Davenport Court
Sunnyvale, CA 94087

John E. Reed
FAA Technical Center
Atlantic City International Airport, NJ 08405

Deborah L. Shortess
Science & Engineering Associates, Inc.
701 Dexter Avenue
Seattle, WA 98109

Nancy J. VanSuetendael
Computer Resource Management, Inc.
950 Herndon Parkway, Suite 360
Herndon, VA 22070

GLOSSARY

α-FAULT. (10) A fault activated by the baseline program (see ß-FAULT).

A-SPECIFICATION. (9) The highest level specification typically produced by the contracting organization to define a system (see MIL-STD-1521).

ABSORPTION LOSS. (11) Attenuation or retention of electromagnetic energy passing through a material, a shield. Absorption loss and reflection loss contribute to total shielding effectiveness (SE).

ACCESS. (18) The process of a transmitting bus user obtaining control of a data bus in order to transmit a message.

ACTION INTEGRAL. (13) The action integral is a critical factor in the production of damage. It relates to the energy deposited or absorbed in a system. This energy cannot be defined without knowing the resistance of the system. The instantaneous power dissipated in a resistor is I^2R and is expressed in watts. For the total energy expended, the power must be integrated over time to get the total joules, watt-seconds. By specifying the integral of $i(t)^2$ over the time interval involved, a useful quantity is defined for application to any resistance value. In the case of lightning, this quantity is defined as the action integral and is specified as $i(t)^2dt$ over the time the current flows.

ACTIVE FAULT. (10) A fault that can produce an error (for some input) while executing the current program.

ACTUAL TRANSIENT LEVEL. (13) The actual transient level is the level of transients which actually appear at the system interfaces as a result of the external environment. This level may be less than or equal to the transient control level but should not be greater.

ADDRESSING CAPACITY. (6) The number of components addressable by the protocol used on a given data bus.

ADVISORY CIRCULAR. (18) An external FAA publication consisting of nonregulatory material of a policy, guidance, and informational nature.

AIR TRANSPORT AIRCRAFT. (18) Aircraft used in interstate, overseas, or foreign air transportation.

AIRCRAFT LIGHTNING INTERACTION. (13) An encounter with lightning that produces sufficient current within or voltages along an aircraft skin or structure to pose a threat to the aircraft electrical/electronic systems, as a result of a direct lightning attachment.

AIRWORTHINESS STANDARDS. (18) Parts 23, 25, 27, 29, and 33 of the Code of Federal Regulations, Title 14, Chapter 1, Subchapter C.

AMBIENT. (16) The substance which absorbs heat from the heat sink.

ANALYTICAL REDUNDANCY. (7) The use of software algorithms which use known mathematical relationships between different sensors for sensor failure detection and replace most of additional redundant sensor hardware.

ANALYTICAL ROOT SOLUTION. (4) Information obtained from the roots of the characteristic equations of the airplane model such as short-period or phugoid frequency response.

ANGLE OF ATTACK. (4) Angle between the longitudinal axis of an aircraft and the direction of movement.

ANODIZE. (11) A preparation by electrolytic process that deposits a protective oxide, insulating film on a metallic surface (aluminum). The oxide defeats electrical bonding. Alodine and iridite finishes on aluminum are conductive.

ANTAGONISTIC QUALITY FACTORS. (17) Quality Factors with conflicting attributes.

APERTURE. (11) An opening, such as a nonconductive panel joint, slot, or crack, allowing electromagnetic energy to pass through a shield.

ARCHITECTURE. (18) The design and interaction of components of a computer system.

ARTIFICIAL INTELLIGENCE. (16) The characteristics of a machine programmed to imitate human intelligence functions.

ASSURANCE ASSESSMENT. (4) Procedures whose purpose is to ensure that a proposed system functions according to design specifications.

ASYNCHRONOUS MESSAGES. (6) Electronic signals with transmission times that are not known a priori. These may include priority signals requiring immediate access to the bus.

ATTACHMENT POINT. (13) A point of contact of the lightning flash with the aircraft.

AUDIO FREQUENCY (AF). (11) The spectrum (20 to 20,000 Hz) of human hearing, often defined as extending from approximately 20 Hz to 50 kHz and sometimes to 150 kHz. Audio noise is nuisance hum, static, or tones from power line 400 Hz, switching regulator and digital clock harmonics, or HF, VHF transmitter frequencies.

AUTOFEATHER. (16) To automatically and swiftly feather the propeller when the engine fails to drive it.

AUXILIARY PROGRAMS. (10) Software executed occasionally.

AVALANCHING LATENT FAULTS. (10) The successive activation of latent faults.

AVIONIC. (18) Electronic equipment used in aircraft.

ß-FAULT. (10) A fault not activated by the baseline program (see α-FAULT).

BABBLING TRANSMITTER. (18) A bus user that transmits outside its allocated time.

BACKSHELL. (11) Metal shell connecting circuit shields or overbraid to an electrical connector.

BACKWARD RECOVERY. (9) Restoration of the system to some previous known correct state and restarting the computation from that point.

BALANCED CIRCUIT. (11) A signal, acting line-to-line, between two conductors having symmetrical voltages identical and equal in relation to other circuits and to ground. "Differential mode" is line-to-line; "common mode" is line to ground.

BALANCED CONFIGURATION. (18) A bus using the HDLC protocol that connects only primary stations.

BANDWIDTH (BW). (11) Frequencies bounded by an upper and lower limit in a given band associated with electronic devices, filters, and receivers.

BASELINE PROGRAM. (10) A set of continuously executed software modules.

BENIGN FAULT. (10) A fault that cannot produce an error while executing the current program, regardless of input, but may produce an error for some other program.

BIDIRECTIONAL DATA BUS. (18) A data bus with more than one user capable of transmitting.

BINARY SEARCH. (17) A searching algorithm in which the search population is repeatedly divided into two equal or nearly equal sections.

BIT-ORIENTED PROTOCOL. (18) A communication protocol where message frames can vary in length, with single bit resolution.

BIT TIME. (6) The time it would take to transmit one bit. Usually this is "blank" time when nothing is being transmitted. One nth of the bus speed (i.e., on a 1 kHz bus, the bit time is 10^{-3} seconds).

BITS. (17) Binary digits.

BLOCK TRANSFER. (6) A data transfer mode allowing the transfer of variable length data blocks.

BOND, ELECTRICAL. (11) Electrical connection at two metallic surfaces securely joined to assure good conductivity often 2.5-mΩ maximum for electrical/elec-

tronic units and 1Ω for electrostatic dissipation or safety. A "faying surface" bond maintains contact between relatively large or long surfaces. Inherently bonded parts are permanently assembled and conductivity exists without special preparation: such as with welding, brazing.

BRAID, OVERBRAID. (11) Fine metallic conductors woven to form a flexible conduit or cableway and installed around insulated wires to provide protection against electric fields and radio frequencies. Best when peripherally connected to backshells. A grounding strap/jumper may be made of braid.

BRIDGE. (18) A BIU that is connected to more than one bus for the purpose of transferring bus messages from one bus to another, where all the buses follow the same protocol.

BROADBAND. (12) A frequency spectrum which is wide compared to the bandwidth of the device used to detect it.

BROADCAST. (4) Transmission of messages to all terminals without reference to the identification of the receiving station or terminal.

BROADCAST CAPABILITY. (6) The capacity to transmit messages to all terminals simultaneously.

BROADCAST DATA BUS. (18) A data bus where all messages are transmitted to all bus users.

BUFFER. (18) Memory used to hold segments of the data transferred between asynchronous processes.

BUS. (18) A conductor that serves as a common connection of a signal to multiple users.

BUS CONTROLLER. (18) The electronic unit that is designed to control the bus communication of all users for a centrally controlled bus.

BUS INTERFACE UNIT. (18) The electronics that interface the host CPU of an LRU to a bus medium.

BUS MESSAGE. (18) A complete set of bits that can be transferred between two bus users.

BUS NETWORK. (18) The collection of all BIUs and bus media associated with one bus.

BUS OVERLOAD. (18) The condition that exists when the time it takes to transmit outstanding messages on a bus exceeds the time allotted for those transmissions.

BUS USER. (18) Any LRU attached to a bus.

BYZANTINE RESILIENCE. (5) A fault tolerant process which is tolerant of intermittent faults that can send good information part of the time.

CABLE OR HARNESS. (11) A bundle of separate, insulated, electrical circuits, shielded or unshielded, usually long and flexible and having breakouts, terminations, overbraid, and mounting provisions completely assembled.

CABLEWAY. (11) A solid metallic housing (liner, foil, coating) surrounding and shielding insulated electrical conductors. Also called conduit, tray, or raceway. Crosswise or transverse openings or breaks in the metallic cableway cause noise voltages to be transferred to internal wire circuits.

CANARD. (16) A tail-first aerodyne, usually with auxiliary horizontal surface at the front and a vertical surface at the back.

CAT IIIa LANDING. (6) One of several landing categories defined in FAR 91. CAT IIIa implies the need for an instrument landing approach.

CENTRAL BUS CONTROL. (18) The bus control approach where a single electronic unit attached to a bus controls all the communication of the bus users.

CENTRAL CONTROL. (6) Control from one master, whether stationary or non-stationary.

CERTIFICATION. (18) The process of obtaining FAA approval for the design, manufacture, and/or sale of aircraft and associated systems, subsystems, and parts.

CHARACTER-ORIENTED PROTOCOL. (18) A communication protocol where messages can vary in length, with single character resolution.

CHARGE TRANSFER. (13) The integral of the current over its entire duration, $i(t)dt$, in coulombs.

CHECKSUM. (18) An error detection code produced by performing a binary addition, without carry, of all the words in a message.

CHORD. (4) The straight line segment intersecting or touching an airfoil profile at two points.

CLOSED-LOOP. (18) A system where the output is a function of the input and the system's previous output.

CODE. (17) The subset of software which exists for the sole purpose of being loaded into a computer to control it.

COMMAND/RESPONSE. (6) "Operation of a data bus system such that remote terminals receive and transmit data only when commanded to do so by the controller." (MIL-STD-1553 Designer's Guide, 1983, p. II-3.)

COMMAND/RESPONSE DATA BUS. (18) A data bus whose protocol initiates each data transfer with a command and terminates the transfer after a proper response is received.

COMMON MODE IMPEDANCE. (11) Impedance or resistance shared by two or more circuits so that noise voltages/currents generated by one are impressed on the others.

COMMON MODE REJECTION. (11) The ability of wiring or an electronic device to reject common mode (line-to-ground) signals and maintain fidelity of differential mode (line-to-line) signals.

COMMON MODE SIGNAL. (11) Identical and equal signals on input conductors or at the terminals of a device relative to ground.

COMPLEMENTARY QUALITY FACTORS.. (17) Quality Factors with interrelated attributes.

COMPONENT DAMAGE. (13) Condition arising when the electrical characteristics of a circuit component are permanently altered beyond its specifications.

CONDUCTED EMISSION (CE) OR INTERFERENCE. (11) Voltage/current noise signals entering or leaving a unit on interface conductors. Emission is the general term, interference is undesired noise.

CONFIGURATION MANAGEMENT. (18) The precise control and documentation of the configuration of an entity at any time during its development and deployment.

CONTENT ADDRESSING. (6) The system of identifying message recipients based on information embedded in the message. This is in contrast to destination terminal addresses.

CONTENTION PROTOCOL. (18) A protocol that allows users to randomly access the bus at any time. When bus contention results, each user tries again to access the bus without contention.

CONTROL LAW. (7) The physical relationship between various sensors and control surfaces.

CONTROL REGISTER. (18) A register in an IC controller that receives commands from a host processor.

CONTROL STRUCTURES. (17) Programming constructs which direct the flow of control.

CORONA. (13) A luminous discharge that occurs as a result of an electrical potential difference between the aircraft and the surrounding atmosphere.

COUPLING. (11) The transfer of energy between wires or components of a circuit electrostatically, electromagnetically, or directly.

COVERAGE. (5) The conditional probability of the system successfully recovering from a component fault and continuing to perform the intended functions correctly, given the presence of the fault. Coverage is the measure of effectiveness of a system's utilization of redundant hardware. Coverage can be qualified and applied to many different components of a system and phases of

recovery process. Examples include, fault detection coverage, fault isolation coverage, latent fault coverage, sensor failure coverage, and memory failure coverage.

COVERAGE. (7) The percent confidence level of a given analytical redundancy fault detection and isolation algorithm for all types of faults.

COVERAGE. (9) The probability that when a fault occurs, it will be detected and recovery from the fault will be successful.

CRITICAL. (13) Functions whose failure would contribute to or cause a failure condition which would prevent the continued safe flight and landing of the aircraft.

CROSS COUPLING (CROSSTALK). (11) Transfer of signals from one channel, circuit, or conductor to another as an undesired or nuisance signal or the resulting noise.

DAMAGE. (11) The irreversible failure of a component.

DATA BUS. (6,18) A system for transferring data between discrete pieces of equipment in the same complex.

DATA BUS PROTOCOL. (18) The set of rules that governs the transfer of data between data bus users.

DATA LATENCY. (5,18) The delay from the time when a piece of information becomes available at a source terminal to the time it is received at the destination.

DATA LINK ASSURANCE OF RECEIPT. (6) The guarantee of good data through the data link level.

DATA REASONABLENESS CHECK. (18) A check performed to see if a value of data is within reasonable bounds for the given context.

dBμV. (12) Decibels referred to one microvolt. Zero db represents one microvolt.

DECIBEL (dB). (11,12) Decibel expresses the ratio between two amounts of power, P1 and P2, at two separate points in a circuit. By definition, the number of dB = 10 log to the base 10 of (P1/P2). For special cases, when a standard power level P2 = 1 mW or 1 W or 1 kW, then the ratio is defined as "dBm," "dBw," or "dBKW." Because $P = V^2/R$ and also I^2R, decibels express voltage and current ratios. Ideally, the voltages and currents are measured at two points having identical impedances. By definition, dB = 20 log V1/V2 and dB = log I1/I2. For convenience, V2 or I2 are often chosen as 1 μV or 1 μA and the ratio is defined as dB above a μV or dB above a μA when graphing emission or susceptibility limits.

DECOUPLED MANEUVERS. (4) Changes in an aircraft's direction and attitude in one axis without affecting direction or attitude in other axes.

DEFAULT DATA. (18) An alternative value used for a parameter whenever the normal data is not supplied.

DESIGN ERROR. (4) A functional flaw resulting from a misinterpretation of the specifications of the system.

DESIGN MARGIN. (13) The difference between the equipment transient design levels and the transient control level.

DETERMINISTIC. (6) A system where all parameters are known, as opposed to a statistical system where the outcome is subject to the laws of probability.

DETERMINISTIC PROTOCOL. (18) A protocol where all parameters are known so that its various states are predictable in sequence and time.

DIAGNOSTIC FILTER. (7) An analytical algorithm which processes data from N functionally related sensors. The data are used to estimate some sensor outputs and assess the correct functioning of the sensors.

DIELECTRIC STRENGTH. (11) Voltage withstand capability that an insulating material sustains before destructive arcing and current flow, usually expressed in volts per mil thickness. Dielectric withstand voltage is the voltage level at which insulation breakdown occurs.

DIFFERENTIAL MODE (DM) SIGNAL. (11) The signal in a two-wire circuit measured from line-to-line.

DIGITAL DATA BUS. (18) A data bus that uses digital electronic signals.

DIRECT EFFECTS. (13) Any physical damage to the aircraft or onboard systems due to the direct attachment of the lightning channel. This includes tearing, bending, burning, vaporization, or blasting of aircraft surfaces or structures, and damage to electrical/electronic systems.

DISSIMILAR REDUNDANCY. (18) The redundancy of systems that provide a redundancy of function, but by a different form.

DISSIMILAR SOFTWARE. (18) Redundant computer programs that provide a redundancy of function, but by a different form.

DISTRIBUTED BUS CONTROL. (18) The bus control approach where the total communication control job is distributed across the bus users, each controlling the communications during its period of responsibility.

DISTRIBUTED CONTROL. (6) Concurrent control from multiple points in the data bus system.

DOUBLE FAIL-OPERATIONAL SYSTEM. (4) A quadruplex (or higher) redundant flight-control system which is designed to incur failures in two redundant lanes (or channels) before it fails.

DUAL-DUAL ARCHITECTURE. (4) Two parallel dual computers with a voting plane at the output of each dual computing lane.

DUAL FAIL-OPERATIONAL. (7) A reliability requirement placed on a system which requires the system to be operational after two failures have occurred.

DUAL GROUND. (11) Equipment case ground/return through two independent circuit paths to structure implemented in flammable zones and water leakage areas-each path meeting electrical conductivity (resistance) requirements.

ELECTRIC FIELD. (11) High-impedance, radiated voltage field, positive or negative, from a voltage source as contrasted to a low-impedance magnetic field from a current source.

ELECTROMAGNETIC COMPATIBILITY (EMC). (11) Operation within performance specification in the intended electromagnetic interference environment.

ELECTROMAGNETIC INTERFERENCE (EMI). (11) Conducted and radiated voltage/current noise signals, broadband (BB) or narrow band (NB), that degrade the specified performance of equipment.

ELECTROMIGRATION. (5) Drifting of metal atoms toward the cathode of a cathode ray tube.

ELECTROSTATIC CHARGE. (11) Electric potential energy with a surrounding electric field, uniform or nonuniform, moving or at rest, on a material.

EMISSION. (11) Voltage/current noise on a wire or in space. Broadband emission has uniform spectral energy over a wide frequency range and can be identified by the response of a measuring receiver not varying when tuned over several receiver "bandwidths." Or, energy present over a bandwidth greater than the resolution bandwidth where individual spectral components cannot be resolved. Broadband (BB) may be of two types: (1) impulse and coherent varies 20 dB per decade of bandwidth and (2) random or statistical, varies 10 dB per decade. A narrow band (NB) emission or signal, sometimes called continuous wave, occurs at a discrete frequency and does not vary with bandwidth.

EMULATION. (18) The duplication of the behavior of a system with a different system.

ENVELOPE LIMITING. (4) General or additional limits imposed on the structural, "g" limits, speed, attitude, etc. of the aircraft. In some cases, envelope limiting imposes additional constraints on the envelope that cannot be exceeded regardless of pilot inputs.

EQUIPMENT TRANSIENT DESIGN LEVEL. (13) The level of transients which the equipment is qualified to withstand.

EQUIPMENT TRANSIENT SUSCEPTIBILITY LEVEL. (13) The transient level which will result in damage or upset to the system components. This level will be greater than the equipment transient design level.

EQUIVALENCE STATEMENT. (17) A FORTRAN statement which equates two variable names.

ERROR. (4) A mistake in specification, design, production, maintenance, or operation of a system causing undesirable performance.

ERROR. (8) A state of the system which (in the absence of any corrective action by the system) could lead to a failure that would not be attributed to any event subsequent to the error. (More accurately known as an erroneous state.)

ERROR MASKING. (18) The process of masking the presence of avionic errors, possibly by using an electronic voter to override an erroneous input with the values of substitute inputs.

EVENT, EXTREMELY IMPROBABLE. (4) An event with a probability of occurrence on the order of 10^{-9} or less.

EVENT, IMPROBABLE. (4) An event with a probability of occurrence on the order of 10^{-5} or less.

EVENT, PROBABLE. (4) An event with a probability of occurrence on the order of 10^{-5} or greater.

EXTERNAL ENVIRONMENT. (13) Characterization of the natural lightning environment with idealized waveforms for engineering purposes.

FAIL-OPERATIONAL. (7) A reliability requirement placed on a system which requires the system to be operational after a single failure has occurred.

FAIL-SAFE. (7) A reliability requirement placed on a system which requires that safe flight not be hindered even after a failure.

FAIL-SAFE. (18) A design philosophy that ensures that any failure in a system does not result in an unsafe condition after the failure.

FAILURE. (4) The inability of a system, subsystem, unit, or part to perform within specified limits.

FAILURE. (5) The deviation of system behavior from specifications (arithmetic failure, storage failure, flight control function failure.)

FAILURE. (8) The situation when the external behavior of a system does not conform to that prescribed by the system specification.

FAILURE, HARD. (5) Repeated use of the same input and initial conditions results in the same incorrect response.

FAILURE, HIDDEN. (4) A failure that is not manifested at the time of its occurrence.

FAILURE MECHANISM. (5) Any situation that could produce an error condition. Examples of failure mechanisms include metal migration, voltage overstress, and lack of air-conditioning.

FAILURE, PERMANENT. (5) Repeated use of the same input and initial conditions results in the same incorrect response.

FAILURE, SOFT. (5) Repeated use of the same input and initial conditions does not result in the same incorrect response.

FAILURE, TEMPORARY. (5) Repeated use of the same input and initial conditions does not result in the same incorrect response.

FAILURE, TRANSIENT. (5) Repeated use of the same input and initial conditions does not result in the same incorrect response.

FALL-TIME. (12) The time required for pulse amplitude to go from a predefined magnitude to a given level.

FALSE ALARM. (7) The declaration of a fault by a fault detection monitor or algorithm when there is no fault.

FAULT. (4) An error in the operation of a system.

FAULT. (5) The phenomenological reason for a failure (open wire, stuck-at fault, design fault, etc.). In general, any condition preventing a digital component from correctly changing state when directed to change by input parameters. For electrical components there is a one-to-one correspondence between faults and failures. The situation is not so simple with digital circuits. For if the circuit is S-A-1, any input causing a one output will be correctly processed; a little like the stopped clock that is correct twice per day. For a processor having a million or so logic gates, it is not possible to test for all the combinations of input and output states.

FAULT. (8) The adjusted cause of error.

FAULT AVOIDANCE. (9) The attempt to prevent any software faults in the final delivered product through disciplined software development practices, testing, and IV&V.

FAULT CONTAINMENT. (6,9) The capacity of a system to prohibit errors and/or failures from propagating from the source throughout the system.

FAULT CURRENT. (11) The maximum current (magnitude and duration) flowing through a fault point. This current is equal to the supply voltage divided by the dc resistance of power line leads, circuit breakers, and the current return in wire or structure.

FAULT DETECTION. (6) The capacity of a system to determine the occurrence of erroneous operation.

FAULT DETECTION. (7) The determination that a sensor is faulted by using a software algorithm.

FAULT, HARD. (4) A defect in the hardware or software of a digital control system that permanently affects some functional performance of the system.

FAULT INSERTION. (4) A testing technique used to obtain information about data latency and built-in test coverage of a digital flight-control system.

FAULT ISOLATION. (6) The capacity of a system to isolate a failure to the required level so it can reconfigure.

FAULT ISOLATION. (7) The determination that a particular sensor is faulted by using a software algorithm.

FAULT, LATENT. (5) A fault which has not yet caused a failure. (For example, a fault in a memory chip that is not being used for the foreground program or in this particular mode of the system is a latent fault.)

FAULT, SOFT. (4) A transient defect in the software of a digital flight-control system that can be overcome by error-correctable code or by recycling of power to the computer system.

FAULT, STUCK-AT. (5) A logic signal which remains at zero (S-A-0) or one (S-A-1).

FAULT TOLERANCE. (6,9) The capability to endure errors and/or failures without causing total system failure.

FAULT TOLERANCE. (7) Accommodation of sensor hardware faults based on some type of comparator scheme.

FAULT TOLERANCE. (18) The ability of a system to continue operation after a fault, possibly in a degraded condition.

FAULT TOLERANT. (4,9) Software which continues to operate satisfactorily in the presence of faults.

FAULT TOLERANT SYSTEM. (5) A system that continues to function although certain components may have faults.

FAULT TREE ANALYSIS. (4) A top-down deductive analysis that identifies the conditions and functional failures necessary to cause a defined failure condition. The fault tree can be used to establish the probability of the ultimate failure condition occurring as a function of the estimated probabilities of contributory events.

FEDERAL AVIATION REGULATIONS. (18) Subchapter C of the Code of Federal Regulations, Title 14, Chapter 1.

FILTER. (11) Device or unit that passes or rejects a frequency band and is designed to block noise from entering or leaving a circuit or unit.

FINITE STATE MACHINE. (18) A state machine with a finite number of states.

FLIGHT CODE. (4) The application software of the digital flight-control system.

FLIGHT-CRITICAL. (4,7,18) A description of functions whose failure would contribute to or cause a failure condition preventing the continued safe flight and landing of the aircraft.

FLIGHT-ESSENTIAL. (4,18) A description of functions whose failure would contribute to or cause a failure condition which would significantly affect the safety of the airplane or the ability of its crew to cope with adverse operating conditions.

FLIGHT-NONESSENTIAL FUNCTION. (18) A function whose failure could not significantly degrade aircraft capability or crew ability.

FLIGHT-PHASE CRITICAL. (4) A description of functions which are critical only during certain phases of flight.

FLY-BY-GLASS. (16) Flight control system where fiber optics carry the signal.

FLY-BY-LIGHT. (4,16) Flight control system where fiber optics carry the signal.

FLY-BY-WIRE. (4,16) Flight control system with electric signaling.

FORWARD RECOVERY. (9) Restoration of the system to a consistent state by compensating for inconsistencies found in the current state so that the system may continue processing.

FOURIER TRANSFORM. (12) A mathematical method for deriving the frequency spectrum from a time dependent function.

FRAME. (18) A formatted block of data words or bits that is used to construct messages.

FUNCTIONAL PARTITIONING. (18) The partitioning of system functions by placing each group of users, which share a common function, on different data buses.

GATEWAY. (18) A bus user that is connected to more than one bus for the purpose of transferring bus messages from one bus to another, where the buses do not follow the same protocol.

GENERAL AVIATION AIRCRAFT. (18) The non-air transport civil aircraft.

GENERATOR POLYNOMIAL. (18) The polynomial code that is used to generate the remainder in the division of the CRC check.

GIGABIT. (16) One billion bits.

GLASS COCKPIT. (9) Advanced state-of-the-art electronic displays utilizing flat panel and/or cathode ray tube display technology for cockpit instrumentation.

GLOBAL STATE. (18) A state that represents the condition of the entire network being modeled, including senders, receivers, and the communication link.

GROUND. (11) A generic term having multiple meanings and indicating a circuit return path or a voltage reference: not "zero" voltage reference. Four hundred millivolts of noise voltage is common on "quiet" grounds. There are several types of returns and references.

GROUND EFFECT. (4) Increase in aircraft lift when operating near the ground.

HALF-DUPLEX. (18) Bidirectional communication between two entities on a single channel by each having a turn to control the channel.

HAMMING CODE. (18) An error detection and correction code based on the Hamming distance.

HAMMING DISTANCE. (18) The number of bit positions in which two binary words differ.

HANDSHAKING. (18) The reciprocal responses given by two electronic systems to sequence the steps of a transfer of data between them.

HARD FAILURE. (12) A failure that requires a reset of the equipment.

HARDWARE. (17) The physical components of a computer.

HARDWARE-IN-THE-LOOP SIMULATION. (18) A partial simulation of a system; part of the actual system is used in the simulation.

HAZARD FUNCTION. (8) The conditional probability that a fault is exposed in the interval t to Δt given that the fault did not occur prior to time t.

IMMUNITY. (11) Capability of a circuit or unit to operate within performance specification in a specified electromagnetic interference environment.

INDIRECT EFFECTS. (13) Voltage and/or current transients induced by lightning in aircraft electrical wiring which can produce upset and/or damage to components within electrical/electronic systems.

INDUCED VOLTAGES. (13) A voltage produced around a closed path or circuit by changing magnetic or electric fields or structural IR voltages.

INITIALIZATION. (6) Setting the beginning parameters and values on system power-up. For redundant systems this includes setting the initial configuration of the system.

INTERNAL ENVIRONMENT. (13) The fields and structural IR potentials produced by the external environment, along with the voltages and currents induced by them.

INTERRUPT VECTOR. (18) The address that points to the beginning of the service routine for an interrupt.

INTERRUPT VECTOR TABLE. (18) The table of interrupt vectors for all interrupts serviced by a system.

ISOLATION. (11) Electrical separation and insulation of circuits from ground and other circuits or arrangement of parts to provide protection and prevention of uncontrolled electrical contact.

JOULE. (12) A unit of energy equal to one watt-second.

JUMPER/STRAP. (11) A short wire, strip, strap, or braid conductor installed to make a safety ground connection, to dissipate electrostatic charge, or establish continuity around a break in a circuit.

KILOBYTE. (16) One thousand bytes.

LABELED ADDRESSING. (6) The system of identifying message recipients based on labels. This is in contrast to destination terminal addresses.

LATENT FAULT. (10) A fault which has not yet produced a malfunction. (In the context of the single-fault model, benign and latent faults are equivalent.)

LIGHTNING FLASH. (13) The total lightning event in which charge is transferred from one charge center to another. It may occur within a cloud, between clouds, or between a cloud and the ground. It can consist of one or more strokes, plus intermediate or continuing currents.

LIGHTNING LEADER STROKE. (13) The leader forms an ionized path for charge to be channeled towards the opposite charge center. The stepped leader travels in a series of short, luminous steps prior to the first return stroke. The dart leader reionizes the return stroke path in one luminous step prior to each subsequent return stroke in the lightning strike.

LIGHTNING RETURN STROKE. (13) A lightning current surge that occurs when the lightning leader makes contact with the ground or an opposite charge center.

LIGHTNING STRIKE. (13) Any attachment of the lightning flash to the aircraft.

LIGHTNING STRIKE ZONES. (13) Locations on the aircraft where the lightning flash will attach or where substantial amounts of electrical current may be conducted between attachment points. The location of these zones on any aircraft is dependent on the aircraft's geometry and operational factors and often varies from one aircraft to another.

LIMITING, VOLTAGE/CURRENT. (11) Semiconductor components, diodes, Transorb, or filter designed to clip and shunt to ground an applied transient or steadystate voltage. Used to protect against noise frequencies, faults, lightning, and inductive switching transients.

LINE REPLACEABLE UNIT. (18) An electronics unit that is made to be replaced on the flight line, as opposed to one that requires the aircraft be taken to the shop for repair.

LINEAR BUS. (18) A bus where users are connected to the medium; one on each end, with the rest connected in between.

LOW-PASS FILTER. (12) An electrical circuit which allows the passage of low frequencies and prevents the passage of high frequencies.

MAGNETIC FIELD. (11) A radiated, low-impedance field having lines of "flux" or magnetomotive force associated with an electrical current.

MALFUNCTION. (11) Failure or degradation in performance that compromises flight safety.

MANCHESTER II MODULATION. (18) A non-return to zero, bipolar modulation of a voltage that encodes bits based on the zero-crossing direction of the signal.

MEAN AERODYNAMIC CHORD (also mean chord). (4) The chord of an airfoil whose length is equal to the area of the airfoil section divided by the span.

MEAN FAILURE RATE. (10) A measure of survivability defined as the reciprocal of the mean time to system failure.

MESSAGE STRUCTURE. (6) The organization of both protocol and data information in a message.

METRIC. (17) A measure.

MICRON. (16) One-millionth of a meter.

MISSED ALARM. (7) The failure of a fault detection monitor or algorithm to detect a fault when there is a sensor fault.

MODELING. (18) Creating a system of mathematical equations that formulate all the significant behavior of a system.

MODULE. (17) A unit of code which implements a function.

MONITORABILITY. (6) The capacity of the protocol to be viewed passively to allow observation of the dynamics of the protocol.

MONOTONIC FUNCTION. (17) A function in which a certain change in the measure always represents a certain change in the property being measured, where either change is simply an increase or decrease in magnitude.

MULTIPLE BURST. (13) A randomly spaced series of bursts of short duration, low amplitude current pulses, with each pulse characterized by rapidly changing currents. These bursts may result from lightning leader progression or branching and may be accompanied by or superimposed on stroke or continuing currents. The multiple bursts appear to be most intense at the time of initial leader attachment to the aircraft.

MULTIPLE STRIKE. (13) Two or more lightning strikes during a single flight.

MULTIPLE STROKE. (13) Two or more return strokes occurring during a single lightning flash.

MULTIPLE TRIP MONITOR. (7) A fault detection algorithm which declares a fault after the sensor output has exceeded a predefined threshold N times.

MULTIVERSION PROGRAMMING. (18) N-version programming.

N-VERSION PROGRAMMING. (18) The independent coding of a number, N, of redundant computer programs that are run concurrently for the purpose of comparing their outputs.

NANOSECOND. (16) One-billionth of a second.

NEGATIVELY STABILIZED. (4) Aircraft design in which the point of effective lift is aft of the center of gravity.

NETWORK CONTROL STRATEGY. (6) The solution proposed by the designer in addressing his specific problem (design flexibility).

NOISE. (11) Conducted or radiated emission causing circuit upset, performance disorder, or undesired sound.

NUMERICAL APERTURE. (6) The angle of acceptance of light from a light source for a given fiber optic cable.

OBJECT CODE. (17) The translation of source code that is loaded into a computer.

OBSERVER. (7) An algorithm which models physical relationships between sensor data and uses the data to provide fault detection for one or more sensors. This is also known as a Luenberger observer or a signal blender.

OPERANDS. (17) The variables or constants on which the operators act.

OPERATORS. (17) Symbols which affect the value or ordering of operands.

OPTIMIZING COMPILER. (17) A computer program which, while translating source code into object code, removes inefficiencies from the code.

OVERHEAD. (18) The message timing gaps, control bits, and error detection bits added to some data to satisfy the data bus protocol.

PARAMETERIZATION CAPABILITY. (6) A measure of how well the attributes of the protocol can be described by parameters.

PARITY. (18) An error detection bit added to a data word based on whether the number of "one" bits is even or odd.

PARTITIONED. (18) Colocated hardware or software functions that are designed so that adverse interactions between them cannot occur.

PEAK RATE OF RISE. (13) The maximum instantaneous slope of the waveform as it rises to its maximum value. Mathematically, the peak rate of rise of a function, i(t), may be expressed as the maximum of d[i(t)]/dt.

PETRI NET. (18) A state analysis diagram that tracks the status of the state transition conditions of a state machine.

PIN LEVEL TEST. (12) An EMC test in which voltage or current is applied directly to a conductor at a connector pin.

POINT-MASS SIMULATION. (4) Same as state variables airplane model (q.v.).

POLLING. (18) A method whereby a CPU monitors the status of a peripheral by periodically reading its status signals.

POLYNOMIAL CODE. (18) A sequence of bits that represents the coefficients of each term in a polynomial.

POSITIVELY STABILIZED AIRCRAFT. (4) Aircraft design in which the effective point of lift is forward of the center of gravity.

PRECIPITATION STATIC (P-static). (11) Electrostatic discharge, corona, arcing, and streamering, steady state or impulsive, causing circuit upset, receiver noise or component damage.

PREDICATE/TRANSITION NETWORK. (4) A bipartite graph (a type of linear graph) to model concurrency between redundant concurrent events. Basically a modified generalized petri net.

PRIMARY STATION. (18) An intelligent HDLC protocol user, usually used to manage the access of other bus users to the bus.

PROGRAM. (17) A detailed set of instructions for accomplishing some purpose.

PROPAGATION DELAY. (18) The time it takes an electrical signal to travel from its source to its destination.

PROTOCOL. (18) The set of rules by which all bus users must abide to access the bus and ensure its specified operation.

Q. (12) The quality factor of a resonant circuit which is the ratio of the energy stored to the power dissipated per cycle.

QUADRUPLEX ARCHITECTURE. (4) The use of four separate lanes (or channels) of computer redundancy. Each lane can fail separately providing a fail-operational capability for the digital flight-control system.

QUALITY MEASURE. (17) A repeatable, monotonic relationship relating measures of objects (a set of numbers) to subjective qualities.

RADIATED EMISSION (RE). (11) Electromagnetic energy transmitted and propagated in space usually considered as audio frequency or radio frequency noise.

RADIO FREQUENCY (RF). (11) Frequencies in the electromagnetic spectrum used for radio communications extending from kilohertz to gigahertz.

RADIO FREQUENCY INTERFERENCE (RFI). (11) Electromagnetic interference in the radio frequency range.

RECONFIGURATION. (6) The capacity of a system to rearrange or reconnect the system elements or functions.

RECONFIGURATION. (18) The process of a system reassigning which hardware performs a particular function.

RECOVERY BLOCK. (18) A block of code executed upon detection of a fault to recover from the erroneous condition that results.

RECOVERY CACHE. (9) The location used to preserve input values until the outputs resulting from them have been accepted.

REDUNDANCY MANAGEMENT. (7) The computer processing which is needed to implement fault detection and isolation algorithms.

REFERENCE. (11) 1. Structure, for electronics, shields, power. 2. A grid of wires, solid sheet, or foil. 3. A wire from circuit to grounding block or case. 4. A wire from circuit to structure. 5. Shield tie. 6. Earth.

REGISTER. (18) A single word of RAM located within an IC controller that is used for transferring data and control information.

RELAXED STATIC STABILITY AIRCRAFT. (4) An aircraft whose center of gravity is behind the wing's point of effective lift.

RELIABILITY ANALYSIS. (4) A means of determining the probability of failure in a system. Military flight-critical systems typically are required to have reliability levels of 10^{-5} to 10^{-7}, whereas civil flight-critical systems have reliability levels of 10^{-9} or less.

REMOTE TERMINAL. (18) The BIU portion of a MIL-STD-1553 bus user.

RESONANCE. (12) Resonance occurs in an electrical circuit when the energy stored in the inductance is equal to the energy stored in the capacitance.

RETURN. (11) 1. Structure, for power, fault, and "discrete" circuits. 2. A grid of wires, solid sheet, or foil. 3. A wire from circuit load back to source or to case. 4. Circuit card "ground plane," also a reference and shield.

RETURN STROKE. (13) See lightning return stroke.

REVERSION MODE. (7) The high level of redundancy in a system having different redundancy requirements for some sensors. Critical sensors may have a high level of redundancy while other sensors have low levels.

RING BUS. (18) A bus where users are connected only to the two adjacent users in a continuous ring; each connected to the next and the last one connected to the first one.

RISE-TIME. (12) The time required for a voltage pulse to reach a predefined magnitude from a given level.

ROBUSTNESS. (9) The ability of the code to perform despite some violation of the assumptions in its specifications usually via substitution of an alternate value and continuation of execution if a software fault is detected.

ROLLBACK. (9) Retrying the calculation in the event that a failure is detected, under the assumption that some external condition may have changed thereby resolving the anomaly.

SEALANT. (11) An applied substance enclosing and protecting the integrity of a joint, fastener, or electrical bond from moisture, contaminants, oxidation, and acid or alkaline corrosion.

SECONDARY STATION. (18) A simple HDLC protocol user.

SENSOR. (7) An instrument which measures a particular physical parameter. The data output may be digital or analog and is utilized by the flight computer.

SENSOR. (18) Any transducer that converts the measurement of a physical quantity to an electrical signal.

SEQUENTIAL LIKELIHOOD RATIO TEST. (7) A fault detection algorithm which is based on two hypothesized density functions of no fault or sensor fault.

SEQUENTIAL PROBABILITY RATIO TEST. (7) See sequential likelihood ratio test.

SERIAL DATA BUS. (18) A data bus capable of sending only one bit at a time, in series.

SERVICE SPECIFICATION. (18) The specification of the service provided by a protocol layer.

SHIELD. (11) A conductive material, opaque to electromagnetic energy, for confining or repelling electromagnetic fields. A structure, skin panel, case, cover, liner, foil, coating, braid, or cable-way that reduces electric and magnetic fields into or out of circuits or prevents accidental contact with hazardous voltages.

SHIELD EFFECTIVENESS (SE). (11) The ability of a shield to reject electromagnetic fields. A measure of attenuation in field strength at a point in space caused by the insertion of a shield between the source and the point.

SHIELDING. (12) Any metallic structure such as the aircraft fuselage or the woven braid on a cable that provides protection against electromagnetic fields.

SIGNAL RETURN. (11) A wire conductor between a load and the signal or driving source. Structure can be a signal and power return. Commonly, it is the low voltage side of the closed loop energy transfer circuit.

SIMULATION. (18) An approximated representation of the behavior of a system with a similar system.

SINGLE-ENDED CIRCUIT. (11) A circuit with source and load ends grounded to case and structure and using structure as return.

SINGLE-POINT FAILURE. (18) A failure of a component that, by itself, causes the failure of the system in which it is contained.

SINUSOID. (12) A wave form that follows the mathematical values of a sine function.

SOFT FAILURE. (12) A failure which causes an alteration of data or missing data.

SOFTWARE. (17) Computer programs and the documentation associated with the programs.

SOFTWARE METRIC. (17) A measure of software objects.

SOFTWARE QUALITY FACTOR. (17) Any software attribute that contributes either directly or indirectly, positively or negatively, toward the objectives for the system in which the software resides.

SOFTWARE QUALITY METRIC. (17) (1) A measure that relates measures of the software objects (the symbols) to the software qualities (quality factors). (2) The measure of a software quality factor.

SOURCE CODE. (17) Code that can be read by people.

SPECIAL CONDITION. (18) A regulatory document that adds to, or otherwise alters, the airworthiness standards for particular aircraft.

STATE-VARIABLE AIRPLANE MODEL (also point-mass model). (4) Fixed aerodynamic variables are used in the solution of the equations of motion of the model instead of using look-up tables in which each derivative varies with airspeed, altitude, etc. The model performance is only accurate at or near the point in the flight envelope for which the variables are chosen.

STATIC MARGIN. (4) The degree of instability in a relaxed statically stable airplane.

STATION. (18) Bus user.

STATIONARY BUS CONTROL. (18) Bus control that is continually performed by a single bus controller, or by one of its backups.

STATUS REGISTER. (18) A register in an IC controller that holds the status of the state of certain controller functions.

STROUD NUMBER. (17) The total number of elementary mental discriminations that a person makes per second.

STRUCTURAL IR VOLTAGE. (13) The portion of the induced voltage resulting from the product of the distributed lightning current, I, flowing through the resistance, R, of the aircraft skin or structure.

STRUCTURE. (11) Basic members, supports, spars, stanchions, housing, skin panels, or coverings that may or may not provide conductive return paths and shields for electrical/electronic circuits.

STUB. (18) The short length of cable used to attach a single LRU to a data bus.

SUBROUTINE. (17) A self-contained body of code which can be called by other routines to perform a function.

SUPER-DIAGNOSTIC FILTER. (7) An algorithm which provides all the capabilities of a diagnostic filter. Additionally, it can isolate a specific faulted sensor. At the current time, this is the most complex technique used to implement analytical redundancy.

SUSCEPTIBILITY. (11) Upset behavior or characteristic response of an equipment when subjected to specified electromagnetic energy. Identified with the point, threshold, or onset of operation outside of performance limits. Conducted Susceptibility (CS) applies to energy on interface conductors; Radiated Susceptibility (RS) to radiated fields.

SWEPT STROKE. (13) A series of successive attachments due to sweeping of the flash across the surface of the airplane by the motion of the airplane.

SYNCHRONOUS MESSAGES. (6) Messages transmitted at a known a priori sequence and time or time interval.

SYSTEM EXPOSURE TIME. (4) The period during which a system may fail. This period extends from the last verified proper functioning to the completion of the next required performance.

SYSTEM FUNCTIONAL UPSET. (13) Impairment of system operation, whether permanent or momentary (e.g., a change of digital or analog state) which may or may not require manual reset.

SYSTEM INTEGRATOR. (18) The developer who has the responsibility to integrate the various subsystems into a working system.

SYSTEM INTEGRITY. (6) The degree to which a system is dependable.

SYSTEM RELIABILITY. (5) The probability of performing a given function from the some initial time, t=0, to time t.

TESTABILITY. (6) A measure of how well the protocol supports completeness of testing and the protocol's ability to produce repeatable or predictable results.

THRESHOLD, NOISE. (11) The lowest electromagnetic interference signal level that produces onset of susceptibility.

THROUGHPUT. (6) The productivity of a data processing system as expressed in computing work per minute or hour.

THYRISTORS. (16) Solid-state devices that convert alternating current to direct current.

TIME CONSTANT. (4) Time required to double the amplitude of the divergent real root in the pitch axis of the aircraft model.

TOKEN PASSING PROTOCOL. (18) A protocol that limits bus access to the user that has just received the token word.

TRANSIENT CONTROL LEVEL. (13) The maximum allowable level of transients appearing at the systems interfaces as a result of the defined external environment.

TRANSPARENT RECOVERY. (4) Correcting a soft fault without interrupting the system's intended performance.

TRIBOELECTRIC CHARGING. (13) Static electricity produced on a structure from the effects of friction.

UNACCEPTABLE RESPONSE. (11) Upset, degradation of performance, or failure, not designated a malfunction, but is detrimental or compromising to cost, schedule, comfort, or workload.

UNBALANCED CONFIGURATION. (18) A bus using the HDLC protocol that connects one primary and one or more secondary stations.

UNDESIRABLE RESPONSE. (11) Change of performance and output, not designated a malfunction or safety hazard, that is evaluated as acceptable as is because of minimum nuisance effects and excessive cost burdens to correct.

UNIDIRECTIONAL DATA BUS. (18) A data bus with only one user that is capable of transmitting.

UPSET. (11) Temporary interruption of performance that is self-correcting or reversible by manual or automatic process.

UPSET. (12) A condition in which the state of a digital device is unintentionally altered, but may be restored by automatic means or by operator intervention.

UPSET. (13) See system functional upset.

VALIDATION. (4,11) Demonstration and authentication that a final product operates in all modes and performs consistently and successfully under all actual

operational and environmental conditions founded upon conformance to the applicable specifications.

VALIDATION. (18) The process of evaluating whether or not items, processes, services, or documents accomplish their intended purpose in their operating environment.

VERIFICATION. (4,11) Demonstration by similarity, previous in-service experience, analysis, measurement, or operation that the performance, characteristics, or parameters of equipment and parts demonstrate accuracy, show the quality of being repeatable, and meet or are acceptable under applicable specifications.

VERIFICATION. (18) The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements.

VOTING PROCEDURE. (8) An algorithm included in fault tolerant software which uses the consensus recovery block method. It compares outputs of the n independent versions and determines which outputs are correct by identifying agreements among two or more versions.

WELL-BEHAVED FUNCTION. (17) A smooth mathematical relationship.

ACRONYMS AND ABBREVIATIONS

λ (17)	Language Level
μm (6)	Micrometer
μs (18)	Microsecond
η (17)	Vocabulary of a Program
η_1 (17)	Number of Unique Operators
η_1^* (17)	Minimum Number of Unique Operators
η_2 (17)	Number of Unique Operands
η_2^* (17)	Number of Different Input and Output Parameters
ϕM (6)	Phase Modulation
$1/E_0$ (17)	Average number of discriminations a person is likely to make for each bug introduced into the code.
3-D (16)	Three-Dimensional
A/C (11)	Aircraft
A/L (3)	Approach/Land
AAES (15)	Advanced Aircraft Electrical System
ac (3,6,12,15)	Alternating Current
AC (3,5,14,17,18)	Advisory Circular
ACAP (13)	Advanced Composite Airframe Program
ACARS (11,12)	ARC Communications Addressing and Reporting System
ACES (13)	Applied Computational Electromagnetics Society
ACK (18)	Acknowledge
ACO (18)	Aircraft Certification Office
ACS (16)	Automatic Control System
ACT (11,12)	Active Controls Technology
ACT (17)	Analysis of Complexity Tool
ADC (11,12)	Air Data Computer
ADF (11,12)	Automatic Direction Finder
ADI (3)	Automatic Direction Indicator
AE (6)	Avionics Equipment
AE4L (5,13)	SAE Subcommittee (Lightning)
AEEC (18)	Airlines Electronic Engineering Committee
AEHP (13)	Atmospheric Electricity Hazards Protection
AERA (16)	Automated En Route Air Traffic Control System
AES-S (6)	Aerospace and Electronic Systems Society
AF (11,12)	Audio Frequency
AFBW (4)	Augmented Fly-By-Wire
AFCS (11,12)	Automatic Flight Control System
AFFDL (7,8,13)	Air Force Flight Dynamics Laboratory
AFM (16)	Advanced Fuel Management
AFSC (18)	Air Force Systems Command
AFWAL (6,13)	Air Force Wright Aeronautical Laboratory
AGARD (8)	Advisory Group for Aerospace Research and Development

AHRS (6)	Attitude Heading Reference System
AI (16)	Artificial Intelligence
AIAA (5,6,9,15,18)	American Institute of Aeronautics and Astronautics
AIM (18)	Advanced Integrated MUX
AIR (18)	Aerospace Information Report
AIRLAB (5,18)	Avionics Integration Research Laboratory
AK (7)	Altitude Kinematics
ALCM (13)	Air Launched Cruise Missile
ALU (3,5,10)	Arithmetic Logic Unit
AM (5)	Amplitude Modulated
AM (6,14)	Amplitude Modulation
AMSC (5)	Document number prefix used by the Department of Defense
ANSI (11,12)	American National Standards Institute
AOA (4)	Angle of Attack
AP (18)	Application Processor
APU (11,12,15)	Auxiliary Power Unit
AR (7)	Analytical Redundancy
ARC (11,12)	Aeronautical Radio, Incorporated
ARIES (3)	Automated Reliability Interactive Estimation System
ARINC (3,6,18)	Aeronautical Radio, Incorporated
ARP (17,18)	Aerospace Recommended Practice
ARTERI (15)	Analytical Redundancy Technology for Engine Reliability Improvement
ASCB (6,18)	Avionics Standard Communications Bus
ASDS (11,12)	Airport Surface Detection System
ASEE (5,15)	American Society of Electrical Engineers
ASME (5)	American Society of Mechanical Engineers
ATCRBS (11,12,14)	Air Traffic Control Radar Beacon System
ATF (16)	Advanced Tactical Fighter
ATI (16)	Access Time Interval
ATTR (5)	Attribute
AWACS (14)	Airborne Warning and Control System
$\hat{B}$ (17)	Number of Bugs (Estimated)
B-dot (13)	Derivative of the magnetic field with respect to time
B-GLOSS (5)	Gate Logic Software Simulator developed by Bendix
BAC (18)	Balanced Asynchronous Configuration
BAT (17)	Battlemap Analysis Tool
BB (11,12)	Broadband
BC (18)	Bus Controller
BCAC (18)	Boeing Commercial Airplane Company
BCD (18)	Binary Coded Decimal
BCI (12)	Bulk Cable Injection
BFCS (18)	Beacon Frame Check Sequence
BGU (10)	Bus Guardian Unit
BIR (6)	Benchmark Information Rate
BIT (6,15,18)	Built-In Test
BITE (6,11,12)	Built-In Test Equipment
BIU (6,18)	Bus Interface Unit
BNR (18)	Binary

BOCP (18)	Bit-Oriented Communications Protocol
BP (18)	Basic Protocol
bps (6)	bits per second
BUSY (18)	Destination Busy
BW (11,12)	Bandwidth
C (7)	Comparator
C/I (5)	Communicator Interstage
CA (18)	Criticality Analysis
CAA (14)	Civil Aviation Authority
CAD (5)	Computer Aided Design
CAP (5)	Collins Application Processor
CAPS (3)	Computer Aided Production Simulator
CARE (3,5)	Computer Aided Reliability Evaluator
CARSRA (3,7)	Computer-Aided Redundant System Reliability Analysis
CAS (11,12)	Criticality Advisory System
CAST (3)	Complementary Analytic Simulative Technique
CBD (5)	Commerce Business Daily
CCITT (6)	Consultative Committee for International Telephone and Telegraph
CD (6)	Collision Detection
cdf (8)	Cumulative Density Function
CDU (11,12)	Control Display Unit
CE (11,12)	Conducted Emission
CE (17,18)	Certification Engineer
CFR (17)	Code of Federal Regulations
CM (11,12)	Common Mode
CMC (18)	Current Mode Coupler
CMOS (5,12)	Complimentary Metal-Oxide Semiconductor
CONUS (14)	Contiguous United States
CP (18)	Combined Protocol
CPA (5)	Central Processor - A
CPU (5,10,18)	Central Processing Unit
CR (5)	Contractor Report
CR (6)	Command Response
CR/LF (17)	Carriage Return/Line Feed
CRC (6,18)	Cyclic Redundancy Check
CRMI (2)	Computer Resource Management, Incorporated
CRT (11,12,16)	Cathode Ray Tube
CS (11,12)	Conducted Susceptibility
CSC (9)	Computer Software Component
CSCI (9,17)	Computer Software Configuration Item
CSDB (18)	Commercial Standard Data Bus
CSDL (5,10,15)	Charles Stark Draper Laboratories
CSMA (6,16,18)	Carrier Sensed Multiple Access
CSMA/CD (6)	Carrier Sense Multiple Access/Collision Detection
CT (2,6)	Technical Center (designation used in FAA report numbering scheme)
CTA (3)	CAPS Test Adapter
CTA (5)	Collins Test Adaptor
CTS (18)	Clear To Send

CW (13)	Continuous Wave
D (17)	Program Difficulty
DADC (6)	Digital Air Data Computer
DARPA (16)	Defense Advanced Research Projects Agency
DATAC (6,18)	Digital Autonomous Terminal Access Communication
dB (6,12)	Decibel
dBi (14)	Decibels with respect to one milliampere
dBm (6)	Decibels with respect to one milliwatt
dc (6,12,15)	Direct Current
DC (18)	Display Computer
DE (5)	Diagnostic Emulation
DEFN (5)	Definition
DET (18)	Driver Enable Timer
DEV (5)	Development
DF (7)	Diagnostic Filter
DFC (7)	Digital Flight Control
DFCS (3,4,7,16)	Digital Flight Control System
DFDAU (11,12)	Digital Flight Data Acquisition Unit
DFDR (11,12)	Digital Flight Data Recorder
DGAC (14)	Directorate Generale Aviation Civile
DISAC (15)	Digital Integrated Servo Actuator Controller
DITS (6,11,12,18)	Digital Information Transfer System
DM (11)	Differential Mode
DM (6)	Delay Modulation
DMA (5,6)	Direct Memory Access
DMA (18)	Direct Memory Addressing
DME (11,12,18)	Distance Measuring Equipment
DNA (13)	Defense Nuclear Agency
DOD (5,8,12,14,16)	Department of Defense
DOE (13)	Department of Energy
DOT (2,3,6,7,8)	Department of Transportation
DR (17)	Direct Ratio (Average)
DRB (9)	Distributed Recovery Block
DS (17)	Direct Score
DSP (3)	Discrete Switch Panel
DTSA (18)	Dynamic Time Slot Allocation
E (17)	Programming Effort
E-dot (13)	Derivative of the electric field with respect to time
E-FIELD (11,12)	Electric Field
E/E (11,12)	Electrical/Electronic
E^3 (11,12)	Electromagnetic Environmental Effects
EADI (11,12)	Electronic Attitude Director Indicator
ECAC (11,12,14)	Electromagnetic Compatibility Analysis Center
ECM (14)	Electronic Counter Measures
ECS (11,12)	Environmental Control System
EEC (5,11,12,18)	Electronic Engine Control
EED (11,12)	Electro-Explosive Device
EES (18)	Electromagnetic Emission and Susceptibility
EFID (18)	Electronic Flight Instrument Display

EFIS (11,12,18)	Electronic Flight Instrument System
EFMA (3)	Executive Failure My A
EFMB (3)	Executive Failure My B
EFOA (3)	Executive Failure Other A
EFOB (3)	Executive Failure Other B
EFW (3)	Executive Failure Word
EGT (11,12)	Exhaust Gas Temperature
EHSI (11,12)	Electronic Horizontal Situation Indicator
EIA (18)	Electronic Industries Association
EICAS (11,12)	Engine Indication and Crew Alerting System
EIU (16)	Electronic Interface Unit
EM (5,11,12,13)	Electromagnetic
EMA (15)	Electromechanical Actuator
EMAS (2,15)	Electromechanical Actuator System
EMC (6,11,12,13,14)	Electromagnetic Compatibility
EMCad™ (12)	Electromagnetic Computer aided design
EME (14)	Electromagnetic Environment
EME (11,12)	Electromagnetic Effects
EMI (5,6,11,12,13,15,16)	Electromagnetic Interference
EMIC (11,12)	Electromagnetic Interference/Compatibility
EMP (11,12,13)	Electromagnetic Pulse
EMR (5,14)	Electromagnetic Radiation
EMUX (6)	Electrical Multiplex
ENRZ (6)	Enhanced Non-Return to Zero
EOF (17)	End of File
EPR (11,12,15)	Engine Pressure Ratio
EPROM (16)	Erasable Programmable Read-Only Memory
ESD (11,12)	Electrostatic Discharge
ESE (11,12)	Electric (field) Shield Effectiveness
ESS (5,9)	Electronic Switching System
ETDL (13)	Equipment Transient Design Level
EUROCAE (14)	European Organization for Civil Aviation Electronics
EXCHNG (5)	Exchange
EXP (5)	Experiment
F/FA (18)	Fault and Failure Analysis
FAA (ALL)	Federal Aviation Administration
FADEC (6,15,16)	Full Authority Digital Engine Controller
FAFTEEC (16)	Full Authority Fault Tolerant Electronic Engine Control
FAR (3,4,6,16)	Federal Acquisition Regulation
FAR (17,18)	Federal Aviation Regulation
FBL (15,16)	Fly-By-Light
FBW (4,7,16)	Fly-By-Wire
FCC (3,4,5,6,10,11,12,15,18)	Flight Control Computer
FCR (5)	Fault Containment Regions
FCS (4,5,10,16)	Flight Control System
FCS (18)	Frame Check Sequence
FD (7)	Fault Detection
FDEP (11,12)	Flight Data Entry Panel
FDFM (15)	Fault Detection and Failure Management

FET (5,16)	Field Effect Transistor
FI (5)	Fault Insertion Circuitry
FI (7)	Fault Isolation
FIAT (5)	Fault Injection Automated Testing
FICA (15)	Failure Indication and Corrective Action
FIIS (5,10)	Fault Insertion and Instrumentation System
FIM (5)	Fault Injection Manager
FIRE (5)	Fault Injection Receptor
FM (5)	Frequency Modulated
FM (6,14)	Frequency Modulation
FMC (11,12)	Flight Management Computer
FMEA (3,9,18)	Failure Mode and Effect Analysis
FMECA (5,15)	Failure Modes and Effects Criticality Analysis
FMECA (18)	Failure Mode, Effects, and Criticality Analysis
FP (17)	Function Point
FSM (18)	Finite State Machine
ft (14)	feet
FT (5)	Fault Tolerant
FTA (18)	Fault Tree Analysis
FTC (5)	Fault Tree Compiler
FTMP (5,10,18)	Fault-Tolerant Multiprocessor
FTP (5)	Fault Tolerant Processor
G/E (13)	Graphite Epoxy
GA (18)	General Aviation
GaAs (6,16)	Gallium Arsenide
GAMA (6,18)	General Aviation Manufacturers Association
GCR (6)	Group Code Recording
GE (5)	General Electric
GEMACS (13)	General Electromagnetic Model for the Analysis of Complex Systems
GEN (5)	Generation
GGLOSS (5)	Generalized Gate-Level Logic System Simulator
GLOSS (5)	Gate Logic Software Simulator
GNC (16)	Guidance, Navigation, and Control
GPC (5)	General Purpose Computer
GPS (11,12)	Global-Positioning-System
GPWS (11,12)	Ground Proximity Warning System
Gr/Ep (11,12)	Graphite/Epoxy
GS (10)	Glideslope
H-FIELD (11,12)	Magnetic Field
H1 (11,12)	Fan Speed
HA (18)	Hazard Analysis
HARP (5,18)	Hybrid Automated Reliability Predictor
HDBK (5)	Handbook
HDLC (6,18)	High-Level Data Link Control
HERF (5,14,16,18)	High-Energy Radio Frequency
HF (11,12,13,14)	High-Frequency
HIRF (15,18)	High-Intensity Radiated Fields
HOL (17)	High Order Language
HSI (3)	Horizontal Situation Indicator

HSRB (6,18)	High-Speed Ring Bus
HVDC (15)	High Voltage dc
HW (18)	Hardware
Hz (3,15,18)	Hertz
I (13)	Current
I (17)	Intelligence Content
I-dot (13)	Derivative of the current with respect to time
I/O (5,10,15,18)	Input/Output
IAA (3)	Integrated Assurance Assessment
IAAC (11,12)	Integrated Application of Active Controls Technology (to an Advanced Subsonic Transport Project)
IACS (18)	Integrated Avionic Computer System
IBM (5)	International Business Machines
IC (3,18)	Integrated Circuit
ICAO (14)	International Civil Aviation Organization
ICIS (5)	Intercomputer Interface Sequencer
ICS (5)	Intercomputer Sequencer
ID (5)	Identification
ID (18)	Identifier
IDG (11,12)	Integrated Drive Generator
IEEE (5,6,9,17,18)	Institute of Electrical and Electronics Engineers, Incorporated
IFC (17)	Information Flow Complexity
IFCS (18)	Information Frame Check Sequence
IFF (14)	Identification - Friend or Foe
IGGLOSS (5)	Gate Logic Software Simulator (improved version developed at NASA Langley)
ILS (3,11,12)	Instrument Landing System
IMR (18)	Interrupt Mask Register
INS (6,11,12)	Inertial Navigation System Institute
IOPA (5)	Input/Output Processor - Channel A
IOS (5)	Input/Output Subsystem
IRS (11,12)	Inertial Reference System
ISO (17,18)	International Standards Organization
ITT (16)	(Consultative Committee for) International Telegraphy and Telephony
IV&V (9)	Independent Verification and Validation
IVT (18)	Interrupt Vector Table
JPL (5)	Jet Propulsion Laboratories
K (6)	Thousand
kA (13,16)	Kiloampere
kHz (5,6,12,14)	Kilohertz
km (6)	kilometer
L (17)	Program Level
$\hat{L}$ (17)	Estimated Program Level
LAN (5)	Local Area Network
LaRC (5)	Langley Research Center

LCC (11,12)	Life Cycle Cost
LCD (16)	Liquid Crystal Display
LED (6)	Light Emitting Diode
LF (13)	Low Frequency
LOC (11,12)	Localizer
LPN (13)	Lumped Parameter Network
LRC (6)	Longitudinal Redundancy Check
LRRA (11,12)	Low Range Radio Altimeter
LRU (5,6,11,12,13,18)	Line Replaceable Unit
LSB (6,18)	Least Significant Bit
LSI (5,18)	Large Scale Integration
LTPB (6,18)	Linear Token Passing Bus
LTRI (13)	Lightning and Transients Research Institute
LVDT (3)	Linear Voltage Differential Transducer
LVDT (15)	Linear Variable Differential Transformers
m (6)	meter
m (18)	Original Address of Last Transmission
M (5)	Mutual (when used with RLC)
M (6)	Million
mA (3)	Milliampere
MAADS (6)	Multibus Avionic Architecture Design Study
MAC (4)	Mean Aerodynamic Chord
MAFT (16)	Multicomputer Architecture for Fault Tolerance
Mbps (6,16)	Million bytes per second
MC (18)	Mode Code
MCDP (11,12)	Maintenance Control and Display Panel
MCFCS (18)	Message Control Frame Check Sequence
MCP (11,12)	Mode Control Panel
MDICU (3)	Modular Digital Interface Control Unit
MDICU (4,5)	Modular Digital Interface Conversion Unit
MDT (6)	Mean Down Time
MFCS (18)	Message Frame Check Sequence
Mflops (16)	Million floating-point operations per second
MFM (6)	Modified-Frequency Modulation
MFR (10)	Mean Failure Rate
MHz (5,6,12,14,18)	Megahertz
mil (11,12)	One thousandths of an inch (0.001)
MIL (5)	Military
MIL-HDBK (18)	Military Handbook
MIL-STD (6,18)	Military Standard
ML (18)	Message Length
MLE (8)	Maximum Likelihood Estimates
MLS (11,12)	Microwave Landing System
MOS (5)	Metal-Oxide Semiconductor
MPP (16)	Massively Parallel Processor
MPSC (18)	Multi-Protocol Serial Controller
MPX (5)	Multiplex
ms (6,10,18)	Millisecond
MSB (6,18)	Most Significant Bit
MSE (11,12)	Magnetic (Field) Shielding Effectiveness
MSI (5)	Medium Scale Integration

MT (18)	Message Time
MTBCF (6)	Mean Time Between Critical Failures
MTBF (9,18)	Mean Time Between Failures
MTTF (8,10)	Mean Time to Failure
MTTR (6,18)	Mean Time To Repair
Mux (5)	Multiplexed
MUX (4,18)	Multiplexer
n (18)	Address of User Performing Computation
N (17)	Implementation Length of a Program
N (18)	Maximum Number of Users
$\hat{N}$ (17)	Estimated Length
N_1 (15)	Low Rotor Speed
N_1 (17)	Total Number of Operator Occurrences
N_1^* (17)	Minimum Number of Operators
N2 (11,12)	Core Engine Speed
N_2 (15)	High Rotor Speed
N_2 (17)	Total Number of Operand Occurrences
N_2^* (17)	Minimum Number of Operands
NA (6)	Numerical Acceptance
NA (3)	Normal Accelerometers
NADC (6,7)	Naval Air Development Center
Naecon (5)	National Avionics and Electronics Conference
NAECON (6)	National Aerospace and Electronics Conference
NAND (5)	Not AND
NASA (2,3,5,7,13,15,18)	National Aeronautics and Space Administration
NASC (13)	Naval Air Systems Command
NATO (14)	North Atlantic Treaty Organization
NB (11,12)	Narrow Band Signal
NCTS (18)	Not Clear To Send
NEC (13)	Numerical Electromagnetics Code
NEMP (5,12)	Nuclear Electromagnetic Pulse
NHPP (8)	Non-Homogeneous Poisson Process
nmi (14)	nautical mile
NPRM (14)	Notice of Proposed Rulemaking
NRZ (6)	Non-return to Zero
NRZ-I (6)	Non-return to Zero Inverted
NRZ-L (6)	Non-return to Zero Dual Level
nsec (6)	Nanosecond
NSWC (13)	Naval Surface Weapons Center
NVS (8)	N-version Software
OMEGA (11,12)	Very Low Frequency Navigation
OMV (16)	Orbital Maneuvering Vehicle
OS (5)	Operating System
OSI (18)	Open Systems Interconnection
OTV (16)	Orbital Transfer Vehicle
P (10)	Processor
P-Static (11,12)	Precipitation Static
PAL (5)	Programmable Array Logic
PAS (6)	Pilot Assist System

PAWS (5)	Padé Approximation With Scaling
PBW (15)	Power-By-Wire
PC (5)	Personal Computer
PC (17)	Processing Complexity
PCA (17)	Processing Complexity Adjustment
PCS (16)	Primary Control System
PCU (11,12)	Power Control Unit
pdf (8)	Probability Density Function
PE (6)	Phase Encoding
pf (12)	picofarad
PLA (15)	Power Level Actuator
PLA (16)	Power Level Angle
PMA (18)	Parts Manufacturer Approval
PMS (5)	Physical Message Switch
PRF (11,12)	Pulse Repetition Frequency
PROC (5)	Processor
PROM (3,10,17,18)	Programmable Read-Only Memory
psi (15)	pounds per square inch
PVI (16)	Pilot/Vehicle Interface
PWM (11,12)	Pulse Width Modulation
PZ (15)	Piezoelectric
QUAD (5)	Quadruple
R (13)	Resistance
R-C (12)	Resistor-Capacitor
RADC (8,17)	Rome Air Development Center
RAE (13)	Royal Aircraft Establishment
RAM (3,5,10,16,17,18)	Random Access Memory
RAT (6,18)	Ring Admittance Timer
RB (8)	Recovery Block
RBDCP (3)	Reliability Block Diagram Computer Program
RCA (16)	Radio Corporation of America
RCVR (5)	Receiver
RDFCS (3)	Redundant Digital Flight Control System
RDFCS (4)	Reconfigurable Digital Flight Control System (facility)
RDFCS (5)	Reconfigurable Digital Flight Control System
RDMI (11,12)	Radio Distance Magnetic Indicator
RE (11,12)	Radiated Emission
REG (5)	Register
REL (3)	Reliability
REL COMP (3)	Reliability Computers
RF (5,11,12,13,14,16,18)	Radio Frequency
RFI (11,12)	Radio Frequency Interference
RIM (18)	Ring Interface Module
RIU (18)	Ring Interface Unit
RL (13,15)	Resistance/Inductance (Out of order in c.15)
RLC (5,13)	Resistance/Inductance/Capacitance
RLCM (13)	Resistance/Inductance/Capacitance/Mutual
RM (7)	Redundancy Management
RNRZ (6)	Randomized Non-return to Zero

ROM (5,10)	Read Only Memory
RPV (16)	Remotely Piloted Vehicle
RR (18)	Read Register
RRT (18)	Ring Rotation Time
RS (11,12)	Radiated Susceptibility
RS (18)	Recommended Standard
RSS (4,7)	Relaxed Static Stability
RT (5,18)	Remote Terminal
RTCA (2,3,11,12,14,16,17,18)	Radio Technical Commission for Aeronautics
RTE (18)	Real-Time Executive
RTI (5)	Remote Terminal Interface
RTS (18)	Request To Send
RZ (6)	Return to Zero
S (17)	Stroud Number
S-a-0 (5,10)	Stuck at Zero
S-a-1 (5,10)	Stuck at One
S-GLOSS (5)	Gate Logic Software Simulator developed by Stevens
S/A (11,12)	Spectrum Analyzer
SAE (2,5,6,13,14,16,17,18)	Society of Automotive Engineers
SAI (18)	Systems Architecture and Interfaces
SAS (4,7)	Stability Augmentation System
SC (18)	Special Condition
SCC (18)	System Configuration Controller
SCM (18)	Software Configuration Management
SCP (18)	*Self-Checking Pair*
SDF (7)	Super-Diagnostic Filter
SE (11,12)	Shielding Effectiveness
SEU (5)	Single Event Upset
SG (18)	Synchronization Gap
SHF (11,12)	Super High-Frequency
SHRD (5)	Shared
SIAM (15)	Society for Industrial and Applied Mathematics
SIF (14)	Selective Identification Facility
SIM (18)	Serial Interface Module
SIR (18)	Shared Interface RAM
SLRT (7)	Sequential Likelihood Ratio Test
SMF (18)	Self Monitor Function
SMOTEC (14)	Special Missions Operation Test and Evaluation Center
SO (17)	Second Order (Average)
SPRT (7)	Sequential Probability Ratio Test
SQA (18)	*Software Quality Assurance*
SQF (17)	Software Quality Factor
SQL (5)	Software Query Language
SQM (17)	Software Quality Metrics
SQPP (17)	Software Quality Program Plan
SRS (17)	Software Requirement Specification
SSA (18)	System Safety Assessment
SSI (5)	Small Scale Integration
SSP (3)	Servo Simulation Panel
SSPC (15)	Solid-State Power Controller

SSS (17)	System/Segment Specification
STANAG (14)	Standardization Agreement (NATO)
STC (2)	Supplemental Type Certification
STC (18)	Supplemental Type Certificate
STEM (5)	Scaled Taylor Expansion Matrix
STOL (16)	Short Takeoff and Landing
str (6)	string
SURE (5)	Semi-Markov Unreliability Range Evaluator
SW (18)	Software
SYN (5)	Synch
$\hat{T}$ (17)	Estimated Programming Time
T/R (6)	Transmitter/Receiver
TACAN (14)	Tactical Air Navigation
TASRA (3)	Tree Aided System Reliability Analysis
T_c (18)	Count Duration
TC (2)	Type Certification
TC (18)	Type Certificate
TCAP (13)	Threshold Circuit Analysis Program
TCAS (11,12,16,18)	Traffic Alert and Collision Avoidance System
TCL (13)	Transient Control Level
TDM (6)	Time Division Multiplex
TDMA (18)	Time Division Multiple Access
T_F (18)	Frame Time
TFCS (18)	Token Frame Check Sequence
TFEDF (18)	Token Frame Ending Delimiter Field
TG (18)	Terminal Gap
THT (6,18)	Token-Holding Timer
TI (18)	Transmit Interval
TLA (11,12)	Thrust Lever Angle
T_m (18)	Wait Time for User
TMC (11,12)	Thrust Management Computer
TMR (10)	Triple Modular Redundant
TRT (18)	Token Rotation Timer
TRU (15)	Transformer Rectifier Unit
TSDF (18)	Token Starting Delimiter Field
TSO (18)	Technical Standard Order
TTL (5,11,12,13,16)	Transistor-Transistor Logic
TV (14)	Television
TWTD (13)	Thin Wire Time Domain
TX (5)	Transmit
U.K. (13,14)	United Kingdom
U.S. (14)	United States
UAC (18)	Unbalanced Asynchronous Configuration
UART (15)	Universal Asynchronous Receiver Transmitter
UHF (11,12,13,14)	Ultra High-Frequency
UNC (18)	Unbalanced Normal Configuration
UNIBUS (5)	Universal Bus
UPS (15)	Uninterruptible Power Supplies
USAF (16)	United States Air Force
USB (16)	Upper Surface Blowing

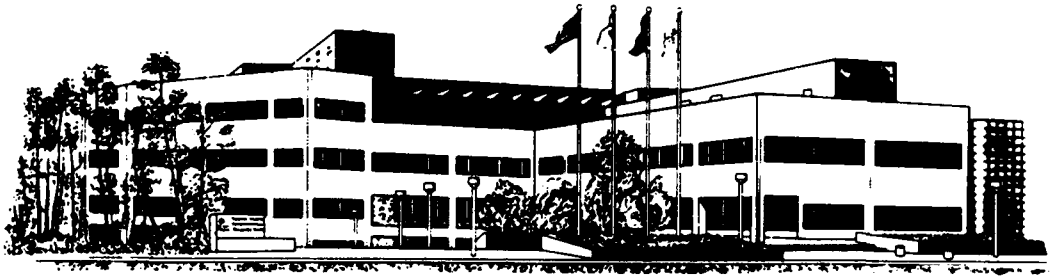
USEG (5)	Unsegmented
V (17)	Volume
V' (17)	Potential Volume
V&V (18)	Verification and Validation
V/m (14)	Volt/meter
VHF (11,12,13,14)	Very High-Frequency
VHSIC (6,16)	Very-High-Speed Integrated Circuits
VLF (11,12)	Very Low-Frequency
VLSI (5,6,14,18)	Very Large Scale Integration
VLSIC (6,16)	Very Large Scale Integrated Circuits
VOR (11,12,14,18)	VHF Omnidirectional Range
VORTA/VHF (11,12)	Omnirange/Tactical Air Navigation
VRC (6)	Vertical Redundancy Check
VSI (11,12)	Vertical Speed Indicator
VSV (15)	Variable Stator Vane
VT (18)	Validation Testing
VTOL (16)	Vertical Takeoff and Landing
W/P (15)	Fuel Flow to Burner Pressure
WAI (3)	Warning Annunciation Indicator
WFM (15)	Main fuel metering valve actuator sensor
WR (18)	Write Register
WRU (11,12)	Weapons Replaceable Unit
WSO (17)	Weighted Second Order (Average)
XAB (5)	Transmit Compare A B
XMT (5)	Transmit
XMTR (3)	Transmitter
XOR (5)	Exclusive OR
ZM (6)	Zero Modulation

U.S. Department of Transportation
Federal Aviation Administration

DOT/FAA/CT-88/10

HANDBOOK-VOLUME II
DIGITAL SYSTEMS VALIDATION

CHAPTER 17
SOFTWARE QUALITY METRICS



FEDERAL AVIATION ADMINISTRATION
TECHNICAL CENTER
ATLANTIC CITY INTERNATIONAL AIRPORT, NEW JERSEY 08405

NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The United States Government assumes no liability for the contents or use thereof.

The United States Government does not endorse products or manufacturers. Trade or manufacturers' names appear herein solely because they are considered essential to the objective of this report.

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/CT-88/10		

4. Title and Subtitle	5. Report Date
SOFTWARE QUALITY METRICS	July 1992
	6. Performing Organization Code
	8. Performing Organization Report No.
7. Author(s)	DOT/FAA/CT-88/10
N. VanSuetendael and D. Elwell	
9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
Computer Resource Management, Inc. 200 Scarborough Drive, Suite 108 Pleasantville, New Jersey 08232	11. Contract or Grant No. DTFA03-86-C-00042
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Tutorial Handbook Chapter 17
U.S. Department of Transportation Federal Aviation Administration Technical Center Atlantic City International Airport, NJ 08405	14. Sponsoring Agency Code ACD-230

15. Supplementary Notes

Pete Saraceni, FAA Technical Center, Program Manager

16. Abstract

When digital technology is employed to perform some function aboard aircraft, the designer documents the technology and the applicant presents a package to the Certification Engineer (CE). Typically, the package might include design and test specifications, test plans, and test results for the system. This package assures the CE that the designer has properly developed and validated the system. Software Quality Metrics (SQM) may be used during the software (SW) development and testing.

SQM technology attempts to quantify various quality-oriented factors, such as reliability and maintainability. The SW developer determines the quality factors that are important to the application. SW Metrics that correlate to these factors are used on the code to determine to what extent these factors have been reached. Based on the results, the developer determines whether the SW meets the requirements set for it and how well the SW will perform.

This technical report documents the results of a study conducted to analyze SQM as they apply to the code contained in avionic equipment and systems. This report is intended to provide an indepth explanation of how SQM may be applied and interpreted.

17. Key Words	18. Distribution Statement
Software Quality Metrics, Software Complexity Metrics, Program Level, Analysis, Data Structure, Effectiveness Metric, Structural Complexity	Document is available to the U.S. public through the National Technical Information Service, Springfield, VA 22161

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	264	

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

ACKNOWLEDGEMENT

The Federal Aviation Administration Technical Center and Computer Resource Management, Incorporated, wish to acknowledge Honeywell, Incorporated, Sperry Commercial Flight Systems Group of Glendale, Arizona, for their assistance in writing this document. Honeywell provided the FAA with flight control software code they developed. The code was used in evaluating and comparing different complexity metrics methodologies. In addition, Honeywell provided continuing technical support for their software during the research. All of this assistance was given at no cost to the Government.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

1. INTRODUCTION

1.1 Software Quality Metrics

Technological advances have led to smaller, lighter, and more reliable computers that are used increasingly in flight applications. Federal Aviation Administration (FAA) Certification Engineers (CEs) are faced with certification of aircraft that depend on this digital technology, and on its associated software.

When digital technology is to be used to perform some function aboard aircraft, the designer documents the technology and the applicant presents a package to the CE. Typically, the package might include design and test specifications, test plans, and test results for the system. This package assures the CE that the designer has properly developed and validated the system. Software Quality Metrics (SQM) may be used during software development and testing.

SQM technology attempts to quantify various quality-oriented factors, such as reliability and maintainability. The software developer determines the quality factors that are important to the application. Software metrics that correlate to these factors are used on the code to determine to what extent these factors have been reached. Based on the results, the developer determines whether the software meets the requirements set for it, and how well the software will perform.

If SQM results are submitted to support a system to be certified, the CE should understand SQM and their results and implications. This chapter shows the step-by-step procedures for applying major SQM to avionic code. It shows the CE how SQM are applied, analyzed, and evaluated. This chapter can therefore be used as a reference to aid the CE when certifying avionic equipment or systems where SQM were used.

1.2 Scope of this Chapter

This chapter only includes metrics that are based on code, and that apply to the types of code used in avionic equipment. Metrics that are well substantiated by empirical evidence receive an in-depth discussion. In particular, this chapter applies and analyzes SQM based on three prominent families of software metrics: Halstead's, McCabe's, and Rome Air Development Center's (RADC).

Other SQM are included, but are addressed indirectly in appendices A and B. For a full discussion on these other metrics, see the Technical Report, Software Quality Metrics (N. VanSuetendael and D. Elwell 1991).

History metrics and metrics that cover all phases of the software development process are not within the scope of this chapter. Both categories, however, do warrant further investigation. In cases where an SQM falls in either of these

categories, the analysis is limited to that portion of the SQM that addresses the code phase.

Because SQM rely on statistics to indicate which software metrics correlate with a quality factor, this chapter does not cover all possible combinations leading to good correlations. The most accepted SQM are discussed. Acceptance is based on what SQM experts have concluded from their correlational studies. Future combinations of software metrics and quality factors are also addressed, since the SQM components are presented independently in appendices A and B.

1.3 How this Chapter is Organized

Section 2 discusses the background of SQM, defines them, and defines the model presented for analyzing them.

Sections 3.1, 3.2, and 3.3 summarize the results of the study and introduce the certification problem addressed by the SQM worked examples.

Section 3.4 discusses SQM based on Halstead's software metrics as they apply to developed avionic code. The SQM are calculated and analyzed step-by-step; then their application to the certification problem is evaluated. Sections 3.5 and 3.6 follow the same format as section 3.4 using SQM based on McCabe's Software Metrics and RADC's SQM methodology, respectively.

Readers interested only in how to apply the metrics of section 3 can read the steps and skip the analysis boxes. Those readers who have an interest in analyzing SQM should read the boxes that provide analyses for each step.

Appendices A and B contain data sheets that summarize the basic data a CE needs to understand, analyze, and evaluate these and other SQM. Appendix A addresses software metrics and appendix B addresses quality factors. Future SQM may correlate the software metrics of appendix A with new quality factors. Conversely, the quality factors of appendix B may be correlated with new software metrics. These two appendices are organized so that new SQM can still be analyzed.

Appendix C contains the software metric data on which the SQM calculations are based.

2. INTRODUCTION TO METRICS

2.1 Background of Software Quality Metrics

Computers play a primary role in industry and government. Because the hardware of modern systems relies heavily on the supporting software, software can critically affect lives. Many applications, e.g., flight critical systems, mission-critical systems, and nuclear power plants, require correct and reliable software. The increased importance of software also places more requirements on it. Thus, it is necessary to have precise, predictable, and repeatable control over the software development process and product. Metrics evolved out of this need to measure software quality.

In the early 70s, Halstead maintained that an attribute of software, such as complexity, is directly related to physical entities (operators and operands) of the code, and thus could be measured (Halstead 1972 and 1977). Halstead's metrics are fairly easy to apply. They are based on mathematical formulas and are backed by over a decade of refinement and follow-on empirical study. SQM based on Halstead's software metrics compose case 1 of the worked examples.

In the mid-70s, another researcher, McCabe, devised the Cyclomatic Complexity Metric, a measure based on the number of decisions in a program. McCabe's metric has been used extensively in private industry (Ward 1989). SQM based on McCabe's software metrics compose case 2 of the worked examples.

At approximately the same time, researchers at TRW became interested in measuring software qualities (Boehm et al 1978). The TRW team was responding to earlier research on the subject by Rubey and Hartwick. This approach considered software quality factors to be based on elementary code features that are assets to software quality. This work was adopted by RADC of the Air Force Systems Command. In 1977, RADC determined that software metrics were viable tools. Software acquisition managers could use them to determine accurately whether requirements were satisfied for new, delivered systems (McCall, Richards, and Walters 1977). RADC's SQM methodology is used in case 3 of the worked examples.

SQM based on these three software metric families are widely accepted. Recent research focuses on refining some of the initial assumptions and on creating new SQM.

In developing new metrics, theory often precedes empirical evidence. Today, numerous companies are marketing SQM. Some do not back their measurement claims with much independent research or empirical substantiation. Other companies, however, have empirically validated their metrics. Many companies use their metric systems to address all phases of the software life cycle.

Metric researchers are split broadly into two camps: those who claim software can be measured, and those who state that the nature of software does not lend itself to metrical analysis. The latter group includes researchers who have been disillusioned with the field, or who have mathematically "proved" that software cannot be measured. To further split the former group, some are busy refining existing metrics, others disproving parts of them. In any case, the majority of researchers are concerned about software quality and the need to quantify it.

Metric research is aimed at reducing software cost, keeping development on schedule, and producing software that functions as intended. Additionally, metrics are being developed to evaluate whether software accomplishes its functions reliably and safely. In the present context, metrics can be used to aid in the certification of avionic equipment and systems.

2.2 Definition of Software Quality Metrics

Before SQM can be understood, the terms need to be defined. A computer-based system consists of a machine and its associated programs and documentation. Hardware consists of the physical components of a computer. Software consists of computer programs and the documentation associated with the programs.

Code is a subset of software, and exists for the sole purpose of being loaded into the machine to control it. Code that can be read by people is source code. The translation of the source code that is loaded into the machine is called object code. This chapter addresses metrics that measure aspects of source code. In this chapter, any reference to code is to be understood as a reference to source code.

As used in the term, "SQM", quality modifies software. SQM measure whether software is quality software. This measure is based on quality factors. A quality factor is any software attribute that contributes either directly or indirectly, positively or negatively, toward the objectives for the system where the software resides. A particular quality factor is defined, or singled out, when it is determined that it significantly affects those objectives. Two examples are software reliability and complexity.

Metrics is another word for measures. The most basic measure is a repeatable, monotonic relationship of numbers to a magnitude of a property to be quantified. As soon as such a relationship is defined, the one set is a measure of the other. Repeatable means anyone will get the same results at any time. Since the relationship is a monotonic function, a certain change in the measure always represents a certain change in the property being measured, where either change is simply an increase or decrease in magnitude. (Increasingly negative numbers represent a decreasing magnitude.) Although such a function is consistently non-increasing or non-decreasing, it is not proportional. In addition, it is desirable that the monotonic measure have a one-to-one correspondence to the magnitude of the measured property so that it is not possible for differing property magnitudes to give the same measure.

A more desirable measure is one where the magnitude of the property is reflected in the measure by a smooth, mathematical relationship. This measure is

considered well-behaved. The most desirable and common well-behaved measure is a linear one, since any change in the magnitude of a property is proportionally represented by the measure.

The magnitude of properties of objects can usually be measured, since a property magnitude can almost always be expressed by a repeatable, monotonic relationship. Further, these measures are usually well-behaved, and are often linear. With respect to software, the objects are the printed symbols that represent ideas. Well-behaved measures exist for these objects (see Curtis 1980, for more details on measurement theory).

If a repeatable, monotonic relationship could be established that relates measures of objects (a set of numbers) to subjective qualities, the result would be, by definition, a quality measure. SQM is the name given to a measure that relates measures of the software objects (the symbols) to the software qualities (quality factors).

All practical SQM establish rules for relating the software symbols to the software quality factors. These rules are usually expressed as well-behaved mathematical relationships. However, few of the SQM produce a completely repeatable and monotonic measure of the software quality. Nearly all SQM in use exhibit variability and lack of monotonicity in their results.

Nevertheless, practical, reliable SQM do exist. Their validity is based on the statistical inference that can be applied to systems under development, drawn from the analysis of systems in operation. Experiments must be performed to analyze how well the values produced by the SQM correlate to the quality it claimed to measure. The strength of the relationship is expressed in terms of statistical correlation or regression. Thus, each SQM has a population or a sample space for which the measurement is valid, as well as a level of confidence that can be attributed to it.

When the relationship between two randomly varying quantities is measured in terms of correlation, the correlational coefficient, ρ, indicates how close the relationship, if any, comes to being linear. It does not give any indication of the coefficients of the linear relationship. A perfectly linear positive relationship is indicated by $\rho=1$. A perfectly linear negative relationship is indicated by $\rho=-1$. A value of $\rho=0$ indicates that the two quantities are not linearly correlated. Two quantities that are not correlated are not necessarily unrelated. They may follow some nonlinear relationship (Hays and Winkler 1970). Therefore, the correlational coefficient can establish the presence of a relationship, but not the absence.

A particular SQM may claim that a software quality is linearly related to a software metric. Correlational experiments may verify this with a value of $\rho=0.9$. However, the proposed linear SQM relationship is still not necessarily valid. In order to be useful as a relative measure, the slope of the linear relationship must be confirmed. In order to be useful as an absolute measure, the zero intercept of the linear relationship must also be determined. Thus, "If the correlation significantly differs from zero and if the estimates are generally close to their actual counterparts, then one could convincingly argue the metric is valid" (Dunsmore 1984).

When the relationship between two quantities is measured by regression analysis, the relationship is addressed more directly. A slope, an intercept, and the coefficient of determination, r^2, are calculated. This coefficient is a measure of the amount of variation in the dependent variable (quality factor) that may be explained by the variation in the independent variable (software metric). "A high coefficient of determination leaves less variability to be accounted for by other factors" (Basili and Hutchens 1983).

In practice, the word "metric" is used loosely. It must be discerned from the context whether the word is short for software metric or SQM. This is not always clear, because many practitioners assume that the software qualities are directly linked with the software objects. They use the word metric as though the measure of the software object were the same as measuring the software quality. In this chapter, the measure of the software objects will be referred to as software metrics, and the measure of the quality factors as SQM.

2.3 The Software Quality Metrics Model

There are two categories of elements measured by an SQM. The software object category contains directly measurable items: lines of code, conditional statements in a program or module, and number of unique operators and operands. The software quality factor category contains qualities software should possess to some degree: reliability, maintainability, simplicity, and ease of use. The two categories contain components for creating an SQM. Either category contains only the raw elements which, by themselves, signify little. When measures of the elements in the first category are combined (or "mapped") with elements from the other, an SQM is formed.

Thus, a particular SQM consists of three components: software metrics, software quality factors, and the mapping between the two. This relationship allows scientists, programmers, and engineers to measure the quality of the software being evaluated.

Practitioners emphasize the importance of agreeing on selecting quality factors appropriate to the application. Selecting appropriate quality factors is subjective. In the worked examples, each project's needs are identified first; the quality factors are then chosen based on these needs.

This model is flexible because new SQM are produced simply by combining new or existing software metrics with new or existing quality factors. The decision whether a software measure successfully "correlates" with a quality factor is subjective and depends on the needs of the application, the company sponsoring the research, and other concerns.

2.4 Limits of Measurements

SQM can be applied to any readable set of symbols. However, the results of a particular SQM will be meaningful only when the SQM is applied to the particular subset of symbols for which it has been prequalified. Thus, a metric can be thought of as a function which has a certain domain over which it is

defined. Outside that domain, the results are unpredictable, and therefore it should not be applied.

Within the domain, the measure of a software quality is only as good as the proposed correlation of the software quality to the software object measured, as determined by experimentation. The domain is typically no larger than the sample space of the experiments, although, if a sufficient number of experiments are performed, the sample space can reliably project the properties of the population. Within the domain, when a highly-correlated SQM is applied, it typically does not measure absolute levels of the software quality.

An SQM is generally a ratio that is expressed as a unitless number. For example, a high correlation is found between software maintenance time and the number of decision statements in a program. If Program A has twice as many decision statements as Program B, the time required to maintain A probably will be double that of B (assuming previous experimental studies show a linear relationship between the two entities). This type of result cannot specify a unit, such as particular number of hours of maintenance. It is a relative measure, only measuring trends. It does not assess software quality, but improvements in quality. The improvement suggested by a doubling of an SQM may actually represent an inconsequential improvement in absolute terms. The measurement could simply indicate the software has improved from unacceptable to barely passable.

If, however, the relationship has been calibrated by making comparisons to software whose qualities are established by after-the-fact experience, then the SQM does provide an absolute measure of software quality and will have units attached to it. Such a metric can be used to assess levels of software quality.

Software metrics, in and of themselves, predict nothing. They simply measure the state of the software at the time of measurement. For example, a particular code metric may indicate that, at the time of measurement, a module contained a specific number of decisions.

On the other hand, when a software metric is correlated to a quality factor, the SQM inherently predict some future state of the software. The quality factors are only meaningful when the code is applied in a context. Code, as printed on paper, exhibits no software quality. But in the context of a programmer having to read the code in order to change it, the presence of the quality factor, maintainability, becomes evident. Thus, when an SQM measures a software quality, it predicts that when the code is viewed in that context, the code will exhibit that quality.

SQM indirectly address testing adequacy. Testing adequacy assumes that the level of software quality desired is produced by testing for failures and fixing them until the level is achieved. Clearly, SQM are helpful here. They provide a quantitative assessment of the quality level of the software produced. This assessment can be compared to the quantitative level specified. Testing is objectively adequate when the specification is fulfilled. Without SQM, testing adequacy is a qualitative decision.

Specifying test coverage requirements is one way of assuring testing adequacy. RTCA Standard DO-178A recommends certain types of testing for software in each of the criticality categories. If all the test cases required by each type are performed, testing adequacy is assured. While SQM do not address test coverage, some software metrics address test coverage by implying the number of test cases necessary. Adequate coverage would require that the count produced by these metrics be met. This would not be sufficient to establish test coverage, however, since the software metric does not address particular test cases. Other software tools are specifically designed to generate test cases.

2.5 Metrics and Standards

SQM can be used to satisfy the requirements of some standards that apply to avionics. Title 14 of the United States Code of Federal Regulations (CFR) contains the Federal laws that apply to aeronautics and space. Chapter 1, Subchapter C, Federal Aviation Regulation (FAR) Part 25, specifies the airworthiness standards for transport category airplanes. In particular, Section 25.1309 specifies that airplane systems and associated components must be designed in such a way that the probability of their failure is not excessive. Compliance with this design requirement must be demonstrated by an analysis. SQM can be used in this analysis. This airworthiness standard currently contains no Special Conditions to which SQM technology would apply.

The requirement of Section 25.1309 is stated in qualitative terms. The FAA publishes formal guidelines for meeting the requirements of the CFR. One such guideline, Advisory Circular (AC) 25.1309-1A, gives specific design and analysis procedures and techniques which can be used to meet the requirements of Section 25.1309 of the CFR. Of particular interest for SQM is that this AC assigns quantitative thresholds to the probability of failure that is acceptable. This may make SQM even more useful for demonstrating compliance with Section 25.1309. The FAA has also adopted the Society of Automotive Engineers' (SAE) Aerospace Recommended Practice (ARP) 1834 as an informal guideline for the fault and failure analysis of digital systems and equipment, as introduced in AC 25.1309-1A. Some day SQM may be incorporated into these guidelines.

The certification process consists of determining whether avionic equipment and systems comply with Section 25.1309. To support this determination, the FAA has adopted the Radio Technical Commission for Aeronautics (RTCA) standard, RTCA/DO-178A, "Software Considerations in Airborne Systems and Equipment Certification" (1985), as another informal guideline. This standard recommends several software design and analysis procedures that lend themselves to SQM application.

The software development standard, DOD-STD-2167A, incorporates the use of SQM into the software development process. The requirements for reliability, maintainability, availability, and other quality factors are to be specified in the System/Segment Specification (SSS), as required by paragraphs 10.1.5.2.5ff of the associated Data Item Description, DID-CMAN-80008A. Furthermore, the software quality standard, DOD-STD-2168, recognizes that software development tools will be used in the software quality program. The tools are to be identified in the Software Quality Program Plan (SQPP) according to the specification of paragraph 10.5.6.2 of the associated Data Item Description, DID-QCIC-80572.

SQM have become so widely used that they are now being incorporated into national and international standards. Standardization of their application and interpretation will make them more useful to the software engineering community. The Institute of Electrical and Electronics Engineers (IEEE) has published standards on several commercial software metrics in IEEE Standard 982. This standard defines and explains the use of measures that can be used to produce reliable software. Furthermore, the IEEE Computer Society has produced a draft standard on an SQM methodology for the International Standards Organization (ISO). The ISO plans to release this standard in 1991.

3. PRACTICAL SOFTWARE QUALITY METRICS

3.1 Study Results

This study of current practical SQM revealed three families of software metrics upon which most viable SQM are based. In order to be a basis for a viable SQM, these software metrics had to meet the following criteria:

1. The metric must apply to code.

2. The metric must apply to code used in avionic equipment and systems. This code is usually written in Assembly, FORTRAN, Pascal, C, or Ada. Hence, any metrics that apply to these languages were generalized to apply to avionic code. Other metrics may also apply based on their universal application to code.

3. The metric must be related to a known software quality factor. In some cases, metrics are not correlated with quality factors, but can still help programmers develop software. For example, a performance metric shows the number of seconds a program takes to execute. The measurement determines to what degree the performance deviates from an accepted value, but does not measure the quality of the program.

4. The metric must be substantiated by independent experimental evidence.

Three additional families of software metrics include promising prospects for viable SQM, but the software metrics and their relationships to software quality factors require further substantiation. Each of the most established software metrics of these six families is summarized on a Software Metric Data Sheet in appendix A. Many other software metrics were encountered, but very little has been published about them. They are not addressed here.

In addition, this study revealed numerous quality factors that represent the concerns of software developers and users. Many of these are summarized on the Quality Factor Data Sheets in appendix B.

Most importantly, this study identified practical SQM which consist of experimentally verified relationships between some of the software metrics of appendix A and the quality factors of appendix B. These SQM are listed in table 3.1-1.

The study also identified SQM which suggest a relationship between some of the software metrics of appendix A and the quality factors of appendix B, but the relationships require further substantiation. These promising SQM are listed in table 3.1-2.

TABLE 3.1-1. VIABLE SOFTWARE QUALITY METRICS

Software Metric Family	Software Metric	Quality Factor
Halstead	N	Maintainability Number of Bugs Reliability
	L	Complexity Simplicity
	V	Complexity Maintainability Number of Bugs Reliability Simplicity
	V*	Conciseness Efficiency
	I	Conciseness Efficiency
	E	Clarity Complexity Maintainability Modifiability Modularity Number of Bugs Reliability Simplicity Understandability
	$\hat{B}$	Maintainability Number of Bugs Reliability
McCabe	v(G)	Complexity Maintainability Modularity Modifiability Number of Bugs Reliability Simplicity Testability Understandability
	ev(G)	Complexity Conciseness Efficiency Simplicity
RADC	AC.1 through VS.3 (see Bowen, Wigle, and Tsai 1985)	Completeness Consistency Correctness Efficiency Expandability Flexibility Integrity Interoperability Maintainability Modularity Portability Reliability Reusability Simplicity Survivability Usability Verifiability

The appendices also contain software metrics and quality factors that are not part of any of the SQM identified in the tables. They represent components that are likely to be incorporated into subsequently proposed and developed SQM. By including them, this chapter better prepares the CE for evaluating whatever SQM an applicant may submit in a certification package.

3.2 Software Quality Metrics Application and Analysis

The CE should be able to understand, analyze, and evaluate any SQM that are presented. Three examples of the application and analysis of some of the SQM are given below. They present SQM based on each of the three most prominent families of software metrics. The examples are step-by-step worked examples of how a developer might have applied SQM to demonstrate the certifiability of some computerized avionics. They are presented as three different solutions to the same problem statement. An analysis of the application is interjected where appropriate. The analysis sections are set off from the metric application process by boxes. The data for the calculations can be found in appendix C.

TABLE 3.1-2. PROMISING SOFTWARE QUALITY METRICS

Software Metric Family	Software Metric	Quality Factor
Albrecht	Function points	Complexity Maintainability Modularity Performance Simplicity Understandability
Ejiogu	S_c	Complexity Maintainability Modularity Performance Simplicity Understandability
Henry	Information flow	Complexity Maintainability Modifiability Modularity Performance Reliability Simplicity Understandability

In the problem statement and the worked solutions of this section, the following qualifications apply:

- The context of the problem and solutions is fictitious.
- The code analyzed is real flight control code, consisting of 143 modules, written in Z80 assembly language.
- The flight control code was categorized by the developer into 59 essential modules and 84 nonessential.
- Two of the essential modules were copied, renamed, and arbitrarily categorized as critical modules, for the sake of providing a more complete example.

3.3 Worked Example Problem Statement

A flight control system submitted for certification utilizes software consisting of 145 modules, 84 of which are classified as nonessential, 59 as essential, and 2 as critical. The developer calculated SQM during the implementation of the code to measure the extent to which certain design and performance requirements

had been met (RTCA/DO-178A, paragraph 6.2.4.2). The developer also used SQM in the requirements coverage analysis and test coverage analysis (RTCA/DO-178A, paragraph 6.2.5.3). The applicant submitted the SQM results to the FAA as part of the implementation and testing assurance documentation.

From this documentation, the CE should determine whether

- the software was developed according to a disciplined approach (RTCA/DO-178A, paragraph 1.1),
- the software was developed and tested by a process appropriate to the software level (RTCA/DO-178A, paragraph 6.2.1),
- the SQM-based requirements analysis substantiates that the associated requirements are fulfilled (RTCA/DO-178A, paragraph 6.2.5.3.1),
- the SQM-based test coverage analysis substantiates test coverage (RTCA/DO-178A, paragraph 6.2.5.3.2), and
- the software will operate at the required level of performance.

Each of the worked examples demonstrates the process a developer followed to use SQM in the software development project described. Each developer used SQM that are based on a different family of software metrics. By observing the process, the CE will understand how to evaluate SQM reports submitted in certification documentation.

3.4 Case #1: Software Quality Metrics Based on Halstead's Software Metrics

In this case, the developer used Halstead's software metrics to assess the code quantitatively. To do this, the developer followed the steps described in the next section.

3.4.1 Metric Application and Analysis

Step 1. Specify the quality factors of concern and select the correlating software metrics for code in each category of criticality.

Halstead Metric Application Analysis 1
• The correlation of the selected quality factor to the selected software metric should be substantiated by previous experimentation. Otherwise the SQM is based solely on intuition.

a. This developer was only contracted to supply the software for this system; another developer supplied the hardware. Therefore, the Software Requirements Specification (SRS) required that the code be written as efficiently as possible. It was estimated that if this were done, the software would not demand more memory resources than the hardware contractor

was to supply. The developer chose the Halstead Estimated Program Level, $\hat{L}$, as indicative of code size efficiency. It was specified that all modules must be programmed to exceed a Halstead Estimated Program Level of 0.05.

Halstead Metric Application Analysis 2
• Although it was a good idea to quantify the efficiency specification, the developer did not refer to any experiments that correlated the Estimated Program Level to object code size. Furthermore, the level chosen as acceptable is based on previous experience that was not stated in the certification documentation. The CE should request further information to substantiate the validity of using this SQM.

b. Since the code was expected to have about a 10-year operational phase, maintenance would be an ongoing expense. Furthermore, it was anticipated that code improvements would be requested on a fast turn-around basis. Therefore, it was decided that a software metric that correlated to maintainability would be used for all three levels of code. The software measures, Lines of Code and Halstead's Volume, V, were chosen. Any module that exceeded the initial calculation of the average value, plus one standard deviation for its category, was to be singled out, possibly to be reduced. The Halstead metric requires all of the basic counts of operators and operands. From experience, the cost of performing these counts was always recouped when maintenance is lessened for such a long operational phase.

Halstead Metric Application Analysis 3
• The relationship of maintainability to Lines of Code and Halstead's Volume is well established by experimentation. However, the CE can accept this SQM as a reliable indicator of maintainability improvement only if the code being developed belongs to the same population of code on which the experiments have been performed. The developer properly used this SQM to indicate which modules should be improved because they would require abnormal amounts of maintenance time. The developer did not propose any improper claims of absolute times or levels of maintainability.

c. It was assumed that the effort to program each module would be a reasonable indicator of the relative effort to debug. Halstead's Programming Effort, E, was used to balance the workload during the debugging phase. Each programmer who was to debug modules was assigned an approximately equal number of Programming Effort units to debug.

Halstead Metric Application Analysis 4
• Halstead's Programming Effort metric has been shown to correlate strongly to actual debugging times. Nevertheless, because of variations in program complexity and programmer capability, the developer was right to limit its use to indicating the relative amount of effort required. The CE should be sure that the experiments substantiating such a relationship apply to the environment of this project.

d. Since no particular method is prescribed by RTCA/DO-178A for assessing the testing adequacy of essential modules, Halstead's Number of Bugs metric, $\hat{B}$, was used to determine testing adequacy. Testing was considered adequate when it uncovered the number of bugs estimated by $\hat{B}$.

Halstead Metric Application Analysis 5
• Halstead's Number of Bugs metric correlates well to the actual number of bugs in the experiments done on this topic. If those experiments apply to this code environment, use of the metric would indicate a certain amount of testing adequacy.

e. Since the critical modules would be subjected to more rigorous testing, it was of particular interest that they rate high in testability. In particular, the number of coding errors that would be discovered during testing should be minimized, so that testing could focus on functional performance. Therefore, before testing began, all critical modules had to be coded so that Halstead's predicted Number of Bugs, $\hat{B}$, was less than one. Number of Bugs was used to measure testability.

Halstead Metric Application Analysis 6
• This is an innocuous use of an SQM. It may or may not have saved the developer testing time. Either way, the quality of the code was probably not impacted. The CE can at least note that the development process was subjected to the disciplined approach of quantitative control (RTCA/DO-178A, 1.1).

The use of the Number of Bugs metric required that evaluation be performed on the past work of the programmers working on essential and critical modules. For each programmer, the typical number of discriminations per bug, E_0, was obtained from this evaluation to be used in the Number of Bugs

calculation. Furthermore, the programmers who produced bugs at a lower rate were assigned to write the modules anticipated to require more mental discriminations. The cost of the evaluation would be justified by the testing adequacy assurance and testing effort reduction that would result from reducing the number of bugs.

f. In addition to bug reduction, it was considered important that the development process was followed by all programmers of essential and critical modules. If any of these modules varied more than one standard deviation from the mean value of Halstead's Estimated Program Level metric, $\hat{L}$, it was assumed that the programmer had not rigorously followed the programming standards for that module. Compliance with this requirement was taken as a measure of how consistently the programmers followed the standards.

Halstead Metric Application Analysis 7
• This is a reasonable way of using SQM to provide a disciplined development process. Excessive variations in compliance to coding standards would result in some modules being overly-concise and others being too wordy. The Estimated Program Level metric would detect these relative extremes, although the absolute level may not have a particular meaning.

g. All essential and critical modules were to be swapped from Programmable Read-Only Memory (PROM) into a certain protected area of the system Random Access Memory (RAM) during execution. Since the development computer was often tied up building system load modules, it was decided that Halstead's Intelligence Content metric, I, would be used to indicate whether each module would result in object code that would fit in the protected area, whatever size that area would be.

Halstead Metric Application Analysis 8
• This measure is only intuitively related to object size. The relationship is based solely on the theoretical assertion that the Intelligence Content of a program does not vary when the program is translated. Experiments have not substantiated how well the Intelligence Content correlates to object size. Whether or not it does, the quality of the code was probably not impacted. A poor estimate primarily would impact the cost incurred by the developer.

h. To help ensure the integrity of the critical modules, they should be kept very simple. It was decided to limit critical modules to a maximum value of Halstead's Program Length, N. The smaller, more numerous modules that would result would incur more system overhead during run-time, but this was considered to be a worthwhile compromise. Exceptions could be granted based

on the peculiar needs of the modules that did not naturally meet this limitation.

Halstead Metric Application Analysis 9
• While the Program Length is strongly correlated to the perceived simplicity of a program, the threshold is set somewhat arbitrarily. Therefore, it is essential that those modules that exceed it receive a special review to determine if the constraint can reasonably be applied to them. The developer did well to allow exceptions to the Program Length maximum, in order to accommodate those modules for which the limitation was unreasonable.

Step 2. Specify the level of quality considered acceptable for each SQM.

a. For the code size Efficiency quality factor, the contract required 100 percent compliance to the specified Estimated Program Level as the acceptable level of quality for all levels of code.

b. For the Maintainability quality factor, the developer required that at least 75 percent of the modules not exceed the limit, in either Lines of Code or Volume. The values of the two software metrics for each failing module gave an idea of what to fix to achieve compliance.

c,d. The Programming Effort metric and the Number of Bugs in the essential modules were not correlated to quality factors so no goals were set for them. They were simply used by managers to control and improve software development.

e. For the Testability quality factor, the developer required 100 percent compliance of the critical modules to the Number of Bugs threshold.

f. For the Consistency quality factor, the developer required 95 percent compliance. This applied only to the essential and critical modules.

g. The Intelligence Content was not correlated to a quality factor so no goal was set for it. It was simply used by managers to control the size of the product.

h. For the Integrity quality factor, the developer required 98 percent compliance by the critical modules.

With these preliminary selections, development began. After the code was written, the following metric steps were taken.

Step 3. Define the rules for determining that a word of source code is an operator, as defined on the data sheet of appendix A. The rules may be contained in a lookup table or may consist of rule statements. A word that does not satisfy the rules for an operator is taken to be an operand.

Halstead Metric Application Analysis 10
• The occurrence of an operator that was not anticipated by the rules will inflate the operand count and deflate the operator count. For instance, if the operator, CALL, occurs, but was not put in the lookup table, it will be counted as an operand. • An operator manifested in a way that was not anticipated by the rules will inflate the operand count and deflate the operator count. For instance, if the operator, CALL C, occurs, but was only put in the lookup table in the form, CALL, it will be counted as an operand. • An operand that accidentally satisfies the rules will inflate the operator count and deflate the operand count. This would only occur if the rules were less stringent than the language processor. • The rules are necessarily different for every computer language or language variation. They are very closely tied to the rules used in the interpreter, compiler, or assembler. It is assumed that the code has been translated, without error, by one of them. Code from a different language processor will produce unreliable results. • Different people will probably define the rules differently. The various sets of rules may produce similar results most, but not all of the time.

For this case, the software metric tool, PC-Metric, was used to calculate Halstead's software metrics. PC-Metric uses a reserved word file as a lookup table of the words which are operators. In addition, other rules are followed. Since the theory of operators and operands is based on algorithm implementation, it does not apply to program comments and declarations. Thus, PC-Metric filters out comments, and declarative keywords are put in the keyword file and flagged that they are not to be counted. Typical entries in the reserved word file used by PC-Metric to calculate Halstead's metrics on Z80 Assembly code would be as follows:

```
2 (
0 )
2 *
2 +
2 ,
2 -
2 /
2 ADC
2 ADD
2 AND
1 ASEG
```

where the "2" flags a countable operator, the "0" flags that the pair of parenthesis operator is counted when the left parenthesis is encountered, and the "1" flags ASEG as a declaration word.

Halstead Metric Application Analysis 11
• The calculations produced by a software metric tool reflect all the user-defined rules and all the decisions built into the program code. The only way to ensure a repeatable, predictable measure for comparison to some previous measure, is to maintain configuration control of the software metric tool and of the code being analyzed. Then, comparisons can be safely made only between two measurements that claim to be produced by, for example, PC-Metric, Version 1.3, on code successfully assembled by the Z80 Assembler, Version 2.6. To further ensure the accuracy of the measurements, it is essential to use the software metric tool to analyze a test case for which the values can be manually calculated for comparison. This will ensure that the proper attribute is measured, and is measured accurately.

Step 4. Define the segment of code that constitutes a module.

Halstead Metric Application Analysis 12
• A module can be defined trivially as a single statement or as the next smallest independent function, or a subroutine, program, subsystem, or entire system of programs. In general, the larger the module is defined, the smaller will be the proportion of the number of unique words to the total number of words.

A file was taken to be a module, since each file constituted a partition categorized, for certification purposes, as nonessential, essential, or critical. In general, this meant that a module was the same as an Assembly language subroutine. However, many files contained internal subroutines and/or additional external subroutines. In all cases, these were lumped together as a single module. The code consisted of 145 modules, thus defined.

Step 5. Define the rules for detecting the beginning and ending of each module.

Halstead Metric Application Analysis 13
• A module whose ending is not delineated, or is delineated in a way other than as specified in the rules, might be combined with the next module. This combination would inflate the counts for the first module and would cause the following module to go undetected. • A module whose beginning is not delineated, or is delineated in a way other than as specified in the rules, might be combined with the previous module or might be ignored. In either case, it goes undetected. • If any of the code accidentally satisfies the rules for indicating the beginning or ending of a module, the counts will likely be inaccurate. • The rules are necessarily different for every computer language or language variation. They are very closely tied to the rules used in the interpreter, compiler, or assembler. It is assumed that the code has been translated, without error, by one of them. Code from a different language processor will produce unreliable results. • Different people will probably define the rules differently. The various sets of rules may produce similar results most, but not all of the time.

The first line of each file was assumed to be the beginning of the module. The end was generated by the EOF (End-of-File). There was, therefore, little ambiguity for meeting the chosen module definition.

Step 6. Segregate the modules into categories of nonessential, essential, and critical so that metric results can be used to aid the different development processes required for each category of code.

Following the guidelines of RTCA/DO-178A, the developer divided the 145 modules into 84 nonessential, 59 essential, and 2 critical.

Step 7. For each module, count the total number of operator occurrences, N_1, and the total number of operand occurrences, N_2. Also count the number of unique operators and the number of unique operands, η_1 and η_2, respectively. The data sheets of appendix A give the definitions of each of these quantities. Also, accumulate the measures for all modules in each category of criticality so that metric results can be used to aid the different development processes required for each category of code.

Halstead Metric Application Analysis 14
• The rules used to determine whether an operator or operand is unique must take into account the possibility that a word of code is to be interpreted as a different operator or operand, as determined by the context. Whenever functionally unique operators or operands are represented by the same word of code, their respective count is deflated. For instance, BASIC usually uses a comma as an argument separator, but in parts of a PRINT statement, it is also a format specifier. • The rules used to determine whether an operator or operand is unique must take into account the possibility of a single operator or operand having more than one form. In this case, two slightly varying forms of the same operator or operand will be taken as the occurrence of two unique words, thereby inflating the unique count. This occurs in BASIC when a variable name may have up to 20 characters, but only the first eight are recognized. The full name and the first eight characters constitute two acceptable forms of the same operand. • The values, η_1 and η_2, are not additive quantities. Adding the counts of unique operators or operands for sub-modules does not give accurate counts for the module that they constitute.

PC-Metric accounted for most of the problems associated with the first point of analysis, mentioned above, by distinguishing between comments, declarations, and executable code. Within executable code, all unique operators and operands in a module are almost always constrained to be unique in form by the interpreter, compiler, or assembler.

In addition, PC-Metric had to distinguish between an asterisk in the first character position as designating a comment, but in subsequent positions as being the multiplication operator.

The second point of analysis in the previous analysis box was addressed by the "0" flag used on the right parenthesis. This flag indicates that a right parenthesis is not a unique operator, but is simply part of the left-right parenthesis pair.

The counts and the derived measures for each module of the code are listed in appendix C. The counts for each category of code are shown in table 3.4-1.

TABLE 3.4-1. OPERATOR AND OPERAND COUNTS, BY CODE CATEGORY

	Nonessential	Essential	Critical
η_1	129	157	28
η_2	1,418	1,059	51
N_1	15,141	11,393	205
N_2	11,602	8,676	160

Step 8. Calculate the various derived measures.

a. Calculate the Vocabulary, η, as follows:

$$\eta = \eta_1 + \eta_2 \text{ words}$$

Halstead Metric Application Analysis 15
• The vocabulary is not an additive quantity.

The values for each category of code were as follows:

Nonessential:

$$\eta = 129 + 1,418$$
$$= 1,547 \text{ words}$$

Essential:

$$\eta = 157 + 1,059$$
$$= 1,216 \text{ words}$$

Critical:

$$\eta = 28 + 51$$
$$= 79 \text{ words}$$

b. Calculate the Implementation Length, N, as follows:

$$N = N_1 + N_2 \text{ words}$$

The values for each category of code were as follows:

Nonessential:

$$N = 15,141 + 11,602$$
$$= 26,743 \text{ words}$$

Essential:

$N = 11,393 + 8,676$
$= 20,069$ words

Critical:

$N = 205 + 160$
$= 365$ words

c. Alternatively, if the number of unique operators and operands is known, the Estimated Length, $\hat{N}$, can be calculated without counting all the occurrences of operators and operands. It is calculated as follows:

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \text{ words}$$

Halstead Metric Application Analysis 16
• This estimate assumes that every combination of operators and operands of length η occurs only once in any program. This means that repeated segments of code, as long as η words, are put into subroutines rather than in redundant code. If such redundancy is present in the code, the Implementation Length will be underestimated by the Estimated Length Equation. The Estimated Length Equation also assumes that operators and operands alternate without variation. Any code that does not follow this convention will likely underestimate the Implementation Length. Whether or not these assumptions are reasonable, experiments show that $\hat{N}$ yields a close estimate of the Implementation Length, N. • Because the Estimated Length Equation is nonlinear, adding the Estimated Lengths of sub-modules would not be expected to give an accurate Estimated Length for the module that they constitute. Nevertheless, experiments indicate that adding the lengths does give a fairly accurate estimate.

The values for each category of code were as follows:

Nonessential:

$\hat{N} = 129 \log_2(129) + 1,418 \log_2(1,418)$
$\approx (129 \times 7.011) + (1,418 \times 10.470)$
$\approx 904 + 14,846$
$\approx 15,750$ words

Essential:

$$\hat{N} = 157 \log_2(157) + 1{,}059 \log_2(1{,}059)$$
$$\approx (157 \times 7.295) + (1{,}059 \times 10.0485)$$
$$\approx 1{,}145.3 + 10{,}641.4$$
$$\approx 11{,}787 \text{ words}$$

Critical:

$$\hat{N} = 28 \log_2(28) + 51 \log_2(51)$$
$$\approx (28 \times 4.807) + (51 \times 5.672)$$
$$\approx 135 + 289$$
$$\approx 424 \text{ words}$$

d. Calculate the Volume, V, as follows:

$$V = N \log_2 \eta \text{ bits}$$

Halstead Metric Application Analysis 17
• This metric calculates the number of binary digits required to uniquely represent all of the operators and operands that occur in a module of Length, N, and Vocabulary, η. It is not primarily a measure of the size or complexity of the function programmed, but a measure of the size of a particular implementation of the function. • Volume is not an additive quantity.

The values for each category of code were as follows:

Nonessential:

$$V = 26{,}743 \log_2(1{,}547)$$
$$\approx 26{,}743 \times 10.5952575$$
$$\approx 283{,}348.971 \text{ bits}$$

Essential:

$$V = 20{,}069 \log_2(1{,}216)$$
$$\approx 20{,}069 \times 10.2479275$$
$$\approx 205{,}665.657 \text{ bits}$$

Critical:

$$V = 365 \log_2(79)$$
$$\approx 365 \times 6.30378$$
$$\approx 2{,}300.88 \text{ bits}$$

e. Calculate the Potential Volume, V^*, as follows:

$$V^* = (2 + \eta_2^*)\log_2(2 + \eta_2^*) \quad \text{bits}$$

where η_2^* is the sum of the number of unique inputs and outputs for the module.

Halstead Metric Application Analysis 18
• This metric calculates the number of binary digits required to represent the function of a module in its most efficient form, namely two operators (i.e., one that says "do the function", and another that groups the operands with the operator) and all the operands that the function requires. • Potential Volume is not an additive quantity.

The number of inputs and outputs for the code is not known, so no calculation of Potential Volume can be made.

f. Calculate the unitless ratio, Program Level, L, as follows:

$$L = \frac{V^*}{V}$$

Halstead Metric Application Analysis 19
• This metric calculates how efficiently the module is implemented. As the Vocabulary and/or Implementation Length decrease (s), the Program Level increases, i.e., the module is written at a higher level because each word carries more weight. • The Program Level is not an additive quantity.

The Potential Volume could not be calculated, so no direct calculation of Program Level was made.

g. Alternatively, if the number of unique inputs and outputs is not known, the Estimated Program Level, $\hat{L}$, can be calculated from the basic counts, as follows:

$$\hat{L} = \frac{2\ \eta_2}{\eta_1\ N_2}$$

The values for each category of code were as follows:

Nonessential:

$$\hat{L} = (2 \times 1{,}418)/(129 \times 11{,}602)$$
$$= 2{,}836/1{,}496{,}658$$
$$\approx 0.00189488848$$

Essential:

$$\hat{L} = (2 \times 1{,}059)/(157 \times 8{,}676)$$
$$= 2{,}118/1{,}362{,}132$$
$$\approx 0.00155491538$$

Critical:

$$\hat{L} = (2 \times 51)/(28 \times 160)$$
$$= 102/4{,}480$$
$$\approx 0.0227679$$

h. Calculate the Intelligence Content, I, as follows:

$$I = \frac{2\ \eta_2}{\eta_1\ N_2} \times (N_1 + N_2)\log_2(\eta_1 + \eta_2) \quad \text{bits}$$

Halstead Metric Application Analysis 20
• This metric simply calculates the product of the Estimated Program Level (how efficiently the module is implemented) and Volume (the size of the implementation) of a module. It represents that constant quantity of information that is present in any implementation of a particular function, in any language, at any level. This quantity is basically an estimated Potential Volume of the program; I and V^* can often be used interchangeably. • Intelligence Content is not an additive quantity.

The values for each category of code were as follows:

Nonessential:

$$I = 0.00189 \times 283{,}349$$
$$\approx 536.9 \text{ bits}$$

Essential:

$$I = 0.00155 \times 205{,}666$$
$$\approx 319.8 \text{ bits}$$

Critical:

$I = 0.02277 \times 2{,}301$
≈ 52.4 bits

i. Calculate the Programming Effort, E, as follows:

$E = V/L$ discriminations

Halstead Metric Application Analysis 21
• This metric calculates the number of elementary mental discriminations done by a programmer to reduce a preconceived algorithm to a module of source code in a language in which the programmer is fluent. It is based on the assumption that, when writing a program, a programmer selects each word of the program by mentally searching a list of words from which to choose. Specifically, the programmer performs a mental binary search of the vocabulary of η words in order to select the N words used in the implementation. Furthermore, each comparison (or mental discrimination) in the selection process requires an effort related to the difficulty of understanding the program. The program difficulty is supplied by the reciprocal of the Program Level. This relationship indicates that the greater the Volume, or the lower the Program Level, the greater the effort required to write a program. • Programming Effort is not an additive property.

Using $\hat{L}$ for L, the values for each category of code were as follows:

Nonessential:

$E = 283{,}348.971/0.00189488848$
$\approx 149{,}533{,}323$ discriminations

Essential:

$E = 205{,}665.657/0.00155491538$
$\approx 132{,}268{,}070$ discriminations

Critical:

$E = 2{,}300.88/0.0227679$
$\approx 101{,}058$ discriminations

j. Calculate the Estimated Programming Time, $\hat{T}$, as follows:

$$\hat{T} = \frac{E}{3600S} = \frac{\eta_1 N_2(\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2)\log_2 \eta}{7{,}200\eta_2 S} \text{ hours}$$

Halstead Metric Application Analysis 22
• This metric calculates the time it takes to make the number of elementary mental discriminations indicated for a program by the Programming Effort. The Stroud Number, S, is the total number of elementary mental discriminations done per second by a person. This calculation assumes that the programmer's attention is entirely undivided and that the range of values found from psychological experimentation, $5<S<20$, applies to programming activity. • Estimated Programming Time is not an additive quantity.

Assuming a value for S of 18, the values for each category of code were as follows:

Nonessential:

$\hat{T}$ = 149,533,323/(3600 x 18)
≈ 2,308 hours

Essential:

$\hat{T}$ = 132,268,070/(3600 x 18)
≈ 2,041 hours

Critical:

$\hat{T}$ = 101,058/(3600 x 18)
≈ 1.6 hours

k. Calculate the Language Level, λ, as follows:

$$\lambda = L^2V$$

Halstead Metric Application Analysis 23
• This metric calculates the efficiency with which algorithms can be implemented in a particular language. For any module written in the particular language, the Language Level should be a constant, regardless of the Volume of the module or the Program Level at which it is written.

Using $\hat{L}$ for L, the values for each category of code were as follows:

Nonessential:

$$\lambda = (0.00189)^2 \times 283,349$$
$$= 1.02$$

Essential:

$$\lambda = (0.00155)^2 \times 205,666$$
$$= 0.50$$

Critical:

$$\lambda = (0.02277)^2 \times 2,301$$
$$= 1.19$$

1. Calculate the Number of Bugs, $\hat{B}$, as follows:

$$\hat{B} = V/E_0 \text{ bugs}$$

where E_0 is the number of discriminations per bug. E_0 is determined by an evaluation of a programmer's previous work.

Halstead Metric Application Analysis 24
• Experimentation has found that 3,200 discriminations per bug is a typical value.

The values for each category of code were as follows:

Nonessential:

$$\hat{B} = 283,349/3,200$$
$$\approx 89 \text{ bugs}$$

Essential:

$$\hat{B} = 205,666/3,200$$
$$\approx 64 \text{ bugs}$$

Critical:

$$\hat{B} = 2,301/3,200$$
$$\approx 0.7 \text{ bugs}$$

Step 9. Calculate the SQMs defined in step 1 of this case.

a. Calculate the Efficiency quality factor, as follows:

$$\text{Efficiency} = \frac{\text{\# of modules with } \hat{L} > 0.05}{\text{total number of modules}} \text{ \% compliance}$$

Halstead Metric Application Analysis 25
• This metric is not a measure of efficiency, but the extent to which the desired efficiency was present. This makes it a measure of quality.

The value for the code can be calculated from the data in appendix C.

Efficiency = 42/145
= 29.0% compliance

Since 100 percent compliance was specified, the development process clearly needed refining, if the code size efficiency goal was to be met.

b. Calculate the Maintainability quality factor, as follows:

$$\text{Maintainability} = \frac{\text{\# of modules that met the goal}}{\text{total number of modules}} \text{ \% compliance}$$

The value for the code can be calculated from the data in appendix C. (The noncompliant data are flagged with an asterisk.)

Maintainability = 119/145
= 82.1% compliance

Thus, the maintainability goal was met on the first check. Furthermore, the 26 noncompliant modules were reviewed and several of them were reduced in size.

c. Based on the total Programming Effort units for the code, the 28 programmers were assigned as follows:

Nonessential: 15 programmers

Effort per programmer = 149,533,323/15
≈ 10.0 million units

Essential and critical: 13 programmers

Effort per programmer = 132,369,128/13
≈ 10.2 million units

This assignment gave less than a three percent variation in programmer load. Similar calculations were made to determine how many modules to assign to each programmer.

d. Generally, for each module of essential code, testing was performed until the predicted number of bugs was found. If less than one bug was predicted, no testing was done. For some modules predicted to have bugs, none were found. These were tested until path coverage was obtained.

e. Calculate the Testability, as follows:

$$\text{Testability} = \frac{\#\text{ of modules with } \hat{B} < 1}{\text{total \# of modules}} \text{ \% compliance}$$

The value for the critical modules of the code can be calculated from the data in appendix C.

Testability = 2/2
= 100% compliance

f. Calculate the Consistency, as follows:

$$\text{Consistency} = \frac{\#\text{ of modules with } \hat{L} \text{ within } \sigma}{\text{total \# of modules}} \text{ \% confidence}$$

The initial value for the essential and critical modules of the code can be calculated from the data in appendix C. (The noncompliant data are flagged with an asterisk.)

Consistency = 54/61
= 88.5% compliance

Four modules were refined to meet the 95 percent compliance required.

g. The Intelligence Content metric indicated that the module, ESSENT34, would result in the largest object module. Therefore, the protected area of memory was made 10 percent larger than the actual size of this module's

object code. This size successfully contained all the other essential and critical object modules.

b. Calculate the Integrity as follows:

$$\text{Integrity} = \frac{\text{\# of modules with } N < 250}{\text{total \# of modules}} \text{ \% compliance}$$

The value for the critical modules of the code can be calculated from the data in appendix C.

$$\text{Integrity} = 2/2$$
$$= 100\% \text{ compliance}$$

3.4.2 Certification Conclusions

For certification purposes, it should be determined whether the metrics, as applied, were indeed reliable indicators of the properties measured. It should also be determined whether such measures equivalently fulfill the requirements of the certification guidelines. This determination can be based on the experience of the developer, the experimentation performed by the developer, or the experimentation that the developer references, as documented in the certification package. It can also be based on the experience of the CE or on reference by the CE to the handbook data sheets.

Based on these inputs, the SQM performed are evaluated to see if they help the CE answer any of the certification determinations posed in section 3.3. This evaluation addresses each SQM in the order in which they were specified in step 1.

a. The code size efficiency SQM was used to substantiate that the software requirements had been met. However, the relationship of $\hat{L}$ to object code size has not been established by experimentation. Furthermore, it has not been established that a program written with an Estimated Program Level of 0.05 is considered a very efficient program, or that the much lower values measured indicate that the modules are very inefficient.

Halstead Metric Application Analysis 26
• The CE would likely conclude that this SQM does not fulfill the requirements analysis recommended by RTCA/DO-178A, paragraph 6.2.5.3.1, for essential and critical software. This SQM does not establish whether the efficiency requirement was or was not met.

b. The use of the Maintainability SQM indicates that the developer subjected the code to a responsible, quantitatively controlled development process.

Halstead Metric Application Analysis 27
• First, the CE should compare this software development project to those of the experiments on Maintainability. The CE would then likely conclude that this SQM offers some assurance that the software was developed by a process appropriate to the software level, as recommended by RTCA/DO-178A, paragraph 6.2.1.

c. Since the debugging load was balanced using the Programming Effort software metric, the developer maintains that it is more likely that the modules are consistently debugged and that they operate properly. Otherwise, some programmers would have been overloaded and their search for bugs would have been truncated in order to keep or schedule.

Halstead Metric Application Analysis 28
• First, the CE should compare this software development project to those of the experiments on debugging times. The CE would then likely conclude that this SQM offers some assurance that the software will operate at the required level of performance.

d. Since the testing of essential modules was regulated with the Number of Bugs metric, the developer maintains that the modules are likely to have fewer bugs than if this criterion had not been established.

Halstead Metric Application Analysis 29
• First, the CE should compare this software development project to those of the experiments on the number of bugs found during testing. The CE could then conclude that this SQM offers some assurance that the software will operate at the required level of performance. However, this metric is not as assuring as others since it claims to be an absolute, rather than a relative measure.

e. Since the critical modules met the Number of Bugs threshold, the developer maintains that they are more easily tested than the other modules. Testing will not be inhibited by undetected coding errors.

Halstead Metric Application Analysis 30
• First, the CE should compare this software development project to those of the experiments on the number of bugs found

during testing. The CE could then conclude that this SQM offers some assurance that the critical code has been subjected to a more demanding development process. However, this metric is not as assuring as others since it claims to be an absolute, rather than a relative measure. Particularly, the critical code satisfied the absolute level, as coded. It is possible that the nonessential and essential code would also have satisfied the absolute level, as coded.

f. The developer used the Estimated Program Level to regulate the consistency of the essential and critical modules with the programming standards. Thus, their consistency was checked but the consistency of the nonessential modules was not. Furthermore, this check resulted in improvements.

Halstead Metric Application Analysis 31
• First, the CE should compare this software development project to those of the experiments on Estimated Program Level. The CE would then likely conclude that this SQM offers some assurance that the software was developed by a process appropriate to the software level, as recommended by RTCA/DO-178A, paragraph 6.2.1. Regardless of the applicability of the experiments, the SQM resulted in inconsistencies being detected and corrected.

g. The Intelligence Content metric was not used in a way that relates to any certification concerns.

h. Since the critical modules met the Program Length threshold, the developer maintains that they have more integrity than the other modules.

Halstead Metric Application Analysis 32
• First, the CE should compare this software development project to those of the experiments on Program Length. The CE could then conclude that this SQM offers some assurance that the critical code has been subjected to a more demanding development process. However, this metric is not as assuring as others since it was used as an absolute measure, rather than a relative measure. Particularly, the critical code satisfied the absolute level, as coded. It is possible that the nonessential and essential code would also have satisfied the absolute level, as coded.

This developer could not use SQM to substantiate test coverage, as defined in RTCA/DO-178A, paragraph 6.2.5.3.2, because Halstead's software metrics do not measure anything that substantiates structural test coverage.

In this example, the developer used both software metrics and SQM to analyze the flight control code being developed. The metrics were applied in various ways and for various purposes. In all cases, the metrics provided a quantitative analysis of the software product. This analysis was used to assess, and sometimes improve, the software development product and process.

3.5 *Case #2: Software Quality Metrics Based on McCabe's Software Metrics*

In this case, the developer used McCabe's software metrics to assess the code quantitatively. To do this, the developer followed the steps described in the next section.

3.5.1 Metric Application and Analysis

Step 1. Specify the quality factors of concern and select the correlating software metrics for code in each category of criticality.

McCabe Metric Application Analysis 1
• The correlation of the selected quality factor to the selected software metric should be substantiated by previous experimentation, otherwise the SQM is based solely on intuition.

a. This developer was concerned that the code developed would be easy to maintain. Therefore, a software metric that correlated to maintainability was used for all three levels of code. The software measure, Cyclomatic Complexity, was chosen. Any module that exceeded the initial calculation of the average value, plus one standard deviation for its category, was to be singled out, possibly to be reduced. Since this metric is easily calculated by a software tool, it was well worth the effort to calculate it for all modules.

McCabe Metric Application Analysis 2
• The relationship of maintainability to McCabe's Cyclomatic Complexity measure is well established by experimentation. However, the CE can accept this SQM as a reliable indicator of maintainability improvement only if the code being developed belongs to the same population of code on which the experiments have been performed. The developer properly used this SQM to indicate which modules should be improved because they would require abnormal amounts of maintenance time. The developer did not propose any improper claims of absolute times or levels of maintainability. Since the threshold is set somewhat arbitrarily, it is essential that those modules that exceed it receive a special review to determine if the constraint can reasonably be applied.

b. Since the Cyclomatic Complexity of each module is a reasonable indicator of the relative number of errors to debug, this measure was used to balance the workload during the debugging phase. Each programmer who was to debug modules was assigned an approximately equal number of Cyclomatic Complexity units to debug.

McCabe Metric Application Analysis 3
• McCabe's Cyclomatic Complexity metric has been shown to correlate strongly to the number of errors found in program modules (Walsh 1979; and Henry, Kafura, and Harris 1981). Nevertheless, because of variations in program size and programmer capability, the developer was right to limit its use to indicating the relative amount of debugging effort required. The CE should be sure that the experiments substantiating such a relationship apply to the environment of this project.

c. Since critical modules require structural coverage analysis for assurance of testing adequacy, the developer chose to perform a branch coverage analysis to determine the test cases to use. The Cyclomatic Complexity of each module was used to specify a number of test cases that would ensure branch coverage. The developer recognized that it is sometimes possible to satisfy branch coverage with even fewer test cases, but the ease of calculating v(G) justified its use.

McCabe Metric Application Analysis 4
• The Cyclomatic Complexity Metric does give a number of test cases sufficient to ensure branch coverage but does not guarantee that the developer chose the proper test cases. RTCA/DO-178A, paragraph 6.2.5.3.2, recommends using structured coverage analysis to ascertain which test cases constitute testing adequacy of the elements of critical modules. This SQM cannot be accepted as an equivalent (RTCA/DO-178A, paragraph 1.1) to the full analysis required, since it only establishes the number of cases, not the validity of each case. The CE can at least note that the development process is being subjected to the disciplined approach of quantitative control (RTCA/DO-178A, paragraph 1.1).

d. Since the critical modules would be subjected to more rigorous testing, it was of particular interest that they rate high in testability. In particular, it was desired to minimize the number of coding errors that would be discovered during testing so that testing could focus on functional

performance. Therefore, before testing began, all critical modules had to be coded so that the Cyclomatic Complexity was no more than 10. Cyclomatic Complexity was used to measure testability.

McCabe Metric Application Analysis 5
• This SQM is a particularly useful one. A study found that not only was the number of errors positively correlated to v(G), but that modules with v(G) greater than 10 showed a disproportionate number of errors compared to those with v(G) less than 10 (Walsh 1979). For the AEGIS Naval Weapon System software, the absolute measure, v(G)=10, was a threshold for error prone modules.

e. The developer reasoned that the module size limitation and the testing proposed were insufficient to uncover the complicated exceptions to simple program control flow that results from unstructured programs. The developer chose to measure the proportion of structured programming in the critical modules with the Essential Complexity metric, and to use this as an indicator of the module integrity. Not only would these modules be limited to 10 branch conditions, but of the 10, the developer required that a minimum of half of them be structured. This requires an Essential Complexity of five, at most.

McCabe Metric Application Analysis 6
• The Essential Complexity certainly measures the amount of structure in the module control flow, but how strongly it correlates to program integrity has not been established. The developer based this choice on intuition. The CE should require that the developer substantiate this relationship.

Step 2. Specify the level of quality considered acceptable for each SQM.

a. For the Maintainability quality factor, the developer required that at least 80 percent of the modules not exceed the limit.

b,c. No quality goals were set for the debugging effort or the number of test case metrics. These metrics were used as inputs into the management of the software development process, rather than as a specification for the product.

d. For the Testability quality factor, the developer required 100 percent compliance of the critical modules to the Cyclomatic Complexity threshold.

e. For the Integrity quality factor, the developer required 100 percent compliance of the critical modules to the Essential Complexity threshold.

With these preliminary selections, development began. After the code was written, the following metric steps were taken.

Step 3. Define the rules for identifying and counting program control flow conditions. The rules may be contained in a lookup table or may consist of rule statements.

McCabe Metric Application Analysis 7
• The predicate portion of a conditional branch of program control flow may contain more than one condition. The conditions must be identified and counted, not just the predicates. • Unconditional branching does not contribute to the count of program control flow conditions. • The occurrence of a conditional branch keyword that was not anticipated by the rules will deflate the condition count. For instance, if a CALL-on-carry-true occurs, but CALLC was not put in the lookup table, this branch condition will not be counted. • A conditional branch manifested in a way that was not anticipated by the rules, will deflate the condition count. For instance, if a CALL-on-not-zero-true may be abbreviated to CALLN, but the lookup table only contains the expanded form, CALLNZ, an occurrence of the branch condition in the form CALLN will not be counted. • In Assembly language, a jump to an indirectly addressed pointer constitutes a case structure which has many branch conditions. The number of unique modifications made to the referenced memory location determines the number of branch conditions produced in this manner. • A keyword that accidentally satisfies the rules will inflate the condition count. This would only occur if the rules were less stringent than the language processor. • The rules are necessarily different for every computer language or language variation. They are very closely tied to the rules used in the interpreter, compiler, or assembler. It is assumed that the code has been translated, without error, by one of them. Code from a different language processor will produce unreliable results. • Different people will probably define the rules differently. The various sets of rules may produce similar results most, but not all of the time.

For this case, the software metric tool, PC-Metric, was used to calculate McCabe's software metrics. PC-Metric uses a reserved word file as a lookup table of which keywords delineate conditional branch predicates. In addition, other rules are followed. Since this metric is based on program control flow, it does not apply to program comments and declarations. Thus, PC-Metric filters out comments, and declarative keywords are put in the keyword file and flagged that they are not to be counted. Typical entries in the reserved word file used by PC-Metric to calculate McCabe's metrics on Z80 Assembly code would be as follows:

```
1 ASEG
1 DEFB
A CALL
3 CALLC
3 CALLZ
A JP
4 JP(IX)
3 JPC
3 JPZ
A RET
3 RETC
3 RETZ
```

where the "3" flags a keyword/condition combination that delineates a statement with a single branch condition, the "4" flags a keyword/condition combination that delineates a statement with multiple branch conditions, the "A" flags an unconditional variant of a keyword that often delineates a statement containing a conditional branch predicate, and the "1" flags declaration words.

A case structure is the only compound predicate that Z80 code contains. PC-Metric maintains an Extended Complexity Count for these statements. The Extended Complexity count is equal to the Cyclomatic Complexity count, which counts all the branching statements, plus the number of those branching statements that contain multiple branch conditions.

McCabe Metric Application Analysis 8
• The calculations produced by a software metric tool reflect all the user-defined rules and all the decisions built into the program code. The only way to ensure a repeatable, predictable measure for comparison to some previous measure, is to maintain configuration control of the software metric tool and of the code being analyzed. Then, comparisons can safely be made only between two measurements that claim to be produced by, for example, PC-Metric, Version 1.3, on code successfully assembled by the Z80 Assembler, Version 2.6. To further ensure the accuracy of the measurements, it is essential to use the software metric tool to analyze a test case for which the values can be manually calculated for comparison.

This will ensure that the proper attribute is measured, and is measured accurately.

Step 4. Define the segment of code that constitutes a module.

McCabe Metric Application Analysis 9
• A module can be defined trivially as a single statement, or as the next smallest independent function, or a subroutine, program, subsystem, or the entire system of programs. In general, the larger the module is defined, the larger the value of the Cyclomatic Complexity. • The Cyclomatic Complexity measure requires that every statement can be reached from the beginning of the module and that flow continues from each statement to the end of the module (McCabe 1976). When this is not the case, the segment of code includes more than one independent module.

A file was taken to be a module, since each file constituted a partition categorized, for certification purposes, as nonessential, essential, or critical. In general, this meant that a module was the same as an Assembly language subroutine. However, in some cases, files contained internal subroutines and/or additional external subroutines. In all cases, these were lumped together as a single module. The code consisted of 145 modules, thus defined.

Step 5. Define the rules for detecting the beginning and ending of each module.

McCabe Metric Application Analysis 10
• A module whose ending is not delineated, or is delineated in a way other than as specified in the rules, might be combined with the next module. This combination would inflate the count for the first module and would cause the following module to go undetected. • A module whose beginning is not delineated, or is delineated in a way other than as specified in the rules, might be combined with the previous module or might be ignored. In either case, it goes undetected. • If any of the code accidentally satisfies the rules for indicating the beginning or ending of a module, the count will likely be inaccurate. • The rules are necessarily different for every computer language or language variation. They are very closely tied to the rules used in the interpreter, compiler, or assembler. It

is assumed that the code has been translated, without error, by one of them. Code from a different language processor will produce unreliable results.

- Different people will probably define the rules differently. The various sets of rules may produce similar results most, but not all of the time.

The first line of each file was assumed to be the beginning of the module. The end was generated by the EOF. There was, therefore, little ambiguity for meeting the chosen module definition.

Step 6. Segregate the modules into categories of nonessential, essential, and critical so that metric results can be used to aid the different development processes required for each category of code.

Following the guidelines of RTCA/DO-178A, the developer divided the 145 modules into 84 nonessential, 59 essential, and 2 critical.

Step 7. For each module, count the number of program control flow conditions. Also, accumulate the measures for all modules in each category of criticality so that metric results can be used to aid the different development processes required for each category of code.

McCabe Metric Application Analysis 11
• The rules used to detect conditional branches must take into account the possibility that the same word of code is to be interpreted as a different operation, as determined by the context. For instance, the equal sign is used in conditions, but it is also used in assignment statements. • It has been shown that counting control flow conditions does not produce a strictly monotonic relationship to subjective judgments of control flow complexity (Myers 1977). For instance compound conditions result in a simpler control flow than do multiple simple ones. Myers' Cyclomatic Complexity Interval measure quantifies complex conditions more accurately than McCabe's measure, but the Interval measure has not achieved much of a following.

PC-Metric accounted for most of the problems associated with the first point of analysis mentioned above, by distinguishing between comments, declarations, and executable code. Within executable code, all unique keywords in a module are nearly always constrained to be unique in form by the interpreter, compiler, or assembler.

The Z80 code presented an additional problem: the conditional branch keywords can also be used in an unconditional form. This required that PC-Metric look

for the keyword and a following condition, before concluding that a conditional branch had been detected.

PC-Metric did not accurately count the case structure branches since each occurrence of an indirect jump was simply flagged by an additional increment to the extended complexity count. The number of cases were not counted.

Furthermore, PC-Metric had to distinguish between an asterisk in the first character position as designating a comment, but in subsequent positions as being the multiplication operator.

Step 8. Calculate the following derived measures.

a. Calculate the Cyclomatic Complexity, v(G), for a single module of code, as follows:

$$v(G) = \pi + 1$$

where π is the number of program control flow conditions.

McCabe Metric Application Analysis 12
• This metric calculates the maximum number of linearly independent paths required to form a basis set for all the paths in a structured program. It is assumed that the program has an additional path from its last statement back to its first statement. Any possible path through the program can be expressed as a linear combination of some subset of this basis set of paths (McCabe 1976). • A set of program path test cases cannot claim path coverage if it does not consist of at least the number of paths in the basis set. If it does, it still may not be a basis set. Although a basis set provides path coverage on only a partial path basis, at least the set of partial paths is finite. Full path coverage assumes that every possible path, from the beginning to the ending of the module, has been generated. A program with loops can have an infinite number of such paths (Prather 1983). • This measure counts all conditions present, whether necessary or not. For example, code may contain a condition in a context where it will always be true. The Cyclomatic Complexity counts the number of conditions present in the algorithm, as implemented. • This formula for structured programs requires that for every branching node there is exactly one collecting node, and that the program has unique entry and exit nodes. Exceptions to the rules always increase v(G). As long as there are relatively few exceptions to these rules, the Cyclomatic Complexity will be not

be substantially affected. If there are many exceptions, the impact of multiple exits can be corrected by computing the following:

$$v(G) = \pi - s + 2$$

where "s" is the number of exits from the module (Harrison 1984). Of course, counting the number of exits requires additional analysis. If the code is primarily structured, the inaccuracy of the original equation is probably not significant enough to warrant the extra effort.

• The Cyclomatic Complexity of a collection of independent modules, as in a main program and its subroutines, is simply the sum of the Cyclomatic Complexities for each module. It is improper to calculate the total Cyclomatic Complexity by analyzing all the modules as if they were one continuous sequence of code. In this case, the value is deflated by the number of modules in the collection minus one.

The Cyclomatic Complexity for each module of the code is listed in appendix C. The Cyclomatic Complexity for each category of code is shown in table 3.5-1. PC-Metric calculated these totals in the manner just warned against in the last point of Analysis Box 12. This can be seen by comparing the critical code total to the individual counts. Thus, the values in table 3.5-1 are understated by the number of modules in the category minus one.

TABLE 3.5-1. CYCLOMATIC COMPLEXITY, BY CODE CATEGORY

	Nonessential	Essential	Critical
v(G)	821	730	13

b. The Essential Complexity, ev(G), can be calculated from the control graph of a prograr as follows:

$$ev(G) = v(G) - m$$

where "m" is the number of proper subgraphs, those structured subgraphs which contain no further subgraphs.

McCabe Metric Application Analysis 13
• This is a measure of the amount of unstructured code in a program. The smaller the value of ev, the greater the proportion of structured code. The Essential Complexity of a

structured program is one. Greater values for the Essential Complexity of a module indicate the size of the non-structured portion of the code; that code that does not conform to the proper subgraph structure (McCabe 1976).

• Like the Cyclomatic Complexity, this measure also reflects the state of the algorithm, as implemented. It does not indicate whether the conditions, as implemented, are necessary to the solution of the problem that the algorithm claims to solve.

From the program control graphs in appendix C, it was determined that the first module contained one proper subgraph and the second module contained four. The values for the critical modules are shown in table 3.5-2.

TABLE 3.5-2. ESSENTIAL COMPLEXITY DATA

	v(G)	m	ev(G)
CRITI001	6	1	5
CRITI002	8	3	5

Step 9. Calculate the SQM defined in step 1 of this case.

a. Calculate the Maintainability quality factor, as follows:

$$\text{Maintainability} = \frac{\text{\# of modules that met the goal}}{\text{total number of modules}} \text{ \% compliance}$$

The value for the code can be calculated from the data in appendix C. (The noncompliant data are flagged with an asterisk.)

Maintainability = 125/145
= 86.2% compliance

Thus, the maintainability goal was met on the first check. The 20 noncompliant modules were reviewed, and some were reduced by separating some of the code into independent modules.

b. Based on the total Cyclomatic Complexity units for the code, the 21 programmers were assigned modules to debug, as follows:

Nonessential modules: 11 programmers

Complexity units = 821/11
= 74.6 units per programmer

Essential and critical modules: 10 programmers

Complexity units = 743/10
= 74.3 units per programmer

This assignment gave less than a one percent variation in programmer load. Similar calculations were made to determine how many modules to assign to each programmer.

c. The number of test cases required by McCabe's Cyclomatic Complexity for ensuring branch coverage was generally reasonable. The highest of the average values for each of the three categories was 14. However, 20 had values that exceeded 22. For these, alternate methods for determining the required number of test cases were used.

d. Calculate the Testability, as follows:

$$\text{Testability} = \frac{\text{\# of modules with } v(G) \leq 10}{\text{total \# of modules}} \text{ \% compliance}$$

The value for the critical modules of the code can be calculated from the data in appendix C.

Testability = 2/2
= 100% compliance

e. Calculate the Integrity, as follows:

$$\text{Integrity} = \frac{\text{\# of modules with } ev(G) \leq 5}{\text{total \# of modules}} \text{ \% compliance}$$

The value for the critical modules of the code can be calculated from the data in table 3.5-2.

Integrity = 2/2
= 100% compliance

3.5.2 Certification Conclusions

For certification purposes, it should be determined whether the metrics, as applied, were reliable indicators of the properties measured. It should also be determined whether such measures equivalently fulfill the requirements of the certification guidelines. This determination can be based on the experience of the developer, the experimentation performed by the developer, or the experimentation that the developer references, as documented in the certification package. It can also be based on the experience of the CE or on references by the CE to the handbook data sheets.

Based on these inputs, the SQM performed are evaluated to see if they help the CE answer any of the certification determinations posed in section 3.3. This

evaluation addresses each SQM, in the order in which they were specified in step 1.

a. The use of the Maintainability SQM indicates that the developer subjected the code to a disciplined, quantitatively controlled development process.

McCabe Metric Application Analysis 14
• First, the CE should compare this software development project to those of the experiments on Maintainability. The CE would then likely conclude that this SQM offers some assurance that the software was developed according to a disciplined approach, as recommended by RTCA/DO-178A, paragraph 1.1.

b. Since the debugging load was balanced using the Cyclomatic Complexity metric, the developer maintains that it is more likely that the modules are consistently debugged and that they operate properly. Otherwise, some programmers would have been overloaded and their search for bugs would have been truncated in order to keep on schedule.

McCabe Metric Application Analysis 15
• First, the CE should compare this software development project to those of the experiments on debugging times. The CE would then likely conclude that this SQM offers some assurance that the software will operate at the required level of performance.

c. Since the testing of critical modules was regulated by the Cyclomatic Complexity, the developer maintains that branch coverage has been achieved. Otherwise, it would be possible to test an insufficient number of cases.

McCabe Metric Application Analysis 16
• The CE would likely conclude that this SQM offers some assurance that test coverage is obtained, as recommended by RTCA/DO-178A, paragraph 6.2.5.3.2. However, this metric gives the CE no assurance that the developer picked unique and comprehensive test cases. The developer may have performed the specified number of test cases, but this SQM does not ensure that the test cases used were a comprehensive set.

d. Since the critical modules met the Cyclomatic Complexity threshold, the developer maintains that they are more testable than the other modules. Testing will not be inhibited by undetected coding errors.

McCabe Metric Application Analysis 17
• First, the CE should compare this software development project to those of the experiments on testing time. The CE could then conclude that this SQM offers some assurance that the critical code has been subjected to a more demanding development process, appropriate to the software level, as recommended by RTCA/DO-178A, paragraph 6.2.1. However, this metric is not as assuring as others since it claims to be an absolute measure, rather than a relative measure. Particularly, the critical code satisfied the absolute level, as coded. It is possible that the nonessential and essential code would also have satisfied the absolute level, as coded.

e. Since the critical modules met the Essential Complexity threshold, the developer maintains that they have more integrity than the other modules.

McCabe Metric Application Analysis 18
• First, the CE should compare this software development project to those of the experiments on Essential Complexity. The CE could then conclude that this SQM offers some assurance that the critical code has been subjected to a more demanding development process, appropriate to the software level, as recommended by RTCA/DO-178A, paragraph 6.2.1. However, this metric is not as assuring as others since it was used as an absolute measure, rather than a relative measure. Particularly, the critical code satisfied the absolute level, as coded. It is possible that the nonessential and essential code would also have satisfied the absolute level, as coded.

McCabe's metrics did not address whether any of the requirements stated for this software were fulfilled. Thus, the developer did not use SQM in the requirements analysis recommended by RTCA/DO-178A, paragraph 6.2.5.3.1.

In this example, the developer used both software metrics and SQM to analyze the flight control code being developed. In either case, the metrics provided a quantitative analysis of the software product. This analysis was used to assess and sometimes improve the software development product and process.

3.6 Case #3: Software Quality Metrics Based on RADC's Software Metrics

In this case, the developer used SQM of the type developed by the RADC to assess the code quantitatively. In particular, the methodology developed by Bowen, Wigle, and Tsai (1985) was used. The developer followed the steps described in the next section.

3.6.1 Metric Application and Analysis

Step 1. Define the groups of modules that have different quality factor requirements and specify the quality factors of concern for each, based on the functions that each group supports.

RADC Metric Application Analysis 1
• The SQM developed by RADC are based on a predefined set of user-oriented quality factors. From this set, the developer selects the factors which apply to each group of modules. These factors have been carefully defined to produce a comprehensive set of relatively independent factors. Furthermore, the SQM that use them have been shown to give meaningful, repeatable results in experiments. However, the CE can accept these SQM as reliable only if the code being developed belongs to the same population of code on which the experiments have been performed. Otherwise, the SQM relationships are based solely on intuition. • The quality factors are not totally independent of one another. Some are complementary. A factor that is complemented by another but selected by itself will result in a false indication of the total quality present. All factors complementary to one that was specified should also be specified.

For this avionic code project, the developer chose to lump all the code into one group, since it all contributes to the single function: flight control. The developer specified that Maintainability, Expandability, and Reusability were to be measured for all the code. Because the developer was using SQM for the first time, the more crucial factors, Reliability, Correctness, and Verifiability, were assured by conventional testing strategies, rather than by SQM. Since the strength of the factors complementary to Maintainability, namely Reliability and Correctness, was also assured, the SQM results for Maintainability would be meaningful.

Maintainability was considered important because flight control code is usually modified many times during the lifetime of the aircraft. Expandability was important because the same code was to be used in several models of the same aircraft, some of which required many additional features. Reusability was important because reusing code is profitable.

Step 2. Specify the level of quality considered acceptable for each factor selected for each group, based on the importance of that factor for the particular system function served, a survey of important factors, the factor interrelationships, and the impact on costs.

RADC Metric Application Analysis 2
• Some of the factors have negative interrelationships. When these are selected, high levels cannot be achieved for all of them; the levels must be balanced. This compromise results in lower quality factor scores than would be otherwise attainable. • When factor scores are specified and calculated, numeric scores are used. However, when the quality goals are chosen, they should be based on a qualitative approximation of the importance of each factor. For instance, one must choose whether the code must have Excellent, Good, or Acceptable Reliability. These are the qualitative levels recommended for the Air Force. They are correlated to the numeric scores as follows (Lasky, Kaminsky, and Boaz 1990): Excellent ≥ 0.9 Good ≥ 0.8 Acceptable ≥ 0.7

The level of quality considered acceptable varied for the software criticality categories. The developer made the specifications shown in table 3.6-1.

TABLE 3.6-1. QUALITY FACTOR GOALS, BY CODE CATEGORY

Quality Factor	Nonessential	Essential	Critical
Maintainability	Excellent	Excellent	Good
Expandability	Excellent	Excellent	Good
Reusability	Acceptable	Good	Acceptable

The Maintainability quality factor was especially important for the non-critical code, since it is subjected to the most revision. This quality factor was not as important for critical code. Critical code is usually subjected to the most thorough testing, and therefore requires less maintenance. It receives fewer enhancements because the certification process for critical code is so rigorous that a developer avoids making any changes which would require recertification.

The Expandability quality factor was important because it was planned that the code would be used in all the models of the particular aircraft. It was especially important for the non-critical code to be expandable because most of the enhancements were to be made to non-critical code.

The Reusability quality factor was important for all the code because reusing code saves money. This more demanding goal was set for essential code which must be reusable with a minimum number of modifications, since modifications to

essential code must all be recertified. In general, critical code must be written from scratch for each application. There is less demand that it be reusable.

RADC Metric Application Analysis 3
• These three factors are complementary to one another. Therefore, it was reasonable for the developer to expect to attain all the quality goals specified. • The CE should note that the development process is being subjected to the disciplined approach of quantitative control for the purpose of ensuring a process commensurate with the code criticality, as required by RTCA/DO-178A.

Step 3. Identify the criteria assigned to each selected quality factor and specify a weighting factor for each, based on the importance of the criterion for the particular function served, the interrelationships of the criterion with the selected factors, and the impact on costs.

RADC Metric Application Analysis 4
• The criteria are a predefined set of software characteristics assigned to the quality factors to which they contribute. These criteria have been carefully defined to produce a comprehensive set of relatively independent criteria for each factor. They have also been shown to give meaningful, repeatable results when used in SQM experiments. The developer assigns a weighting factor, from zero to one, to multiply by the score of each criterion. A weight of zero is assigned to those criteria that are not to be counted. The sum of the weighting values for each factor must be one. • The criteria are not totally independent of other factors. Some may contribute to more than one of the selected factors. These may each be weighted lower so that one characteristic does not contribute too heavily to the total quality. Some may conflict with other factors. These may be weighted lower so that the quality goal for each factor can be attained, based on the strength of other criteria.

The Maintainability criteria, with the specified weighting factors, are shown in table 3.6-2.

TABLE 3.6-2. MAINTAINABILITY CRITERIA

Criterion	Weight
Consistency	0.0
Visibility	0.0
Modularity	0.5
Self-Descriptiveness	0.5
Simplicity	0.0

The Expandability criteria, with the specified weighting factors, are shown in table 3.6-3.

TABLE 3.6-3. EXPANDABILITY CRITERIA

Criterion	Weight
Augmentability	0.0
Generality	0.6
Virtuality	0.0
Modularity	0.2
Self-Descriptiveness	0.2
Simplicity	0.0

The Reusability criteria, with the specified weighting factors, are shown in table 3.6-4.

TABLE 3.6-4. REUSABILITY CRITERIA

Criterion	Weight
Application Independence	0.0
Document Accessibility	0.0
Functional Scope	0.3
Generality	0.3
Independence	0.0
System Clarity	0.0
Modularity	0.2
Self-Descriptiveness	0.2
Simplicity	0.0

Each criterion was of equal relevance to the nonessential, essential, and critical code. Therefore, the function of the code was not a factor in determining the weights. Other concerns determined the specified weights.

In order not to complicate the learning process, the developer chose the minimum set of criteria that would exercise the different ways that criterion scores are combined. To reduce the cost of calculating the metrics, criteria that were based on easily automatable metrics were favored. Other criteria were assigned a zero weight.

Generality and Self-Descriptiveness were chosen because they represent criteria that are the most cost-effective to measure; they contribute to more than one factor. Modularity was chosen for that reason, and also because it represents a metric that is based on a combination of metric elements. Functional Scope was chosen as representative of criteria that do not contribute to any other factor.

Modularity and Self-Descriptiveness were weighted less heavily for the Expandability and Reusability factors so that those factors could be distinguished from Maintainability by the other criteria. This also keeps these criteria from having too large an impact on the overall quality. Since none of the chosen criteria exhibit noticeable interrelationships with the other factors measured, there was no reason to otherwise weight the criteria unevenly.

RADC Metric Application Analysis 5
• This weight specification is a fairly reasonable one. It is especially important that the developer recognized the need to distinguish the factors from each other by emphasizing the unique criteria for each. It would have been even better if the number of unique criteria had been increased to match or exceed the number of criteria common to other factors. The developer also simplified the weighting process by using criteria that did not negatively impact any of the measured factors.

Step 4. Identify the metrics assigned to each criterion and specify which metric elements are to be measured, based on the applicability of the metric element to the particular system and the cost of measuring the metric element.

RADC Metric Application Analysis 6
• The metrics are a predefined set of software-oriented details assigned to the criterion to which they contribute. These metrics have been carefully defined to produce a set of independent metrics for each criterion. They have also been shown to given meaningful, repeatable results when used in SQM in experiments. The developer chooses to use those that apply and that will not cost more to measure than is gained from doing so.

- Metric elements must all be defined in the same sense, such that an increase in any metric score produces an increase in the associated quality factor score.

- Because the factor scores are based on these metric questions, the SQM goals can be used to improve the software. If a goal is not met, simply fix the individual metric element failures to achieve a higher quality.

The metric elements to be calculated were selected from Metric Worksheets 4A and 4B (Bowen, Wigle, and Tsai 1985). Each criterion is usually based on many metrics, and each metric is usually based on several metric elements. In this case, only those metric elements that were easily automated were selected. Thus, the metric, MO.1, was based on two elements and SD.2, GE.1, and FS.1 were based on only one. These metric elements are listed in table 3.6-5. With these preliminary selections, development began. After the code was written, the following metric steps were taken.

TABLE 3.6-5. SELECTED METRIC ELEMENTS

Criterion	Metric Element Number and Description
Modularity	MO.1(3) - Are the estimated lines of source code for this unit 100 lines or less, excluding comments?
	MO.1(7) - Is control always returned to the calling unit when execution is completed?
Self-Descriptiveness	SD.2(1) - Are there prologue comments which contain all information in accordance with the established standard?
Generality	GE.1(1) - How many units are called by more than one other unit?
Functional Scope	FS.1(2) - Is a description of the function(s) provided in the comments?

Step 5. Define the rules for identifying and counting metric element objects and for calculating factor scores. The rules may be contained in a lookup table or may consist of rule statements.

RADC Metric Application Analysis 7
• The occurrence of a metric element object that was not anticipated by the rules will deflate the element count. For instance, if a comment count detects comment lines but not in-line comments, the measure of the amount of commenting would be deflated. • A metric element object manifested in a way that was not anticipated by the rules will deflate the element count. For instance, if a nesting indentation check required that each level be indented with three spaces, but the code was consistently indented with a tab of three spaces, the code indentation would not be counted. • Code that accidentally satisfies the rules for detecting a metric element object will inflate the element count. This would only occur if the rules were less stringent than the language processor. • The rules are necessarily different for every computer language or language variation. They are very closely tied to the rules used in the interpreter, compiler, or assembler. It is assumed that the code has been translated, without error, by one of them. Code from a different language processor will produce unreliable results. • Different people will probably define the rules differently. The various sets of rules may produce similar results most, but not all of the time.

For this case, the developer wrote a program to calculate the previously selected RADC metric elements for the Z80 Assembly code. All the rules were imbedded in the program statements. None were in a table or otherwise user-defined.

RADC Metric Application Analysis 8
• The calculations produced by a software metric tool reflect all the decisions built into the program code. The only way to ensure a repeatable, predictable measure, for comparison to some previous measure, is to maintain configuration control of the software metric tool and of the code being analyzed. Then, comparisons can be safely made only between two measurements that claim to be produced by, for example, RADC-Metric, Version 1.3, on code successfully assembled by the Z80 Assembler, Version 2.6. It is essential to use the software metric tool to analyze a test case for which the values can be manually calculated for

comparison. This check will ensure that the proper attribute is measured accurately.

Step 6. Define the segment of code that constitutes a module.

RADC Metric Application Analysis 9
• A module can be defined trivially as a single statement, or as the next smallest independent function, or a subroutine, program, subsystem, or the entire system of programs. These metrics can be applied at any level, but the detail of the calculations makes their application prohibitive at the lower levels. For code developed under DOD-STD-2167, it is not recommended that they be applied any lower than the level of the Computer Software Configuration Items (CSCI) (Lasky, Kaminsky, and Boaz 1990).

A file was taken to be a module, since each file constituted a partition categorized, for certification purposes, as nonessential, essential, or critical. In general, this meant that a module was the same as an Assembly language subroutine. However, in some cases, files contained internal subroutines and/or additional external subroutines. In all cases, these were lumped together as a single module. The code consisted of 145 modules, thus defined.

Step 7. Define the rules for detecting the beginning and ending of each module.

RADC Metric Application Analysis 10
• A module whose ending is not delineated, or is delineated in a way other than as specified in the rules, might be combined with the next module. This combination would inflate the metric for the first module and cause the following module to go undetected. • A module whose beginning is not delineated, or is delineated in a way other than as specified in the rules, might be combined with the previous module or might be ignored. In either case, it goes undetected. • If any of the code accidentally satisfies the rules for indicating the beginning or ending of a module, the metric will likely be inaccurate. • The rules are necessarily different for every computer language or language variation. They are very closely tied to the rules used in the interpreter, compiler, or assembler. It is assumed that the code has been translated, without error, by

one of them. Code from a different language processor will produce unreliable results.

- Different people will probably define the rules differently. The various sets of rules may produce similar results most, but not all of the time.

The first line of each file was assumed to be the beginning of the module. The end was generated by the EOF. There was, therefore, little ambiguity for meeting the chosen module definition.

Step 8. Segregate the modules into categories of nonessential, essential, and critical code so that metric results can be used to aid the different development processes required for each category of code.

Following the guidelines of RTCA/DO-178A, the developer divided the 145 modules into 84 nonessential, 59 essential, and 2 critical. These three categories were treated as CSCIs and each module was taken to be a unit, as defined by this methodology.

Step 9. Compute all the metrics for each module. Also, accumulate the measures for all modules in each category of criticality so that metric results can be used to aid the different development processes required for each category of code.

RADC Metric Application Analysis 11
• The rules used to detect a particular metric element object must take into account the possibility that the same word of code is to be interpreted differently, as determined by the context. For instance, the equal sign is used in conditions, but it is also used in assignment statements.

The custom program accounted for most of the problems associated with the above analysis by distinguishing between comments, declarations, and executable code. Within executable code, all unique keywords in a module are nearly always constrained to be unique in form by the interpreter, compiler, or assembler.

Furthermore, the program had to distinguish between an asterisk in the first character position as designating a comment, but in subsequent positions as being the multiplication operator.

The metric element values for each category of code were calculated as follows:

$$\text{Metric Element} = \frac{\text{the sum of the unit scores}}{\text{the number of applicable units in the CSCI}}$$

where each unit is scored with either a calculated value normalized to be between zero and one, or with a one for a yes answer and a zero for a no answer. The values are shown in table 3.6-6. They were calculated from the data in appendix C.

TABLE 3.6-6. METRIC ELEMENT SCORES, BY CODE CATEGORY

Metric	Nonessential	Essential	Critical
MO.1(3)	45/84 = 0.5	28/59 = 0.5	2/2 = 1.0
MO.1(7)	75/84 = 0.9	48/59 = 0.8	2/2 = 1.0
SD.2(1)	44/84 = 0.5	1/59 = 0.0	0/2 = 0.0
GE.1(1)	11/84 = 0.1	11/59 = 0.2	0/2 = 0.0
FS.1(2)	84/84 = 1.0	56/59 = 1.0	2/2 = 1.0

The metric scores can then be calculated form the element scores of table 3.6-6. The Modular Implementation metric, MO.1, is calculated as follows:

$$\text{MO.1} = \frac{\text{the sum of the numerators of MO.1(3) and MO.1(7)}}{\text{the sum of the denominators of MO.1(3) and MO.1(7)}}$$

The scores for the other metrics are simply equal to the value of the single metric elements composing each. The values for each category of code are shown in table 3.6-7.

TABLE 3.6-7. METRIC SCORES, BY CODE CATEGORY

Criterion	Nonessential	Essential	Critical
MO.1	120/168 = 0.7	76/118 = 0.6	4/4 = 1.0
SD.2	0.5	0.0	0.0
GE.1	0.1	0.2	0.0
FS.1	1.0	1.0	1.0

RADC Metric Application Analysis 12

- When a metric is based on several metric elements, several methods can be used to combine the metric element scores into a metric score, according to the following procedures and considerations:

1. Direct ratio scoring adds all the numerator scores and divides this sum by the sum of the denominator scores. This

has the effect of treating every individual score of every metric element equally. If one of the composing metric element objects occurs much more frequently than the others, (i.e., the denominator is much larger) then the metric score primarily reflects that metric element. The others contribute insignificantly to the metric score.

This attribute of direct ratio scoring is helpful if it draws attention to a single frequently occurring code weakness among many code features that are consistently and properly implemented. It is harmful if a single frequently occurring code strength masks many code features that are not consistently or properly implemented. The CE should examine the data to determine if any negative effects have occurred which would mask the true quality.

When each metric element has the same value for the denominator, the value calculated by this method is the same as that calculated by second order averaging.

2. Second order averaging adds all the metric element scores together and divides the sum by the number of metric elements. This has the advantage of giving each metric element an equal contribution to the metric score. If one of the composing metric elements is especially weak or strong, it will still have an effect on the average, but not in proportion to the number of occurrences of the code feature measured.

   This attribute of second order averaging is helpful if only the percentage of errors is important. It is harmful if the number of errors of a certain type is important. The CE should examine the data to determine if any of the negative effects have occurred, masking the true quality.

3. Weighting factors can be introduced to tailor the impact a particular metric element has on the metric score. Each metric element is multiplied by a weight, between zero and one, before the summation is performed. Normalization is achieved by assuring that the sum of the weights is one. This can be applied to either of the two types of averaging. When weighting factors are used, the CE should examine the weights applied to each metric element to determine if the quality is measured accurately.

Step 10. Calculate the criterion scores, as follows:

The scores for the criteria, Modularity, Self-Descriptiveness, Generality, and Functional Scope, are simply equal to the value of the single metric composing each. The values for each category of code are shown in table 3.6-8.

TABLE 3.6-8. CRITERIA SCORES, BY CODE CATEGORY

Criterion	Nonessential	Essential	Critical
MO	0.7	0.6	1.0
SD	0.5	0.0	0.0
GE	0.1	0.2	0.0
FS	1.0	1.0	1.0

RADC Metric Application Analysis 13
• The methods of combining metric element scores to get a metric score, as discussed above, also apply to combining metric scores to get a criterion score.

Step 11. Calculate the SQM selected in step 1, and defined in step 3, as follows:

Maintainability = (0.5 x MO) + (0.5 x SD)

Expandability = (0.6 x GE) + (0.2 x MO) + (0.2 x SD)

Reusability = (0.3 x FS) + (0.3 x GE) + (0.2 x MO) + (0.2 x SD)

The values for each category of code are shown in table 3.6-9.

TABLE 3.6-9. SOFTWARE QUALITY METRICS SCORES, BY CODE CATEGORY

Quality Factor	Nonessential	Essential	Critical
Maintainability	0.6	0.3	0.5
Expandability	0.3	0.2	0.2
Reusability	0.6	0.5	0.5

RADC Metric Application Analysis 14
• The methods of combining metric element scores to get a metric score, as discussed above, also apply to combining criterion scores to get a factor score. The above calculations are an example of combining by weighting second order averages.

- Criteria can contribute to more than one factor, as shown in the calculations above. (This is not the case for the RADC software metrics. Each software metric contributes to only one criterion.) When this occurs, a particular criterion can have an exaggerated contribution to the selected SQM. Criteria dampening is an additional method for combining criteria. This method corrects the problem. Criteria dampening can be applied to either the direct ratio or second order averaging scores, whether weighted or not.

  Criteria dampening scales back the contribution a particular criterion makes to the factor scores. Each score for that criterion is multiplied by the proportion that score represents of the total score for that criterion for all factors. When the criterion scores are direct ratios, criteria dampening is achieved by multiplying each criterion score numerator by the ratio of its denominator to the sum of the denominators. When the criterion scores are second order averages, criteria dampening is achieved by dividing each criterion score by the number of factors to which it contributes. In the above calculations, each Modularity and Self-Descriptiveness score would be divided by three and each Generality score would be divided by two. The weighting that was done in step 3 exhibits a certain amount of criteria dampening that was done intuitively, since the multiply-occurring Modularity and Self-Descriptiveness criteria were weighted so low. Nevertheless, the criteria dampening method was not applied.

The Maintainability quality factor was not initially achieved for any category. The metric element scores were examined to determine how each module failed. In general, the Modularity criterion scores were good, but the Self-Descriptiveness scores were poor. The critical code was brought to a Maintainability measure of one by simply entering "Not Applicable" for header fields that did not apply to a failing module. This entry was also added, where necessary, to the remaining code. This fix was sufficient to bring the nonessential code into compliance, since it needed to have only good maintainability. The essential modules required more work.

The essential code contained many modules which greatly exceeded 100 lines of source code, and a few modules that transferred control to other modules, rather than returning to the calling module. Seven modules were fixed by reducing the code or restructuring the returns. This was sufficient to bring the essential code into compliance.

RADC Metric Application Analysis 15
• Notice that this SQM not only quantified the quality of the software product, but also directly indicated how to improve the product. This indication makes an SQM particularly useful. • If the long modules had been broken into subroutines rather than shortened, it would have produced more modules that were called by only one other module, thus lowering the score for the Generality criterion.

The Expandability quality factor was not initially met for any category. The metric scores were examined to determine how each module failed. The weakness was with the Self-Descriptiveness and Generality criteria. The previous fix for the Self-Descriptiveness problem was applied but was insufficient to bring the factor score into compliance. Furthermore, the Generality criterion score was not able to be improved for any of the categories. The application was not general enough to allow many modules to be called by more than one other module. The developer chose to make no further improvement. The Expandability goal was not met.

RADC Metric Application Analysis 16
• From the description of the Generality metric used, the CE can determine that the expandability required for this code will not be significantly impacted by a low score for this metric. The expansions required by the various models of the aircraft would occur within single modules or by the addition of modules. The existing module interaction would remain intact.

The Reusability quality factor was not initially met for any category. The metric scores were examined to determine how each module failed. The Self-Descriptiveness and Generality criteria scores were particularly low.

Again, the Self-Descriptiveness fix was applied to bring its criterion score to one. This fix sufficed to bring the critical code into compliance and the nonessential code very close to compliance. Since the particular Generality metric used reflects the specificity of the calling environment, rather than the generality of the subroutines, the developer decided to deemphasize that criterion and emphasize the Functional Scope criterion. Furthermore, 100 percent compliance was achieved for the Functional Scope metric. By weighting Functional Scope by 0.5 and Generality by 0.1, the non-critical code was brought into compliance.

RADC Metric Application Analysis 17
• Rather than arbitrarily changing the weight applied to the Generality criterion, the developer could have added more generality metrics in order to exhibit reusability qualities that were present in the code, but not recognized. This probably would have been sufficient to overcome the low score on the GE.1(1) metric element. In general, a criterion should be based on several metrics. Otherwise, the criterion is being reduced to the presence of a single feature of code.

The final values for all the SQM are shown in table 3.6-10.

TABLE 3.6-10. FINAL SOFTWARE QUALITY METRICS SCORES, BY CODE CATEGORY

Quality Factor	Nonessential	Essential	Critical
Maintainability	0.9	0.9	1.0
Expandability	0.4	0.4	0.4
Reusability	0.8	0.9	0.9

3.6.2 Certification Conclusions

For certification purposes, it should be determined whether the metrics, as applied, were indeed reliable indicators of the properties measured. It should also be determined whether such measures equivalently fulfill the requirements of the certification guidelines. This determination can be based on the experience of the developer, the experimentation performed by the developer, or the experimentation that the developer references, as documented in the certification package. It can also be based on the experience of the CE or on references by the CE to the handbook data sheets.

Based on these inputs, the SQM performed are evaluated to see if they help the CE answer any of the certification determinations posed in section 3.3. This evaluation addresses each of the questions, in the order in which they were posed.

a. The use of these SQM indicates that the developer subjected the code to a disciplined, quantitatively controlled development process.

RADC Metric Application Analysis 18
• After comparing this software development project to those of the experiments on RADC's metrics, the CE would likely conclude that these SQM offer some assurance that the software was developed according to a disciplined approach, as recommended by RTCA/DO-178A, paragraph 1.1.

b. The developer set more demanding goals for each level of criticality. These goals regulated the amount of improvement required to make the modules meet the goals set for them.

RADC Metric Application Analysis 19
• After comparing this software development project to those of the experiments on RADC's metrics, the CE would likely conclude that these SQM offer some assurance that the software was developed by a process appropriate to the software level, as recommended by RTCA/DO-178A, paragraph 6.2.1. Regardless of the experiments, the SQM resulted in inconsistencies being detected and corrected.

c. RADC's metrics did not address whether any of the requirements stated for this software were fulfilled. Thus, the developer did not use SQM in the requirements analysis recommended by RTCA/DO-178A, paragraph 6.2.5.3.1.

d. This developer could not use SQM to substantiate test coverage, as defined in RTCA/DO-178A, paragraph 6.2.5.3.2, because RADC's software metrics do not measure anything that substantiates structural test coverage.

e. Maintainability and Expandability were partly performance requirements for this software. The developer submitted these two SQM as indicators of the extent to which those qualities were present in the delivered software.

RADC Metric Application Analysis 20
• After comparing this software development project to those of the experiments on RADC's metrics, the CE would likely conclude that these SQM offer some assurance that the software will operate at the required level of performance.

In this example, the developer used SQM to analyze the flight control code being developed. This analysis was used to assess and sometimes improve the software development product and process.

APPENDIX A - SOFTWARE METRIC DATA SHEETS

This appendix contains individual data sheets on each of the most prevalent software metrics encountered in this study. The data sheets are a summary of the concepts, constraints, criticisms, and other data needed to understand and critically evaluate the SQM that rely on these metrics.

Although the data sheets are not primarily intended as a guide for applying the metrics, some of the data sheets contain sufficient information that the metric could be properly and fully calculated by following the data sheet. On the other hand, some of the metrics are so extensive that the data sheets could not encompass all the necessary parts. In either case, the Technical Report, Software Quality Metrics (N. VanSuetendael and D. Elwell 1991), discusses each of them in detail. See the references listed on the data sheet for a complete coverage of a particular metric.

Each data sheet covers only one metric and they all contain the same fields of information. The fields are defined only as they apply to measuring source code. Many of the metrics can be applied more broadly, but the other possibilities are not addressed here.

The "SQM RELATIONSHIPS" field lists those quality factors that are known to be related to the particular software metric. This serves as a cross-reference to the applicable Software Quality Factor Data Sheet of appendix B.

The "TOOL" information is supplied on an as-available basis. It is not to be taken as a comprehensive list of tools available. Nor is the information to be taken as a recommendation.

Metrics that share a common heritage of theory are grouped together by family. The family name is given at the top of each sheet. The data sheets are arranged alphabetically by family name. Within each family they are arranged in the order in which they build upon each other, from the fundamental level to the highest level. When there is no hierarchical relationship within a family, the metric data sheets are arranged alphabetically by metric name.

Function Points

This metric calculates a weighted sum of the number of elementary functions supported by a module of source code. The functions are categorized into five types. The number of Function Points (FP) provided to the user by a module of source code is given by the following rule:

RULE: FP = FC x PCA

FC is the function count produced as follows:

Function Count = Number of External Inputs x 4, plus
Number of External Outputs x 5, plus
Number of Logical Internal Files x 10, plus
Number of External Interface Files x 7, plus
Number of External Inquiries x 4

where the weights given were determined experimentally. They are the values that made the measure most accurately reflect the perceived amount of function delivered in real projects of average complexity. PCA is the Processing Complexity Adjustment Factor, calculated as follows:

PCA = 0.65 + (0.01 x PC),

where PC is the Processing Complexity. It is based on an estimate of the degree of influence that each of 14 specified application characteristics (Albrecht and Gaffney 1983) has on the complexity of the functions counted. Each characteristic is assigned a "0", "1", or "2" degree of influence. Then the PC is the sum of these 14 numbers. This produces a PCA which adjusts the function count by ±35 percent for application and environment complexity.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function that relates to user inputs, user outputs, internal files, interface files, and/or user inquiries.

RANGE: This metric produces a real number greater than or equal to 1.3, the number of function points in the simplest algorithm which must contain an input and an output (assuming the lowest possible weight of one for each category and the lowest possible PCA of 0.65).

NECESSARY CONDITIONS: The criteria for determining the inputs, outputs, internal files, interface files, and inquiries from code must be defined for the context. The criteria for determining whether they are internal or external must be defined for the context. In order to use the weights given, the practitioner must also use Albrecht's definitions for the five types of function (Albrecht and Gaffney 1983).

QUALIFICATIONS: This metric has been validated primarily in a data processing system environment, rather than a real-time flight control environment. The given weights probably do not apply to code in avionic equipment.

The given weights are for programs that fall into a qualitatively chosen category of average complexity. Different sets of weights are given for simple or complex programs (Albrecht and Gaffney 1983).

The FP of an application is not necessarily equal to the sum of the FP of each of its sub-applications. Thus, FP is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously identify an input, output, internal file, interface file, or inquiry. In this case, an increase in the measure does not represent an increase in the amount of function. Furthermore, a different reader will likely judge the attributes differently. This effect has been observed in experimental analysis (Low and Jeffery 1990).

A problem with repeatability can also result from the arbitrariness of establishing application boundaries. The larger the application size, the more functions will be included. Furthermore, as the boundary is enlarged files will shift from being external interface files to being logical internal files. It is important that all measurements be based on an unambiguous specification for establishing application boundaries.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying the attributes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

The FP metric uses the source code to deduce the characteristics of the algorithm solved by the program. The rules used for this step are dependent on the programming language and technology used. But it is the algorithm that is measured, not the program. Thus, as Drummond (1985) also points out, the FP metric has the advantage of being language independent, as well as being independent of programmer style or experience. It is even independent of any changes in technology. Once the size of the algorithm is calculated, any program in any language that performs the same algorithm, whether poorly or well written, will provide the same number of function points.

SQM RELATIONSHIPS: Complexity, Simplicity

TOOLS: Checkpoint, Software Productivity Research, Inc., Burlington, MA
SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Behrens 1983; Drummond 1985; Jones 1988; Low and Jeffery 1990

H - Height of a Tree

This metric quantifies the depth of nesting in a set of modules of source code. It is based on the number of levels in a hierarchy tree of the control flow between modules.

RULE: The Height of a hierarchy tree is the maximum height attained by any node in the hierarchy tree, except that the Height of a single node tree is defined to be one.

The Height of a node is the number of levels of nesting below the root node. Thus, the root node has a height of zero and any child node of the root node has a height of one, being nested one level below the root node. Because the tree is drawn upside-down, height increases with lower levels of nesting.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to one, the Height of the tree for source code that has no nested modules.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously nest a node when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the depth of nesting. Furthermore, a different reader will likely judge nesting differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Performance, Simplicity, Understandability

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984[1], 1984[2], 1987, 1988, 1990

Twin Number

This metric quantifies the breadth of explosion of a module (or node) of source code. It is based on the number of nodes nested directly below it (child nodes) in a hierarchy tree of the control flow between nodes.

RULE: The Twin Number of a node is the number of child nodes of which it is the parent, except that the Twin Number of the node in a single node tree is defined to be one.

Every node that is immediately related to a particular node at the next lowest level (one level closer to the root) is considered a child of that node.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to zero, the Twin Number of a module with no child nodes (monad).

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously nest a node when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the number of child nodes. Furthermore, a different reader will likely judge nesting differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Simplicity

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984[1], 1984[2], 1987, 1988, 1990

M - Monadicity

This metric quantifies the number of irreducible modules (or nodes) in a set of modules of source code. It is based on the number of childless nodes in a hierarchy tree of the control flow between modules.

RULE: The Monadicity of a hierarchy tree is the number of nodes that have no children (a monad), except that a tree with a Height of one is constrained to have a Monadicity of one.

Every node that is immediately related to a particular node at the next lowest level (one level closer to the root) is considered a child of that node.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

This is a measure of the number of leaves on the tree, where the node at the end of each branch is a leaf. It is a measure of the "bushiness" of the tree.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to one, the Monadicity of tree with a Height of one.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is discontinuous between hierarchy trees of Height one and two. A tree with five child nodes at level one and one child node at level two has a monadicity of five. But if the single monad at level two is removed, the monadicity becomes one.

Even for hierarchy trees with a Height greater than one, this measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously divide a monad into two monads when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the number of monads. Furthermore, a different reader will likely judge nodes differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count

produced by a tool is right or wrong, it will certainly produce a repeatable result. Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Simplicity

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984[1], 1984[2], 1987, 1988, 1990

S - Software Size

This metric calculates the size of a set of modules (or nodes) of source code. It is based on the number of nodes in the hierarchy tree of the control flow between modules.

RULE: Count the number of nodes above the root node.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to zero, the size of a single-node tree.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not monotonic. When two different modules call the same subroutine, it appears as two distinct nodes on the hierarchy tree. This increases the size even though there is no increase in function.

Even if a program contains no such problem, this measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously divide a monad when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the number of monads. Furthermore, a different reader will likely judge nodes differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

Very little experimental validation has been performed for this metric and none of it has been published. No independent experiments have been performed.

SQM RELATIONSHIPS: Complexity, Performance, Simplicity, Understandability

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984[1], 1984[2], 1987, 1988, 1990

S_c - Structural Complexity

This metric calculates the structural complexity of a set of modules of source code. It is based on the number and relationship of nodes in a hierarchy tree of the control flow between nodes,

RULE: $S_c = H \times R_t \times M$

where H is the Height of the tree, R_t is the Twin Number of the root node, and M is the Monadicity of the tree. Each of these arguments is defined on a previous data sheet.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

The Structural Complexity reflects the nature of "the relation of conglomerates of thoughts expressing the functional composition of the system" (Ejiogu 1984). It combines into one measure both the number of related modules of thought and the closeness of their relationship. This measure was designed to be applied to issues relating to programming productivity, since it quantifies the amount of complexity in a program.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to one, the Structural Complexity of a single-node tree.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While this metric is based on structured programming concepts, unstructured programs do not render it useless. Any effect that lack of structure has on the value of the measure is to be taken as a reflection of the increased complexity that results when a program deviates from the ideal. Excessive values alert the developer to the presence of unstructured code so that it can be determined whether its presence is an asset or a detriment. For instance, when two nodes call the same subroutine the count is increased, even though there is no increase in function. But since calling a subroutine twice is not a detrimental effect, this apparent increase in complexity is allowed to remain.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See

the respective software metric data sheets for details. Very little experimental validation has been performed for this metric and none of it has been published. No independent experiments have been performed.

SQM RELATIONSHIPS: Complexity, Performance, Simplicity, Understandability

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984[1], 1984[2], 1987, 1988, 1990

η_1 - Number of Unique Operators

This metric quantifies the number of unique operators that occur in a module of source code.

RULE: Count the number of unique operators in the defined module.

An operator is any word of source code that represents an operation to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) The operation is performed on the objects that are the arguments of the operator. In computer program source code, most operators are the keywords that define a program statement. Some keywords have multiple parts that must occur as a set. They constitute one compound operator. The classification of each word of code must be based on an analysis of what that word of code does, from a functional point of view.

Uniqueness is to be determined according to the uniqueness of function rather than syntactical form. A unique function is rarely represented by more than one form. When this does occur, the various forms are not to be counted as unique operators. For example, if an exponent is denoted by either an "E" or a "^", these two forms represent one function. Conversely, two unique functions may be represented by the same form. When this occurs, the form is to be counted as two distinct operators, as indicated by the context. For example, if an asterisk denotes either multiplication or a comment, this one form represents two functions.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to two, the number of operators in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator, the criteria for establishing uniqueness, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The number of unique operators in a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, η_1 is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that is not an operator. In this case, an increase in the measure does not represent an increase in the number of operators. Furthermore, a different reader will likely define a different set of operators.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operators, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

η_2 - Number of Unique Operands

This metric quantifies the number of unique operands that occur in a module of source code.

RULE: Count the number of unique operands in the defined module.

An operand is any word of source code that represents an argument upon which an operation is to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) In computer program source code, most operands are the variables and constants of the program. The classification of each word of code must be based on an analysis of what that word of code does from a functional point of view.

Uniqueness is to be determined according to the uniqueness of function rather than syntactical form. A unique argument is rarely represented by more than one form. When this does occur, the various forms are not to be counted as unique operands. For example, if a variable name can be denoted by either VELOCITY or VEL, these two forms represent one argument. Conversely, two unique arguments may be represented by the same form. When this occurs, the form is to be counted as two distinct operands, as indicated by the context. For example, if the constant, 10, denotes 10 array elements in one part of a program, but it also denotes 10 volts in another part, this one operand represents two arguments.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to one, the number of operands in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operand, the criteria for establishing uniqueness, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The number of unique operands in a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, η_2 is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that

is not an operand. In this case, an increase in the measure does not represent an increase in the number of operands. Furthermore, a different reader will likely define a different set of operands. A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operands, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

N_1 - Total Number of Operator Occurrences

This metric quantifies the number of occurrences of operators in a module of source code.

RULE: Count the total number of occurrences of operators in the defined module.

An operator is any word of source code that represents an operation to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) The operation is performed on the objects that are the arguments of the operator. In computer program source code, most operators are the keywords that define a program statement. Some keywords have multiple parts that must occur as a set. They constitute one compound operator. The classification of each word of code must be based on an analysis of what that word of code does from a functional point of view.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to two, the number of operator occurrences in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that is not an operator. In this case, an increase in the measure does not represent an increase in the number of operators. Furthermore, a different reader will likely define a different set of operators.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operators, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

N_2 - Total Number of Operand Occurrences

This metric quantifies the number of occurrences of operands in a module of source code.

RULE: Count the total number of occurrences of operands in the defined module.

An operand is any word of source code that represents an argument upon which an operation is to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) In computer program source code, operands are the variables and constants of the program. The classification of each word of code must be based on an analysis of what that word of code does, from a functional point of view.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to one, the number of operand occurrences in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operand must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that is not an operand. In this case, an increase in the measure does not represent an increase in the number of operands. Furthermore, a different reader will likely define a different set of operands.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operands, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

η - Vocabulary

This metric calculates the total number of unique operators and operands that occur in a module of source code.

RULE: $\eta = \eta_1 + \eta_2$ words

where η_1 is the number of unique operators and η_2 is the number of unique operands. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

The Vocabulary represents the number of unique words in a module. Every word of executable code should be listed once in a vocabulary listing. However, when an operator and an operand are identical in form, they still constitute unique words since they serve unique functions. They are distinguished by their context.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to three, the Vocabulary of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing uniqueness, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Vocabulary of a module is not necessarily equal to the sum of the Vocabularies of each of its sub-modules. Thus, Vocabulary is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for details.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff

1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

N - Implementation Length

This metric calculates the total number of occurrences of operators and operands in a module of source code.

RULE: $N = N_1 + N_2$ words

where N_1 is the total number of operator occurrences and N_2 is the total number of operand occurrences. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

The Implementation Length represents the count of every word in a module. No word of executable code should be left uncounted.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to three, the Implementation Length of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for details.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987;

Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{N}$ - Estimated Length

This metric estimates the total number of occurrences of operators and operands in a module of source code.

RULE: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ words

where η_1 is the number of unique operators and η_2 is the number of unique operands. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to two, the Estimated Length of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of words should be rounded up to the next highest integer.

This estimate assumes that every combination of operators and operands of length η occurs only once in the module. This means that repeated segments of code, as long as η words, are put into subroutines rather than in redundant code. If such redundancy is present in the code, the Implementation Length will be underestimated by the Estimated Length Equation.

The Estimated Length Equation also assumes that operators and operands alternate without variation. Any code that does not follow this convention will likely produce an underestimation of the Implementation Length.

Because the Estimated Length Equation is nonlinear, adding the Estimated Lengths of sub-modules would not be expected to give an accurate Estimated Length for the module that they constitute. Nevertheless, an experiment by Ingojo indicated that the relationship holds fairly well (Halstead 1977).

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to problems with the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems with the Estimated Length, however, are due to the simplifying assumptions upon which the estimate is based. Whether or not these assumptions are reasonable, experiments show that $\hat{N}$ yields a close estimate of the Implementation Length, N (Halstead 1977).

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

V - Volume

This metric calculates the number of binary digits required to uniquely represent all of the operators and operands that occur in a module of length, N, and Vocabulary, η. It is not primarily a measure of the size or complexity of the function programmed, but a measure of the size of a particular implementation of a function. This metric should primarily be used to compare the size of the same program written in various languages or by various programmers.

RULE: $V = N\log_2\eta$ bits

where N is the Implementation Length and η is the Vocabulary. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Volume of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Volume of a module is not necessarily equal to the sum of the values of each of its sub-modules. Thus, Volume is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

This measure is related to the size of the function being implemented, but it is not a monotonic relationship. A larger algorithm could be implemented so tersely that its volume is less than a smaller algorithm. Conversely, a smaller algorithm could be implemented so verbosely that its volume is greater than a larger algorithm.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

V^* - Potential Volume

This metric calculates the number of binary digits required to implement the function of a module in its most efficient form, namely two operators (i.e., one that says "do the function" and another that groups the operands with the operator), and all the operands that it requires.

RULE: $V^* = (2 + \eta_2^*)\log_2(2 + \eta_2^*)$ bits

where η_2^* is the sum of the number of unique inputs and outputs for the module.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined. The operand ensures that there is at least one input or output.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Potential Volume of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Potential Volume of a module is not necessarily equal to the sum of the values of each of its sub-modules. Thus, Potential Volume is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for some part of the code that is not an input or output. In this case, an increase in the measure does not represent an increase in the size of the function. Furthermore, a different reader will likely define a different set of inputs and outputs.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying inputs and outputs, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a

repeatable result. Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Conciseness, Efficiency

TOOLS: No listing.

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

L - Program Level

This metric calculates how efficiently a module of source code is implemented. It is the ratio of the minimum number of bits it takes to represent an algorithm to the number of bits in a particular implementation. As the Vocabulary and/or Implementation Length decrease, the Program Level increases, i.e., the module is written at a higher level because each word carries more weight. Thus, Program Level indicates the amount of function per word rather than the number of words per function. In this sense, it measures what a program hides rather than what it shows.

RULE: $$L = \frac{V^*}{V} \frac{\text{bits of function}}{\text{bits of implementation}}$$

where V^* is the Potential Volume and V is the Volume. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined. The operand ensures that there is at least one input or output.

RANGE: This metric produces a positive real number less than or equal to one, the Program Level of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Program Level of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Program Level is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{L}$ - Estimated Program Level

This metric calculates how efficiently a module of source code is implemented. It is an estimate of the Program Level metric that can be used when the number of unique inputs and outputs, η_2^*, is not known. The Estimated Program Level increases as the module is written at a higher level. This occurs as the number of operators decreases to its minimum value of two and as the number of operand occurrences decreases to only one occurrence of each operand.

RULE: $$\hat{L} = \frac{2\ \eta_2}{\eta_1\ N_2} \quad \text{(unitless)}$$

where η_2 is the number of unique operands, η_1 is the number of unique operators, and N_2 is the total number of operand occurrences. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a positive real number less than or equal to one, the Estimated Program Level of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Estimated Program Level of a module is not necessarily equal to the sum of the estimated values for each of its sub-modules. Thus, the Estimated Program Level is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems are due to the simplifying assumptions upon which the estimate is based. Whether or not these assumptions are reasonable, experiments show that $\hat{L}$ yields a close estimate of the Program Level, L (Bulut 1974, Elshoff 1976, Halstead 1977, and Ottenstein 1981). They can be used interchangeably.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

I - Intelligence Content

This metric simply calculates the product of the Estimated Program Level (how efficiently the module is implemented) and the Volume (the size of the implementation) of a module of source code. It represents that constant quantity of information that is present in any implementation of a particular function, in any language, at any level. This quantity is basically an estimated Potential Volume of the program; I and V^* can often be used interchangeably.

RULE: $$I = \frac{2\ \eta_2}{\eta_1\ N_2} \times (N_1 + N_2)\log_2(\eta_1 + \eta_2) \quad \text{bits}$$

where η_2 is the number of unique operands, N_1 is the total number of operator occurrences, N_2 is the total number of operand occurrences, and η_1 is the number of unique operators. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Intelligence Content of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Intelligence Content of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Intelligence Content is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

The Intelligence Content measures the size of an algorithm based on its implementation. The Potential Volume measures the size of an algorithm based on its functional aspects: inputs and outputs. Therefore, when the Intelligence Content is used as an estimate for the Potential Volume, the accuracy of the estimate depends upon the quality of the programming. Redundant and extraneous uses of operators and operands alter the estimate. Halstead has characterized improper usages and has categorized them into six groups that he calls impurity classes (Halstead 1977).

Although it is generally undesirable that the constancy of the Intelligence Content be affected by the presence of program impurities, this effect results in an additional use of this metric. It provides a measure of the amount of impurity in an algorithm implementation. The closer the estimate is to the Potential Volume, the purer the algorithm implementation.

SQM RELATIONSHIPS: Conciseness, Efficiency

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

E - Programming Effort

This metric calculates the number of elementary mental discriminations done by a programmer to reduce a preconceived algorithm to a module of source code in a language in which the programmer is fluent. It is based on the assumption that, when writing a program, a programmer selects each word of the program by mentally searching a list of words from which to choose. Specifically, the programmer performs a mental binary search of the vocabulary of η words in order to select the N words used in the implementation. Furthermore, each comparison (or mental discrimination) in the selection process requires an effort related to the difficulty of understanding the program. This program difficulty is supplied by the reciprocal of the Program Level.

RULE: $E = V/L$ discriminations

where V is the Volume and L is the Program Level. Each of these arguments is defined on a previous data sheet. A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

This relationship indicates that the greater the Volume, or the lower the Program Level, the greater the effort required to write a program.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Programming Effort for the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Program Effort of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Program Effort is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric. Often, the Estimated

Program Level is used in place of the Program Level. In this case, any lack of monotonicity in the Programming Effort metric would be produced primarily by the assumptions underlying the Estimated Program Level.

SQM RELATIONSHIPS: Clarity, Complexity, Maintainability, Reliability, Simplicity

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{T}$ - Estimated Programming Time

This metric calculates the time it took a programmer to make the number of elementary mental discriminations performed when a module of source code was programmed. The number of elementary mental discriminations is given by the Programming Effort.

RULE: $$\hat{T} = \frac{E}{3600S} = \frac{\eta_1 N_2(\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2)\log_2 \eta}{7{,}200\eta_2 S} \text{ hours}$$

where E is the Programming Effort and the Stroud Number, S, is the total number of elementary mental discriminations that a programmer makes per second. This calculation assumes that the programmer's attention is entirely undivided and that the range of values found from psychological experimentation, $5<S<20$, applies to programming activity.

Each of the remaining arguments of this rule is defined on a previous data sheet. A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about one quarter of a second, the Estimated Programming Time for the most succinct implementation of an algorithm (with $S=20$).

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Estimated Programming Time of a module is not necessarily equal to the sum of the estimated values for each of its sub-modules. Thus, Estimated Programming Time is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems are due to the variability associated with the Stroud Number. It is not necessarily monotonic or repeatable for all programming contexts. It is not necessarily monotonic because a

programmer's knowledge in one discipline may result in a shorter programming time for a program with a greater value of Program Effort than for a program with a lesser value of Program Effort. This could occur when the former program is in a discipline with which the programmer is familiar, but the latter program is not. It is not necessarily repeatable because on some days a programmer will concentrate better than on others.

SQM RELATIONSHIPS: Performance

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, *SET Laboratories*, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

λ - Language Level

This metric calculates the efficiency with which algorithms can be implemented in a particular language. For any module of source code written in the particular language, the Language Level should be a constant, regardless of the Volume of the module or the Program Level at which it is written.

RULE: $\lambda = L^2V$

where L is the Program Level and V is the Volume. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a positive real number less than or equal to the Volume of the module.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Language Level of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Language Level is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Often, the Estimated Program Level is used in place of the Program Level. In this case, any lack of monotonicity in the Language Level would be produced primarily by the assumptions underlying the Estimated Program Level.

SQM RELATIONSHIPS: None.

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{B}$ - Number of Bugs

This metric calculates an estimate of the number of bugs that a module of source code contains at the time of measurement.

RULE: $\hat{B} = V/E_0$ bugs

where V is the Volume and E_0 is the number of discriminations per bug. The Volume is as defined on a previous data sheet. E_0 is determined by an evaluation of a programmer's previous work. Experimentation has found that 3,200 discriminations per bug is a typical value.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a positive real number.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bugs should be rounded up to the next highest integer.

The Number of Bugs in a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Number of Bugs is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems are due to the variability associated with E_0. It is not necessarily monotonic or repeatable for all programming contexts. It is not necessarily monotonic because a programmer's knowledge in one discipline may result in fewer bugs for a program with a greater Volume than for a program with a lesser Volume. This could occur when the former program is in a discipline with which the programmer is familiar, but the latter program is not. It is not

necessarily repeatable because on some days a programmer will concentrate better than on others.

SQM RELATIONSHIPS: Maintainability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

Information Flow

This metric calculates the complexity of a module of source code based on the size of the module and the flow of information within it.

RULE: The Information Flow Complexity (IFC) of a module is the sum of the IFCs for each procedure that composes it. The IFC for each procedure is calculated as follows:

$$IFC = Length \times (Fan\text{-}in \times Fan\text{-}out)^2$$

where Length is given by the number of lines of source code in the procedure, Fan-in is the number of local data flows that terminate at the procedure, and Fan-out is the number of local data flows that emanate from the procedure.

A module is defined as the set of procedures that either directly update or directly retrieve information from the particular data structure with which the module is associated.

Local data flow is data that is passed from procedure to procedure without going through the data structure.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function that has inputs and outputs.

RANGE: This metric produces an integer greater than or equal to one, the IFC of a one statement procedure that has one input and one output.

NECESSARY CONDITIONS: The criteria for determining the local inputs and outputs must be defined for the context. The criteria for delimiting procedures and modules must be defined for the context.

QUALIFICATIONS: The IFC of a module is not necessarily equal to the IFC of each of its submodules. Thus, IFC is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously count local inputs, local outputs, or lines of source code. In this case, an increase in the measure does not represent an increase in the amount of information flow. Furthermore, a different reader will likely judge the attributes differently.

A problem with repeatability can also result from the arbitrariness of establishing module boundaries. The larger the module, the greater will be the IFC. Furthermore, as the module boundaries shift, the fan-in and fan-out counts will change. It is important that all measurements be based on an unambiguous specification for establishing module boundaries.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying the inputs and outputs and for counting the lines of code, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

The IFC metric uses the source code to deduce the characteristics of the data flow of the algorithm solved by the program. The rules used for this step are dependent on the programming language and technology used. But it is primarily the data flow that is measured, not the program. Although the process for deducing the data flow is implementation dependent, the resulting value of the IFC metric has the advantage of being highly language independent, as well as highly independent of programmer style or experience. It is even highly independent of any changes in technology. Once the data flow complexity of an algorithm is calculated, most any program in any language that performs the same algorithm, whether poorly or well written, will have the same IFC value.

This complexity measure has been shown to represent the decrease in complexity that sometimes can be made by adding lines of code (Henry and Kafura 1984). It appears to measure a dimension of complexity somewhat different from that measured by Halstead's and McCabe's metrics (Kafura and Reddy 1987).

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: No listing.

REFERENCES: Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987

v(G) - Cyclomatic Complexity

This metric calculates the number of linearly independent program flow paths in the basis set of paths for a module of source code. Every possible path of program flow through a module can be represented as a linear combination of some subset of the basis set of paths.

RULE: $v(G) = e - n + 2p$

where e is the number of edges and n is the number of nodes in the program flow control graphs of all the modules. The variable p is the number of modules.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

To draw the control graph, first, block the code into predicate nodes, collecting nodes, and function nodes. A predicate node is a block of code ending with a statement that branches to multiple locations. A collecting node begins with a labeled statement to which flow branches. When a predicate node is labeled, it is both a predicate node and a collecting node. Each strictly sequential block of statements between any two predicate or collecting nodes compose a function node.

Once the code is blocked, graphically represent each block with a circle and the flow of control between them with lines. In this graph, the circles are nodes and the lines are edges. The function nodes can be omitted from the graph without affecting the value of the Cyclomatic Complexity.

DOMAIN: This metric can be applied to any source code that can be blocked into nodes.

RANGE: This metric produces an integer greater than or equal to one, the Cyclomatic Complexity of code with purely sequential flow.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes each type of node must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a node when drawing the graph. In this case, a net increase in the edge-node difference is not

represented by an increase in the measure. Furthermore, a different reader will likely judge the nodes differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Modularity, Performance, Reliability, Simplicity, Testability, Understandability

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Cote et al 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

v(G) - Cyclomatic Complexity
Alternate Rule 1

This metric quantifies the number of linearly independent program flow paths in the basis set of paths for a module of source code. Every possible path of program flow through a module can be represented as a linear combination of some subset of the basis set of paths.

ALTERNATIVE RULE 1: The Cyclomatic Complexity of a module is equal to the number of regions delineated by its control graph, with an additional edge from the last node to the first. The control graph must be drawn with no crossing edges: it must be a planar graph.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

To draw the control graph, first, block the code into predicate nodes, collecting nodes, and function nodes. A predicate node is a block of code ending with a statement that branches to multiple locations. A collecting node begins with a labeled statement to which flow branches. When a predicate node is labeled, it is both a predicate node and a collecting node. Each strictly sequential block of statements between any two predicate or collecting nodes compose a function node.

Once the code is blocked, graphically represent each block with a circle and the flow of control between them with lines. In this graph, the circles are nodes and the lines are edges. Also draw an edge from the exit node to the entrance node. The function nodes can be omitted from the graph without affecting the value of the Cyclomatic Complexity. The region surrounding the graph must also be counted as a region.

DOMAIN: This metric can be applied to any source code that can be blocked into nodes and can be drawn in a planar graph.

RANGE: This metric produces an integer greater than or equal to one, the Cyclomatic Complexity of code with purely sequential flow.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes each type of node must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: The control graphs of all structured and many unstructured programs can be drawn as planar graphs. Sometimes the control graph of unstructured code cannot be drawn as a planar graph.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a node when drawing the graph. In this case, a net increase in the edge-node difference is not represented by an increase in the measure. Furthermore, a different reader will likely judge the nodes differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Modularity, Performance, Reliability, Simplicity, Testability, Understandability

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Cote et al 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

v(G) - Cyclomatic Complexity
Alternate Rule 2

This metric quantifies the number of linearly independent program flow paths in the basis set of paths for a module of source code. Every possible path of program flow through a module can be represented as a linear combination of some subset of the basis set of paths.

ALTERNATIVE RULE 2: $v(G) = \pi + 1$

where π is the number of control flow branch conditions in the module.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

This formula for structured programs requires that for every predicate node there is exactly one collecting node, and that the program has unique entry and exit nodes. Exceptions to the rules always alter v(G). As long as there are relatively few exceptions to these rules, the Cyclomatic Complexity will be not be substantially affected. If there are a significant number of exceptions, the impact of multiple exits can be corrected by computing the following:

$$v(G) = \pi - s + 2$$

where s is the number of exits from the module (Harrison 1984). Of course, counting the number of exits requires additional analysis. If the code is primarily structured, the inaccuracy of the original equation is probably not significant enough to warrant the extra effort.

DOMAIN: This metric can be applied to any executable source code.

RANGE: This metric produces an integer greater than or equal to one, the Cyclomatic Complexity of code with purely sequential flow.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes a branch condition must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: This measure counts all conditions present, whether necessary or not. For example, code may contain a condition in a context where it will always be true. The Cyclomatic Complexity counts the number of conditions present in the algorithm, as implemented.

The Cyclomatic Complexity of a collection of independent modules, as in a main program and its subroutines, is simply the sum of the Cyclomatic Complexities for each module. For this rule, it is improper to calculate the total Cyclomatic Complexity by analyzing all the modules as if they were one continuous sequence of code. In this case, the value is deflated by the number of modules in the collection minus one.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a condition when reading the code. In this case, a net increase in the number of branch conditions is not represented by an increase in the measure. Furthermore, a different reader will likely judge the conditions differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying branch conditions, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Modularity, Performance, Reliability, Simplicity, Testability, Understandability

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Cote et al 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

ev(G) - Essential Complexity

This metric calculates the amount of unstructured code in a module.

RULE: ev(G) = v(G) - m

where m is the number of proper subgraphs with unique entry and exit nodes.

A proper subgraph is any part of the graph that consists of only a "SEQUENCE" construct, an "IF-THEN-ELSE" construct, a "DO-UNTIL" construct, a "DO-WHILE" construct, or a "CASE" construct.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

The smaller the value of ev(G), the greater the proportion of structured code. The Essential Complexity of a structured module is one. Greater values for the Essential Complexity of a module indicate the size of the nonstructured portion of the code; that code that does not conform to the proper subgraph structure (McCabe 1976).

Like the Cyclomatic Complexity, this measure also reflects the state of the algorithm, as implemented. It does not indicate whether the conditions, as implemented, are necessary for the algorithm.

To draw the control graph, first block the code into predicate nodes, processing nodes, and collecting nodes. A predicate node is a block of code ending with a statement that branches to multiple locations. A collecting node begins with a labeled statement to which flow branches. When a predicate node is labeled, it is both a predicate node and a collecting node. Each strictly sequential block of statements between any two predicate or collecting nodes compose a function node.

Once the code is blocked, graphically represent each block with a circle and the flow of control between them with lines. In this graph, the circles are nodes and the lines are edges. The function nodes can be omitted from the graph without affecting the value of the Cyclomatic Complexity.

DOMAIN: This metric can be applied to any source code that can be blocked into nodes.

RANGE: This metric produces an integer greater than or equal to one, the Essential Complexity of a totally structured module.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes each type of node must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: This measure only applies to executable source code.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a node when drawing the graph. In this case, a net increase in the edge-node difference is not represented by an increase in the measure. Furthermore, a different reader will likely judge the nodes differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Conciseness, Efficiency, Modularity, Performance, Reliability, Simplicity, Understandability

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Cote et al 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

FS.1(2) - Function Specificity

This metric calculates, for a single Computer Software Configuration Item (CSCI), the ratio of modules in which the functions of the module are described in the comments to total applicable modules.

RULE: For each module of the CSCI, answer the question:

"Is a description of the functions provided in the comments?" (Bowen, Wigle, and Tsai 1985)

Divide the total number of affirmative answers by the number of modules to which the question applied.

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any module of source code that performs a function.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that constitute a description of the function must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: This single question requires both that the comments exist and that they contain a description of the functions.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously judge that a description in a comment satisfies the requirement. In this case, an increase in the measure does not represent an increase in how specifically the function is described. Furthermore, a different reader will likely judge the sufficiency of the description differently for the same code. At best, it only establishes that the function was described, not how well it was described.

A software tool can also produce non-monotonic results. Although it has well-defined rules for evaluating the descriptions, the rules will likely misinterpret some uncommon implementation. Whether the count produced by a tool is right or wrong, the tool will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Reusability

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1990; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985[1], 1985[2], 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

GE.1(1) - Unit Referencing

This metric calculates, for a single Computer Software Configuration Item (CSCI), the ratio of modules that are sufficiently general that they are called by more than one other module to total number of modules.

RULE: For each CSCI, answer the question:

"How many units are called by more than one other unit?" (Bowen, Wigle, and Tsai 1985)

Divide the answer by the number of modules in the CSCI.

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any module of source code.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that constitute a call must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily monotonic. A module may be sufficiently general to be called by many other modules in various applications, but only be required by one other module in a particular application. Thus, it is not a measure of the generality of the module called.

This metric is highly repeatable, even when determined by a program reader, since a module is either called or not called.

SQM RELATIONSHIPS: Expandability, Flexibility, Reusability

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1990; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

M0.1(3) - Modular Implementation

This metric calculates, for a single Computer Software Configuration Item (CSCI), the ratio of modules that are sufficiently modularized (so that they are not too large) to total applicable modules.

RULE: For each module of the CSCI, answer the question:

"Are the estimated lines of source code for this unit 100 lines or less, excluding comments?" (Bowen, Wigle, and Tsai 1985)

Divide the total number of affirmative answers by the number of modules in the set to which the question applied.

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any module of source code that contains executable statements.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that constitute a comment, the combinations of characters that constitute lines of source code, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The count of lines of source code is called an estimate because it is recognized that there is no one standard for defining code that constitutes source code.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously judge that a line of code is not source code. In this case, an increase in the measure does not represent an increase in the modularity of the implementation. Furthermore, a different reader will likely judge lines of code differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying the comments and lines of source code, the rules will likely misinterpret some uncommon implementation. Whether the count produced by a tool is right or wrong, the tool will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Expandability, Flexibility, Interoperability, Maintainability, Portability, Reusability, Survivability, Verifiability

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT. Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1990; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

MO.1(7) - Modular Implementation

This metric calculates, for a single Computer Software Configuration Item (CSCI), the ratio of modules that always transfer control back to the calling unit to total applicable modules.

RULE: For each module of the CSCI, answer the question:

"Is control always returned to the calling unit when execution is completed?" (Bowen, Wigle, and Tsai 1985)

Divide the total number of affirmative answers by the number of modules in the set to which the question applied.

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any module of source code that is called by another module.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that determine when execution of a module is completed, what constitutes a calling module, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily monotonic. A module may transfer control back to the calling module, but not to the next statement in the sequence. This would break up the modularity of the calling module. In this case, an increase in the measure does not represent an increase in the modularity of the implementation.

This metric is highly repeatable, even when determined by a program reader, since control is either returned to the calling module or is not returned to the calling module.

SQM RELATIONSHIPS: Expandability, Flexibility, Interoperability, Maintainability, Portability, Reusability, Survivability, Verifiability

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1990; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

SD.2(1) - Effectiveness of Comments

This metric calculates, for a single Computer Software Configuration Item (CSCI), the ratio of modules in which header comments are complete to total applicable modules.

RULE: For each module of the CSCI, answer the question:

"Are there prologue comments which contain all information in accordance with the established standard?" (Bowen, Wigle, and Tsai 1985)

Divide the total number of affirmative answers by the number of modules in the set to which the question applied.

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any source code for which a standard has been set for header comments.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that identify a prologue comment, the combinations of characters that must constitute each required piece of information, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: This question requires that the comment exist and that it contain the specified information. The only case in which the question does not apply is for a module for which no prologue comments are required.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously judge that a piece of information in a comment satisfies the requirement. In this case, an increase in the measure does not represent an increase in the comment effectiveness. Furthermore, a different reader will likely judge the sufficiency of the comments differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for evaluating the information, the rules will likely misinterpret some uncommon implementation. Whether the count produced by a tool is right or wrong, the tool will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Expandability, Flexibility, Maintainability, Portability, Reusability, Verifiability

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1990; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

APPENDIX B - SOFTWARE QUALITY FACTOR DATA SHEETS

This appendix contains individual data sheets on each of the most prevalent software quality factors encountered in this study. The data sheets are a summary of the concepts, constraints, criticisms, and other data needed to understand and critically evaluate the application of Software Quality Metrics (SQM) that measure these quality factors.

Although the data sheets are not primarily intended as a guide for applying SQM, some of the combinations of these data sheets with the Software Metric Data Sheets that they reference contain sufficient information that an SQM could be properly and fully calculated from them. On the other hand, some of the SQM are so extensive that the data sheets could not encompass all the necessary parts. In either case, the Technical Report, Software Quality Metrics (N. VanSuetendael and D. Elwell 1991), discusses the most prominent SQM in detail. See the references listed on these data sheets for complete coverage of a particular quality factor. See the references listed on the Software Metric Data Sheets for complete coverage of a particular software metric that an SQM uses to measure a quality factor.

Each data sheet covers only one factor and they all contain the same fields of information. The fields are defined only as they apply to measuring source code. Many of the metrics can be applied more broadly, but the other possibilities are not addressed here. The data sheets are arranged alphabetically by factor name.

The "CLAIM" field is filled by the particular quality factor description or definition that is the most comprehensive, without being too general, and the most independent. The goal of identifying quality factors is to come up with a set of factors that are independent of one another and that comprehensively cover the full range of quality factors of concern for software.

The "SQM RELATIONSHIPS" field lists those software metrics that are known to be related to the particular quality factor. This serves as a cross-reference to the applicable Software Metric Data Sheet of appendix A. However, not all of the referenced software metrics are addressed in appendix A. See the articles listed in the bibliography for further information.

The "TOOLS" information is supplied on an as-available basis. It is not to be taken as a comprehensive list of tools available. Nor is the information to be taken as a recommendation.

SOFTWARE QUALITY FACTOR DATA SHEET

Accuracy

This quality factor addresses the concern that programs provide the precision required for each output. Accuracy is important because most computer manipulations are not exact, but are limited approximations.

CLAIM: Some researchers claim that

"A software product possesses accuracy to the extent that its outputs are sufficiently precise to satisfy their intended use." (Boehm et al 1978).

DOMAIN: This factor only applies to source code for which the required precision has been explicitly stated for each output.

NECESSARY CONDITIONS: The required precision must be defined for the context. The outputs must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This factor does not address whether a calculation or output should or should not be present. The former is the concern of completeness, the latter of conciseness. Nor does it address whether the proper quantity is being calculated or output. That is the concern of correctness.

SQM RELATIONSHIPS: RADC's AC.1(7)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Clarity

This quality factor addresses the concern that programs be easily understood by people. Programmers are concerned particularly that source listings be easily understood. Users are concerned that the human interface of the program is easily understood.

CLAIM: One researcher claims that software clarity is a

"Measure of how clear a program is, i.e., how easy it is to read, understand, and use" (Kosarajo and Ledgard, as quoted in McCall, Richards, and Walters 1977).

Although users are concerned about the clarity of the program dialogue, it is best to let the users' concern for clarity be handled by the quality factor, usability. Then clarity only addresses the ease of reading and understanding the program.

DOMAIN: This factor applies to any source code. Programming is such an iterative process that the code should be programmed with clarity at least to aid the programmer who writes some code, reviews it, and then writes some more.

NECESSARY CONDITIONS: The audience to which the code must be clear must be defined for the context. The criteria for determining that the program is clear must be determined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: Program clarity can suffer either because the necessary information is not provided, or because the information provided is not well presented. Either problem impacts program clarity, since this quality factor is concerned about the net impact on the ease of understanding the program, regardless of the cause. In this case, clarity can be equated with self-descriptiveness.

SQM RELATIONSHIPS: Halstead's Effort, RADC's SD.2(1), SD.2(3)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Completeness

This quality factor addresses the concern that program functions be implemented completely.

CLAIM: Some researchers claim that software completeness is determined by

"Those characteristics of software which provide full implementation of the functions required" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor only applies to source code for which the required functions have been explicitly stated.

NECESSARY CONDITIONS: The required functions and their extent must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This definition of completeness distinguishes functional completeness from the completeness of the source code. The latter encompasses additional attributes, such as program comments, that are desired by programmers rather than the user. Furthermore, these attributes are already addressed by the understandability and clarity quality factors.

SQM RELATIONSHIPS: RADC's CP.1(2)

TOOLS: ***, IETRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Complexity

This quality factor addresses the concern that programs not be complex. There are many types of program complexity, but the complexity of concern is the complexity that is constructed in the mind of a programmer, the psychological complexity.

CLAIM: The authors claim that

The complexity of a program is the extent to which it is involved or intricate, composed of many interwoven parts.

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for determining complexity must be defined for the context.

QUALIFICATIONS: Although the psychological complexity is the important one, a given SQM may claim that psychological complexity is totally encompassed by the complexity of a particular structure of a program. In this case, the two complexities can be used interchangeably. It is important that the type of complexity measure be qualified. Different types must not be directly compared.

This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor addresses the complexity of program code, not the complexity of the user interface. Programmers are concerned that programs not be complex so that they can easily and assuredly satisfy programmer and user needs. Any complexity that may exist for the user is addressed by the quality factor, usability.

SQM RELATIONSHIPS: Albrecht's Function Points; Ejiogu's Structural Complexity; Halstead's Length, Volume, and Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity; RADC's SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
Checkpoint, Software Productivity Research, Inc., Burlington, MA
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA

(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Behrens 1983; Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Cote et al 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Drummond 1985; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978, 1988; Kafura and Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Low and Jeffrey 1990; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Conciseness

This quality factor addresses the concern that programs not contain any extraneous information.

CLAIM: Some researchers claim that software conciseness is determined by

"The ability to satisfy functional requirements with minimum amount of software" (McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code for which the functional requirements have been explicitly stated.

NECESSARY CONDITIONS: The functional requirements must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor basically addresses how efficiently the source code was used. Any efficiency quality factors may already encompass this quality factor. Another word used for efficiency is *effectiveness*.

SQM RELATIONSHIPS: Halstead's Potential Volume, McCabe's Essential Complexity,

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA

REFERENCES: Halstead 1977; McCabe 1976, 1982

SOFTWARE QUALITY FACTOR DATA SHEET

Consistency

This quality factor addresses the concern that the source code syntax and constructs in programs be implemented uniformly.

CLAIM: Some researchers claim that software consistency is determined by

> "Those characteristics of software which provide for uniform design and implementation techniques and notation" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The standard notation must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor addresses self-consistency. External consistency is more properly addressed by the quality factors, completeness, correctness, and performance, since they all compare the implementation with an external standard.

SQM RELATIONSHIPS: RADC's software metrics

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Correctness

This quality factor addresses the concern that software design and documentation formats conform to the specifications and standards set for them. It is not concerned with any content affecting software operation or performance.

CLAIM: Some researchers claim that software correctness is determined by the

"Extent to which the software conforms to its specifications and standards" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor only applies to source code for which specifications and standards have been explicitly stated.

NECESSARY CONDITIONS: The specifications and standards must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with completeness. Although each specification may have been completely addressed, it may have been addressed incorrectly. Completeness of a function must be satisfied before its correctness can be established.

SQM RELATIONSHIPS: RADC's CP.1(2)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Efficiency

This quality factor addresses the concern that programs optimally use any computer resources.

CLAIM: Some researchers claim that

> "A software product possesses the characteristic Efficiency to the extent that it fulfills its purpose without waste of resources." (Boehm et al 1978)
>
> In practice, one is not usually concerned with general efficiency, but with the efficiency of utilization of a particular resource, for example, efficient use of processing time.

DOMAIN: This factor only applies to source code for which the purpose has been explicitly stated.

NECESSARY CONDITIONS: The criteria for establishing that a resource is wasted must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with how the software rates for a particular performance measure. Performance is concerned with how close the measure comes to a standard. Efficiency requires that the standard be met, and then addresses how effectively it was met.

SQM RELATIONSHIPS: Halstead's Potential Volume, McCabe's Essential Complexity, RADC's EP.2(4)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Halstead 1977; Lasky, Kaminsky, and Boaz 1989; McCabe 1976, 1982; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Expandability

This quality factor addresses the concern that program limitations be easy to extend.

CLAIM: Some researchers claim that software expandability is determined by the

> "Relative effort [required] to increase the software capability or performance by enhancing current functions or by adding new functions or data" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for distinguishing a change from an expansion must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with flexibility, maintainability, or reusability. Expandability assumes that all existing functions are still performed, but possibly added to or enhanced. Expandability also assumes that there is no change in the context in which the program is used. Another word used to describe this quality factor is augmentability.

SQM RELATIONSHIPS: RADC's GE.1(1), GE.2(2), MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Flexibility

This quality factor addresses the concern that programs be easy to change to meet different requirements, with no change in the context.

CLAIM: Some researchers claim that software flexibility is determined by the

"Ease of effort for changing the software missions, functions, or data to satisfy other requirements" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for distinguishing different requirements from a change in context or an expansion must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with expandability, maintainability, or reusability. Expandability addresses the simple case of satisfying extended requirements, rather than different requirements. In order to keep flexibility independent of expandability, flexibility should assess the ability to accommodate different requirements. Maintainability assumes no change in the functions of a program. Reusability requires a change of context.

SQM RELATIONSHIPS: RADC's GE.1(1), GE.2(2), MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Integrity

This quality factor addresses the concern that programs must continue to perform their function even under adverse conditions: inputs that are unexpected, improper, or harmful.

CLAIM: Some researchers claim that software integrity is determined by the

> "Ability of software to prevent purposeful or accidental damage to the data or software" (United States Army Computer Systems Command as quoted in McCall, Richards, and Walters 1977).
>
> This definition assumes that damage to the data or software will result in the loss of function.

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The adverse conditions must be defined for the context.

QUALIFICATIONS: This definition assumes that program integrity is an issue only when the program is operating. When it is not operating, the operating system must guard the program and its data from damage; the program cannot.

CRITICAL ANALYSIS: This quality factor must not be confused with security or reliability. Security contributes directly to integrity, but security is only concerned with unauthorized access to information. When a program is not secure, the confidentiality of the data is compromised, whether or not the program is subjected to adverse conditions. Similarly, reliability is determined by program failures that occur during execution under the specified conditions, not during adverse conditions.

SQM RELATIONSHIPS: RADC's software metrics

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Maintainability

This quality factor addresses the concern that programs be easy to fix, once a failure is identified.

CLAIM: Some researchers claim that software maintainability is determined by the

"Ease of effort for locating and fixing a software failure within a specified time period" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor only applies to source code for which operational failure and restoration is defined.

NECESSARY CONDITIONS: The criteria for program failures must be defined for the context.

QUALIFICATIONS: In general, maintainability encompasses the ease of identifying and making any changes to programs. But when used as one of a set of quality factors, maintainability is usually restricted to changes that are made to fix software failures as in the definition above. Often, maintainability is further restricted to the ease of making the fix, once the cause for the failure is found. This restriction keeps it highly independent. In particular, it makes maintainability independent of complexity and understandability.

CRITICAL ANALYSIS: This quality factor must not be confused with expandability, flexibility, modifiability, portability, or reusability. None of these quality factors apply to changes made in response to software failures. They address the concerns related to making changes for other reasons.

SQM RELATIONSHIPS: Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity; RADC's MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Cote et al 1988;

Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

Modifiability

This quality factor addresses the concern that programs be easy to change, regardless of the reason for the change.

CLAIM: Some researchers claim that

> "A software product possesses modifiability to the extent that it facilitates the incorporation of changes, once the nature of the desired change has been determined." (Boehm et al 1978)

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for a program change must be defined for the context.

QUALIFICATIONS: The effort expended to understand how to change a program is not addressed by this quality factor. Only the next step of implementing the change is addressed.

CRITICAL ANALYSIS: This quality factor encompasses all the concerns addressed by expandability, flexibility, maintainability, portability, and reusability. This does not mean that they are identical, but rather that modifiability is fundamental to them all. The ease of making all types of changes must be addressed for this quality factor. The other quality factors emphasize a particular type of modifiability.

SQM RELATIONSHIPS: Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Cote et al 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Henry 1982; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton

1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Modularity

This quality factor addresses the concern that programs be composed of many small, simple, independent steps that are clearly delineated by the code.

CLAIM: One researcher claims that software modularity is the

> "Formal way of dividing a program into a number of sub-units each having a well defined function and relationship to the rest of the program" (Mealy as quoted in McCall, Richards, and Walters 1977).
>
> These sub-units are called modules. Thus, the quality factor is modularity. The program itself may be a sub-unit of some larger program.

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for distinguishing modules and intermodule relationships must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor is usually used to indicate the presence of one of the following quality factors: expandability, flexibility, maintainability, modifiability, portability, reusability, and understandability. Depending on which of these quality factors is to be indicated by modularity, different modules and intermodule relationships are identified. For example, a program that is broken into one set of modules to enhance expandability may be broken into a different set of modules to enhance understandability.

SQM RELATIONSHIPS: McCabe's Cyclomatic Complexity and Essential Complexity; RADC's MO.1(3), MO.1(5), MO.1(7)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Boehm et al 1978; Bowen, Wigle, and Tsai 1985;

Carver 1986; Cote et al 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988

SOFTWARE QUALITY FACTOR DATA SHEET

Performance

This quality factor addresses the concern of how well a program attribute or function is implemented with respect to some standard. Often, this is related to the utilization of resources.

CLAIM: One researcher claims that software performance is determined by

> "The effectiveness with which resources of the host system are utilized toward meeting the objective of the software system" (Dennis as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code for which performance standards have been set.

NECESSARY CONDITIONS: The standard against which the implementation is to be compared must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with completeness or correctness. Performance is concerned with how well the job is done, given that a program completely and correctly meets its specifications. The performance quality factor is used to determine which satisfactory program performs better.

SQM RELATIONSHIPS: Albrecht's Function Points; Ejiogu's Structural Complexity; Halstead's Length, Volume, and Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
Checkpoint, Software Productivity Research, Inc., Burlington, MA
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Behrens 1983; Bulut 1974; Carver 1986; Cote et al 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Drummond 1985; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978, 1988; Kafura and Henry 1982; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Low and

Jeffrey 1990; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Portability

This quality factor addresses the concern that programs be changed easily to operate on a different set of equipment.

CLAIM: One researcher claims that software portability is determined by

> "How quickly and cheaply the software system can be converted to perform the same functions using different equipment" (Kosy as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code that performs functions that are supported on other equipment.

NECESSARY CONDITIONS: The criteria for distinguishing lack of modifiability from lack of portability must be defined for the context.

QUALIFICATIONS: The portability of the code, not the function that it performs, is addressed by this quality factor. If the hardware on which it operates is unique, there is no machine to which the code can be transported to perform the same function. This is not the fault of the code.

Every change required to operate a program on different equipment is weighted by the modifiability of the code. If the modifiability is poor, a very machine-independent program may still appear to be not very portable.

CRITICAL ANALYSIS: This quality factor must not be confused with reusability. Portability addresses only those changes required to make the program work on a different machine; everything else remains the same.

SQM RELATIONSHIPS: RADC's MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Suñazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Reliability

This quality factor addresses the concern that programs continue to perform properly over time.

CLAIM: One researcher claims that software reliability is determined by

> "The probability that a software system will operate without failure for at least a given period of time when used under stated conditions" (Kosy as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code for which the proper operating conditions are specified.

NECESSARY CONDITIONS: The proper operating conditions and the criteria for establishing a failure must be defined for the context.

QUALIFICATIONS: For code-based SQM, this quality factor must be given by probabilities based solely on program code. Most reliability estimates are based on experimental data.

CRITICAL ANALYSIS: This quality factor must not be confused with accuracy or correctness. Reliability addresses the continuance of functions already presumed to be both accurate and correct.

SQM RELATIONSHIPS: Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity, RADC's SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Cote et al 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989;

Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Reusability

This quality factor addresses the concern that programs be easy to reuse in a different application.

CLAIM: Some researchers claim that software reusability is determined by the

"Relative effort to convert a software component for use in a different application" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The proper operating conditions and the criteria for establishing a failure must be defined for the context.

QUALIFICATIONS: Every change required to use a program in a different application is weighted by the modifiability of the code. If the modifiability is poor, a very application-independent program may still appear to be not very reusable.

This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with expandability, flexibility, or portability. All three address ease of changing a program, but for reasons other than using the program to perform the same function in a different application.

SQM RELATIONSHIPS: RADC's AP.3(2), FS.1(2), GE.1(1), GE.2(2), MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1), ST.5(2)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

Simplicity

This quality factor addresses the concern that, as much as possible, programs be implemented in strictly sequential steps that depend only on the step before it. Comments should exhibit a similar straight forwardness.

CLAIM: Some researchers claim that software simplicity is determined by

> "Those characteristics of software which provide for definition and implementation of functions in the most noncomplex and understandable manner" (Bowen, Wigle, and Tsai 1985).
>
> Simplicity is the opposite of the quality factor, complexity.

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for determining simplicity must be defined for the context.

QUALIFICATIONS: Simplicity of implementation must be evaluated independently of the simplicity or complexity of the algorithm. Simplicity addresses how simply an algorithm of a certain complexity is implemented.

> This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor is usually represented by other quality factors. Since simplicity is desired because it enhances understanding, clarity and understandability are more commonly used.

SQM RELATIONSHIPS: Albrecht's Function Points; Ejiogu's Structural Complexity; Halstead's Length, Volume, and Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity; RADC's SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
Checkpoint, Software Productivity Research, Inc., Burlington, MA
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Behrens 1983; Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Cote et al 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Drummond 1985; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978, 1988; Kafura and Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Low and Jeffrey 1990; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Testability

This quality factor addresses the concern that programs be easy to test.

CLAIM: Some researchers claim that

> "A software product possesses the characteristic Testability to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance." (Boehm et al 1978).

DOMAIN: This factor only applies to source code for which acceptance criteria can been established.

NECESSARY CONDITIONS: The test plan must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with maintainability. Testability is only concerned with finding problems, not fixing them.

SQM RELATIONSHIPS: McCabe's Cyclomatic Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD.
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby and Phillips 1983; Carver 1986; Cote et al 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

SOFTWARE QUALITY FACTOR DATA SHEET

Understandability

This quality factor addresses the concern that programs be easy to understand.

CLAIM: One researcher claims that software understandability is determined by the

"Ease with which the implementation can be understood" (Richards as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for establishing whether a program is understood by a program reader must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with modifiability. Before any change can be made to a program, the code must be understood as implemented. The next step may be to modify it.

SQM RELATIONSHIPS: Ejiogu's Structural Complexity; Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Cote et al 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Henry 1982; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Usability

This quality factor addresses the concern that programs be easy to use.

CLAIM: Some researchers claim that

> "A software product possesses the characteristic Usability to the extent that it is convenient and practicable to use." (Boehm et al 1978).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for establishing that a program is easy to use must be defined for the context.

QUALIFICATIONS: To keep this quality factor independent of maintainability, portability, and reusability, this quality factor must address the usability of an error-free program that is run on the intended machine, in the intended context or application.

CRITICAL ANALYSIS: This quality factor must not be confused with programmer concerns, like understandability. Usability is a strictly user-oriented quality factor.

SQM RELATIONSHIPS: RADC's software metrics

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

APPENDIX C - SOFTWARE METRIC DATA

This appendix contains the software metric data from which the software metrics and Software Quality Metrics (SQM) of the worked examples were computed. Halstead's and McCabe's software metrics are as computed by PC-Metric, Release 2.0, of the Assembler version. Rome Air Development Center's (RADC) software metrics and SQM were computed by the authors' own text analyzer. Values that tripped the thresholds for particular SQM in the worked examples are marked with an asterisk (*).

The code analyzed was supplied by a major avionics manufacturer. It is a modular, real-time, flight control application written in Assembly language. It is a standalone, memory-resident application. It does not rely on an operating system. The Assembly language uses simple constructs and does not rely on any library functions. Diagnostics and self-checks are included.

The code consists of 143 files of subroutines, averaging about 164 lines of executable code per file. Many of the files contain additional internal and external subroutines. The files were categorized by the developer into 84 nonessential files and 59 essential files. Each file is documented with a header of information of consistent format. The code is well commented throughout. The majority of the files are fairly simple, but many are very complex. The files of the essential code are about 10 percent larger and more complex than the nonessential code.

Two of the essential modules were copied, renamed, and arbitrarily categorized as critical modules, for the sake of providing a more complete example.

The 84 modules of nonessential code had the Halstead and McCabe measures shown in the following table. Two modules are not listed because they contained no executable statements. The total scores are the measures of the code when the modules are treated as one large module.

	η_1	η_2	N_1	N_2	N	$\hat{N}$	V	E	v(G)	Li	St	$\hat{L}$	I	$\hat{B}$
NONESS01	35	98	573	447	1020	828	7196*	574426	30*	355*	275	0.013	90.2	2.2
NONESS02	21	39	135	116	251	298	1483	46304	13	87	65	0.032	47.5	0.5
NONESS03	23	29	187	154	341	245	1944	118709	7	111	100	0.016	31.8	0.6
NONESS04	27	30	129	92	221	276	1289	53367	8	72	61	0.024	31.1	0.4
NONESS05	29	36	179	138	317	327	1909	106114	12	111	86	0.018	34.3	0.6
NONESS06	12	17	61	48	109	113	530	8971	1	37	28	0.059	31.3	0.2
NONESS07	15	45	211	157	368	306	2174	56879	18	121	90	0.038	83.1	0.7
NONESS08	11	10	20	13	33	71	145	1036	1	13	8	0.140	20.3	0.0
NONESS09	13	18	49	50	99	123	490	8856	4	41	29	0.055	27.2	0.2
NONESS10	13	16	47	48	95	112	462	8999	5	41	29	0.051	23.7	0.1

	η_1	η_2	N_1	N_2	N	$\hat{N}$	V	E	v(G)	Li	St	$\hat{L}$	I	$\hat{B}$
NONESS11	23	72	439	318	757	548	4973*	252606	19	248*	195	0.020	97.9	1.6
NONESS12	27	87	528	368	896	689	6122*	349603	29*	318*	246	0.018	107.2	1.9
NONESS13	21	94	366	270	636	708	4354*	131307	11	189	156	0.033	144.4	1.4
NONESS14	15	20	77	57	134	145	687	14692	1	52	38	0.047	32.2	0.2
NONESS15	25	29	120	90	210	257	1209	46882	9	70	55	0.026	31.2	0.4
NONESS16	11	13	45	33	78	86	358	4993	1	57	27	0.072	25.6	0.1
NONESS17	15	11	33	32	65	97	306	6666	5	33	23	0.046	14.0	0.1
NONESS18	26	53	204	167	371	426	2339	95799	14	124	99	0.024	57.1	0.7
NONESS19	23	62	294	242	536	473	3435	154207	15	191	139	0.022	76.5	1.1
NONESS20	27	86	531	403	934	681	6370*	402980	50*	323*	235	0.016	100.7	2.0
NONESS21	27	36	158	124	282	314	1686	78380	8	91	76	0.022	36.2	0.5
NONESS22	15	16	53	40	93	123	461	8639	3	35	26	0.053	24.6	0.1
NONESS23	21	43	238	196	434	326	2604	124629	16	128	110	0.021	54.4	0.8
NONESS24	22	32	118	95	213	258	1226	40030	7	78	55	0.031	37.5	0.4
NONESS25	40	55	276	221	497	531	3265	262405	21	173	141	0.012	40.6	1.0
NONESS26	9	10	30	20	50	62	212	1912	1	19	13	0.111	23.6	0.1
NONESS27	43	61	391	304	695	595	4657*	498965	24*	224*	191	0.009	43.5	1.5
NONESS28	25	77	385	333	718	599	4791*	258983	36*	230*	183	0.018	88.6	1.5
NONESS29	15	17	65	48	113	128	565	11965	1	41	32	0.047	26.7	0.2
NONESS30	16	1	29	1	30	64	123	981	1	19	15	0.125	15.3	0.0
NONESS31	12	5	18	7	25	55	102	858	2	16	9	0.119	12.2	0.0
NONESS32	11	28	120	106	226	173	1195	24871	14	95	59	0.048	57.4	0.4
NONESS33	18	26	78	57	135	197	737	14542	3	53	35	0.051	37.4	0.2
NONESS34	12	5	18	7	25	55	102	858	2	14	9	0.119	12.2	0.0
NONESS35	18	33	170	136	306	242	1736	64381	3	103	83	0.027	46.8	0.5
NONESS36	15	18	68	56	124	134	626	14595	5	55	34	0.043	26.8	0.2
NONESS37	18	26	87	67	154	197	841	19499	9	58	39	0.043	36.3	0.3
NONESS38	7	6	17	13	30	35	111	842	1	13	8	0.132	14.6	0.0
NONESS39	14	22	64	49	113	151	584	9108	5	35	28	0.064	37.5	0.2
NONESS40	15	6	29	9	38	74	167	1878	3	19	13	0.089	14.8	0.1
NONESS41	26	69	299	234	533	544	3502	154381	18	195	144	0.023	79.4	1.1
NONESS42	28	35	204	123	327	314	1955	96165	11	108	92	0.020	39.7	0.6
NONESS43	31	18	98	42	140	229	786	28429	6	65	57	0.028	21.7	0.2
NONESS44	14	39	113	86	199	259	1140	17595	3	52	45	0.065	73.8	0.4
NONESS45	22	30	110	81	191	245	1089	32337	12	66	49	0.034	36.7	0.3
NONESS46	27	38	189	151	340	328	2048	109843	19	115	89	0.019	38.2	0.6
NONESS47	20	15	72	41	113	145	580	15843	10	46	33	0.037	21.2	0.2
NONESS48	5	1	7	1	8	12	21	52	1	8	4	0.400	8.3	0.0
NONESS49	18	39	132	104	236	281	1377	33037	7	90	60	0.042	57.4	0.4
NONESS50	29	86	498	388	886	694	6065*	396771	24*	313*	234	0.015	92.7	1.9

	η_1	η_2	N_1	N_2	N	$\hat{N}$	V	E	v(G)	Li	St	$\hat{L}$	I	$\hat{B}$
NONESS51	30	102	528	419	947	828	6671*	411054	53*	318*	232	0.016	108.3	2.1
NONESS52	11	26	106	82	188	160	979	16988	4	54	44	0.058	56.5	0.3
NONESS53	5	7	14	11	25	31	90	352	1	11	6	0.255	22.8	0.0
NONESS54	34	55	274	217	491	491	3180	213263	14	184	134	0.015	47.4	1.0
NONESS55	28	60	328	251	579	489	3740	219040	15	205	163	0.017	63.9	1.2
NONESS56	34	96	564	416	980	805	6882*	506968	34*	330*	258	0.014	93.4	2.2
NONESS57	27	54	382	270	652	439	4134*	279017	4	239*	192	0.015	61.2	1.3
NONESS58	12	31	153	109	262	197	1422	29993	6	76	64	0.047	67.4	0.4
NONESS59	26	34	200	169	369	295	2180	140843	6	249*	101	0.015	33.7	0.7
NONESS60	26	36	156	116	272	308	1620	67841	13	99	75	0.024	38.7	0.5
NONESS61	26	56	186	168	354	447	2251	87772	12	121	95	0.026	57.7	0.7
NONESS62	18	29	89	75	164	216	911	21203	4	59	45	0.043	39.1	0.3
NONESS63	35	68	375	306	681	593	4554*	358589	24*	281*	182	0.013	57.8	1.4
NONESS64	30	60	289	232	521	502	3382	196171	21	167	133	0.017	58.3	1.1
NONESS65	8	15	47	36	83	83	375	3604	2	28	20	0.104	39.1	0.1
NONESS66	8	16	49	37	86	88	394	3647	2	31	23	0.108	42.6	0.1
NONESS67	10	19	79	60	139	114	675	10662	4	48	34	0.063	42.8	0.2
NONESS68	34	54	275	217	492	484	3178	217107	14	183	135	0.015	46.5	1.0
NONESS69	27	46	214	174	388	382	2402	122641	9	140	102	0.020	47.0	0.8
NONESS70	37	94	585	439	1024	809	7202*	622265	33*	329*	260	0.012	83.4	2.3
NONESS71	22	42	253	171	424	325	2544	113935	2	161	123	0.022	56.8	0.8
NONESS72	33	45	156	104	260	414	1634	62318	7	99	74	0.026	42.9	0.5
NONESS73	26	38	154	116	270	322	1620	64288	13	97	73	0.025	40.8	0.5
NONESS74	23	27	97	75	172	232	971	31010	4	66	49	0.031	30.4	0.3
NONESS75	14	12	38	31	69	96	324	5865	4	33	19	0.055	17.9	0.1
NONESS76	7	12	25	22	47	63	200	1281	1	20	12	0.156	31.1	0.1
NONESS77	16	41	186	148	334	284	1948	56260	15	113	90	0.035	67.5	0.6
NONESS78	12	14	45	30	75	96	353	4533	2	26	20	0.078	27.4	0.1
NONESS79	34	79	525	392	917	671	6254*	527561	36*	365*	254	0.012	74.1	2.0
NONESS80	32	65	414	307	721	551	4759*	359599	12	274*	194	0.013	63.0	1.5
NONESS81	1	2	1	2	3	2	5	2	1	188	1	2.000	9.5	0.0
NONESS82	5	1	7	2	9	12	23	116	1	8	4	0.200	4.7	0.0
Average	21	38	185	141	326	305	2061	116121	11	119	87	0.075	46.8	0.6
Std Dev	9	26	161	123	284	222	1994	154596	11	98	74	0.222	27.1	0.6

Total Scores

η_1	η_2	N_1	N_2	v(G)	St	LOC
129	1418	15141	11602	821	7161	15561

The 59 modules of essential code had the Halstead and McCabe measures shown in the following table. Two modules are not listed because they contained no executable statements. The total scores are the measures of the code when the modules are treated as one large module.

	η_1	η_2	N_1	N_2	N	$\hat{N}$	V	E	v(G)	Li	St	$\hat{L}$	I	$\hat{B}$
ESSENT01	19	22	79	62	141	179	755	20225	6	47	37	0.037	28.2	0.2
ESSENT02	33	54	264	209	473	477	3048	194618	21	171	122	0.016	47.7	1.0
ESSENT03	27	21	103	69	172	221	961	42610	6	64	55	0.023	21.7	0.3
ESSENT04	23	19	78	62	140	185	755	28330	4	98	39	0.027	20.1	0.2
ESSENT05	9	9	23	18	41	57	171	1539	4	21	17	0.111	19.0	0.1
ESSENT06	8	8	17	15	32	48	128	960	4	18	12	0.133*	17.1	0.0
ESSENT07	6	2	9	3	12	18	36	162	2	12	8	0.222*	8.0	0.0
ESSENT08	22	31	148	116	264	252	1512	62243	12	97	70	0.024	36.7	0.5
ESSENT09	17	29	109	89	198	210	1094	28530	5	83	49	0.038	41.9	0.3
ESSENT10	19	85	318	224	542	626	3632	90919	3	200	143	0.040	145.1	1.1
ESSENT11	25	65	257	165	422	508	2740	86928	4	155	115	0.032	86.3	0.9
ESSENT12	15	40	155	128	283	271	1636	39267	17	97	69	0.042	68.2	0.5
ESSENT13	8	17	38	30	68	93	316	2229	1	24	17	0.142*	44.7	0.1
ESSENT14	27	51	239	184	423	418	2659	129496	17	141	106	0.021	54.6	0.8
ESSENT15	22	40	198	164	362	311	2155	97209	16	133	92	0.022	47.8	0.7
ESSENT16	13	51	329	261	590	337	3540	117757	38*	227	146	0.030	106.4	1.1
ESSENT17	63	108	686	476	1162	1106	8620*	1196680	30*	403*	317	0.007	62.1	2.7
ESSENT18	13	26	93	83	176	170	930	19302	9	197	49	0.048	44.8	0.3
ESSENT19	19	14	59	49	108	134	545	18114	5	45	37	0.030	16.4	0.2
ESSENT20	6	5	8	8	16	27	55	266	1	9	5	0.208*	11.5	0.0
ESSENT21	18	32	117	99	216	235	1219	33944	6	101	59	0.036	43.8	0.4
ESSENT22	50	56	289	226	515	607	3465	349582	61*	192	170	0.010	34.3	1.1
ESSENT23	14	13	35	31	66	101	314	5238	6	31	22	0.060	18.8	0.1
ESSENT24	11	9	23	21	44	67	190	2440	4	19	14	0.078	14.8	0.1
ESSENT25	36	86	549	375	924	739	6404*	502640	33*	376*	284	0.013	81.6	2.0
ESSENT26	14	15	33	33	66	112	321	4938	5	27	20	0.065	20.8	0.1
ESSENT27	11	9	15	15	30	67	130	1189	3	15	10	0.109	14.1	0.0
ESSENT28	6	5	11	11	22	27	76	502	1	11	7	0.152*	11.5	0.0
ESSENT29	5	5	12	10	22	23	73	365	1	12	8	0.200*	14.6	0.0
ESSENT30	6	5	7	6	13	27	45	162	2	9	5	0.278*	12.5	0.0
ESSENT31	29	69	394	306	700	562	4630*	297748	24	221	181	0.016	72.0	1.4
ESSENT32	27	46	209	162	371	382	2296	109180	14	131	100	0.021	48.3	0.7
ESSENT33	37	79	451	374	825	691	5658*	495526	39*	297*	218	0.011	64.6	1.8
ESSENT34	11	57	192	146	338	371	2058	28986	3	116	86	0.071	146.1	0.6
ESSENT35	28	53	158	119	277	438	1756	55202	6	94	74	0.032	55.9	0.5

	η_1	η_2	N_1	N_2	N	$\hat{N}$	V	E	v(G)	Li	St	$\hat{L}$	I	$\hat{B}$
ESSENT36	48	64	450	230	680	652	4629*	399251	19	278*	224	0.012	53.7	1.4
ESSENT37	13	26	52	46	98	170	518	5957	2	37	29	0.087	45.0	0.2
ESSENT38	16	50	150	153	303	346	1831	44834	12	133	88	0.041	74.8	0.6
ESSENT39	26	56	284	221	505	447	3211	164713	20	168	135	0.019	62.6	1.0
ESSENT40	21	37	184	146	330	285	1933	80094	21	143	101	0.024	46.7	0.6
ESSENT41	27	55	292	231	523	446	3325	188527	19	188	146	0.018	58.6	1.0
ESSENT42	37	72	423	350	773	637	5232*	470499	37*	282*	203	0.011	58.2	1.6
ESSENT43	13	17	40	32	72	118	353	4323	3	23	19	0.082	28.9	0.1
ESSENT44	10	10	27	25	52	66	225	2809	3	18	11	0.080	18.0	0.1
ESSENT45	15	49	327	247	574	334	3444	13C204	5	183	151	0.026	91.1	1.1
ESSENT46	38	71	485	336	821	636	5557*	499632	22	270*	222	0.011	61.8	1.7
ESSENT47	23	50	221	167	388	386	2402	92247	13	130	103	0.026	62.5	0.8
ESSENT48	28	44	220	150	370	375	2283	108955	13	124	94	0.021	47.8	0.7
ESSENT49	34	55	247	169	416	491	2694	140720	19	153	114	0.019	51.6	0.8
ESSENT50	29	85	498	387	885	686	6047*	399216	40*	306*	228	0.015	91.6	1.9
ESSENT51	25	51	240	193	433	405	2705	127974	21	143	109	0.021	57.2	0.8
ESSENT52	29	68	435	318	753	555	4970*	336992	32*	268*	193	0.015	73.3	1.6
ESSENT53	26	32	107	79	186	282	1090	34969	4	78	53	0.031	34.0	0.3
ESSENT54	33	74	431	354	785	626	5292*	417715	39*	309*	223	0.013	67.0	1.7
ESSENT55	24	41	178	143	321	330	1933	80911	12	110	80	0.024	46.2	0.6
ESSENT56	25	37	126	98	224	309	1334	44158	8	74	57	0.030	40.3	0.4
ESSENT57	32	52	290	233	523	456	3343	239680	9	192	145	0.014	46.6	1.0
Average	22	41	200	152	352	336	2250	141744	14	132	96	0.053	49.1	0.7
Std Dev	12	25	163	119	281	231	1980	205466	13	100	77	0.059	29.6	0.6

Total Scores

η_1	η_2	N_1	N_2	v(G)	St	LOC
157	1059	11393	8676	730	5491	11656

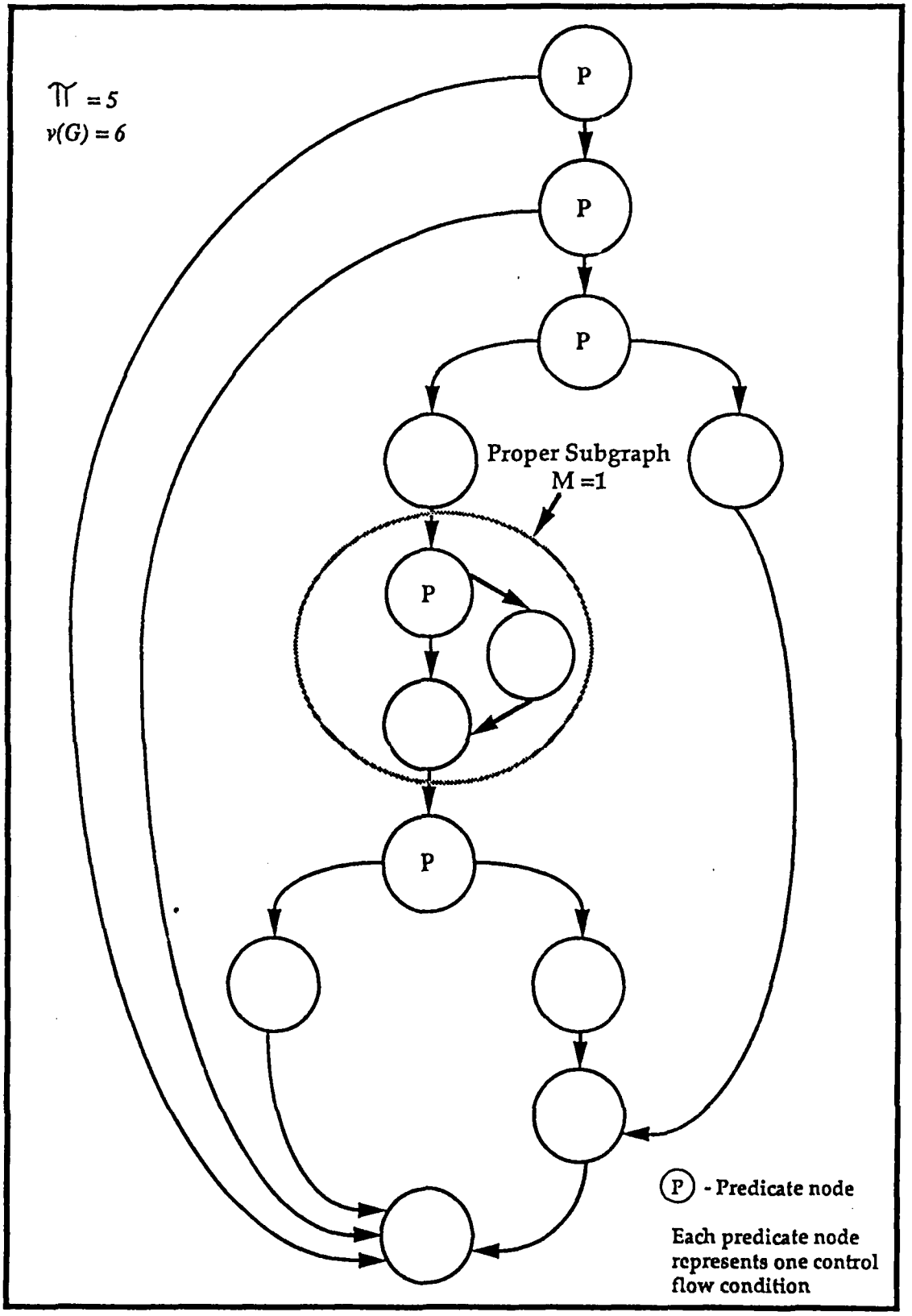
The two modules of critical code had the Halstead and McCabe measures shown in the following table. The total scores are the measures of the code when the modules are treated as one large module.

	η_1	η_2	N_1	N_2	N	$\hat{N}$	V	E	v(G)	Li	St	$\hat{L}$	I	$\hat{B}$
CRITIC01	19	22	79	62	141	179	755	20225	6	47	37	0.037	28.2	0.2
CRITIC02	25	37	126	98	224	309	1334	44158	8	74	57	0.030	40.3	0.4
Average	22	30	103	80	183	244	1045	32192	7	61	47	0.034	34.3	0.3
Std Dev	3	8	24	18	42	65	290	11967	1	14	10	0.004	6.0	0.1

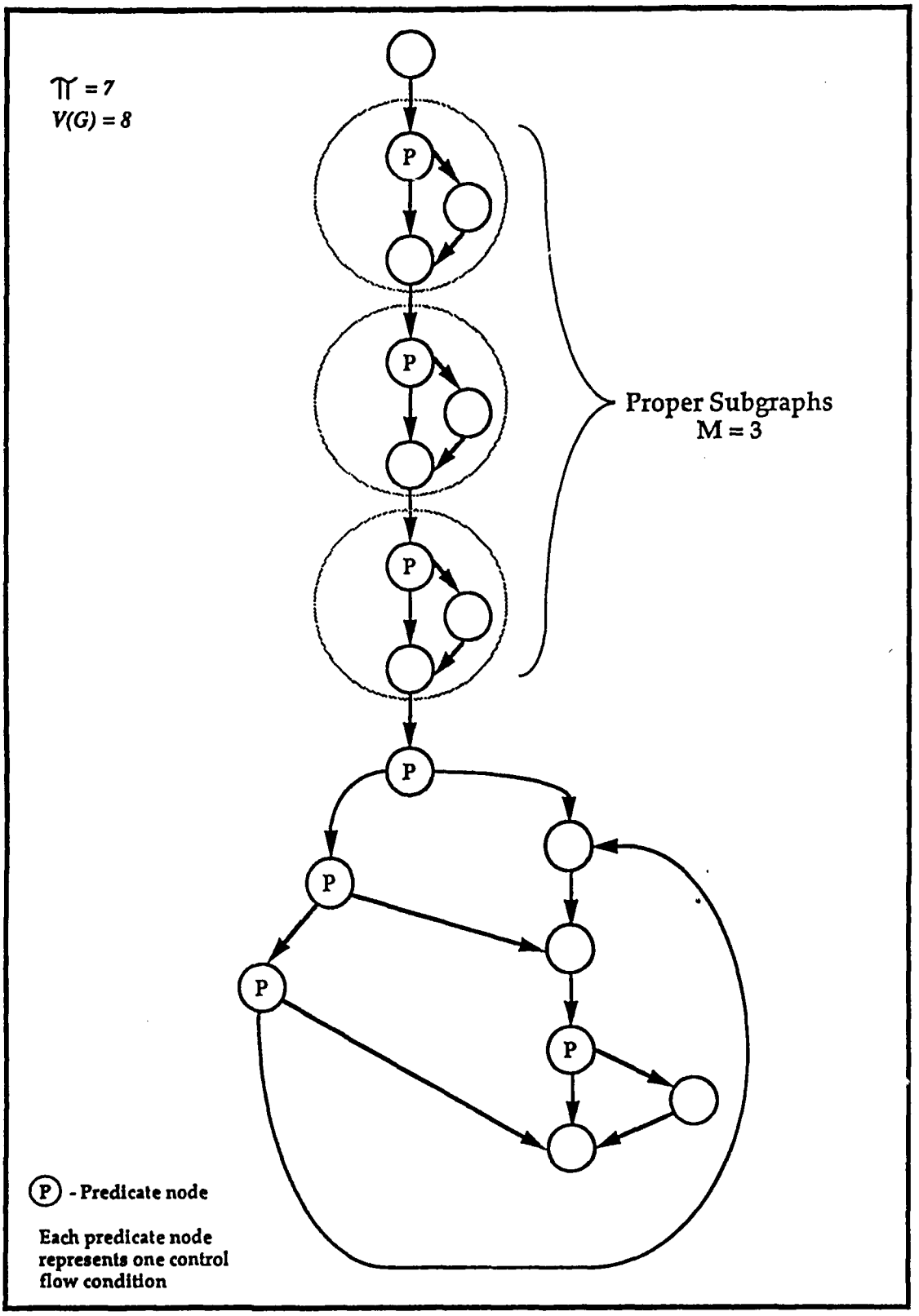
Total Scores

η_1	η_2	N_1	N_2	v(G)	St	LOC
28	51	205	160	13	94	220

The program control graphs for CRITIC01 and CRITIC02 are shown on the following two pages.



PROGRAM CONTROL GRAPH OF CRITICO1



PROGRAM CONTROL GRAPH OF CRITICO2

The 84 modules of nonessential code had the RADC measures shown in the following tables.

	MO.1(3)	MO.1(7)	SD.2(1)	GE.1(1)	FS.1(2)
NONESS01	0	1	0	0	1
NONESS02	1	1	1	0	1
NONESS03	0	1	1	0	1
NONESS04	1	1	1	0	1
NONESS05	0	1	1	0	1
NONESS06	1	1	1	0	1
NONESS07	0	1	0	0	1
NONESS08	1	1	1	0	1
NONESS09	1	1	0	1	1
NONESS10	1	1	0	0	1
NONESS11	0	1	1	0	1
NONESS12	0	1	1	0	1
NONESS13	0	1	1	0	1
NONESS14	1	1	1	0	1
NONESS15	1	1	1	1	1
NONESS16	1	0	0	1	1
NONESS17	1	1	0	0	1
NONESS18	0	1	1	0	1
NONESS19	0	0	0	0	1
NONESS20	0	1	0	0	1
NONESS21	1	1	1	0	1
NONESS22	1	1	0	0	1
NONESS23	0	0	0	0	1
NONESS24	1	1	1	0	1
NONESS25	0	1	1	0	1
NONESS26	1	1	0	0	1
NONESS27	0	1	1	1	1
NONESS28	0	1	1	1	1
NONESS29	1	1	1	0	1
NONESS30	1	1	0	0	1
NONESS31	1	1	0	0	1
NONESS32	1	1	0	0	1
NONESS33	1	1	0	0	1
NONESS34	1	1	0	0	1
NONESS35	0	1	1	0	1
NONESS36	1	1	1	0	1
NONESS37	1	1	1	0	1
NONESS38	1	1	0	0	1
NONESS39	1	1	1	0	1
NONESS40	1	1	0	0	1

	MO.1(3)	MO.1(7)	SD.2(1)	GE.1(1)	FS.1(2)
NONESS41	0	1	1	0	1
NONESS42	0	1	0	0	1
NONESS43	1	1	1	0	1
NONESS44	1	1	0	0	1
NONESS45	1	1	1	0	1
NONESS46	0	1	1	0	1
NONESS47	1	1	0	0	1
NONESS48	1	1	0	0	1
NONESS49	1	1	1	0	1
NONESS50	0	1	0	0	1
NONESS51	0	1	0	0	1
NONESS52	1	1	0	1	1
NONESS53	1	1	0	0	1
NONESS54	0	1	1	1	1
NONESS55	0	1	1	0	1
NONESS56	0	1	1	0	1
NONESS57	0	1	0	0	1
NONESS58	1	1	0	0	1
NONESS59	0	0	1	0	1
NONESS60	1	1	1	0	1
NONESS61	0	1	1	1	1
NONESS62	1	1	1	0	1
NONESS63	0	0	0	0	1
NONESS64	0	0	0	0	1
NONESS65	0	1	1	0	1
NONESS66	0	1	1	0	1
NONESS67	1	1	0	1	1
NONESS68	1	1	0	1	1
NONESS69	1	1	0	0	1
NONESS70	0	1	1	0	1
NONESS71	0	1	1	0	1
NONESS72	0	1	1	0	1
NONESS73	0	1	0	0	1
NONESS74	0	1	1	0	1
NONESS75	1	1	1	0	1
NONESS76	1	1	0	0	1
NONESS77	1	1	0	0	1
NONESS78	1	0	0	0	1
NONESS79	0	1	0	0	1
NONESS80	1	1	1	1	1

	MO.1(3)	MO.1(7)	SD.2(1)	GE.1(1)	FS.1(2)
NONESS81	0	1	1	0	1
NONESS82	0	1	1	0	1
NONESS83	0	0	0	0	1
NONESS84	1	0	0	0	1
Total	45	75	44	11	84

Modularity, MO = MO.1(3) & MO.1(7)

MO.1(3)	45/ 84 =	0.54
MO.1(7)	75/ 84 =	0.89

DS 120/168 = DR 0.71 SO 0.71

Maintainability = MO & SD

MO	120/168 =	0.71 x 0.5 = 0.36
SD	44/ 84 =	0.52 x 0.5 = 0.26

DS 164/252 = DR 0.65 SO 0.62 WSO 0.62

Expandability = GE & MO & SD

GE	11/ 84 =	0.13 x 0.6 = 0.08
MO	120/168 =	0.71 x 0.2 = 0.14
SD	44/ 84 =	0.52 x 0.2 = 0.10

DS 175/336 = DR 0.52 SO 0.46 WSO 0.33

Reusability = FS & GE & MO & SD

FS	84/ 84 =	1.00 x 0.3 = 0.30
GE	11/ 84 =	0.13 x 0.3 = 0.04
MO	120/168 =	0.71 x 0.2 = 0.14
SD	44/ 84 =	0.52 x 0.2 = 0.10

DS 259/420 = DR 0.62 SO 0.59 WSO 0.59

The 59 modules of essential code had the RADC measures shown in the following tables.

	MO.1(3)	MO.1(7)	SD.2(1)	GE.1(1)	FS.1(2)
ESSENT01	1	1	0	0	1
ESSENT02	0	1	0	0	1
ESSENT03	1	1	1	0	1
ESSENT04	1	0	0	0	1
ESSENT05	1	1	0	0	1
ESSENT06	1	0	0	0	1
ESSENT07	1	1	0	1	1
ESSENT08	1	1	0	0	1
ESSENT09	1	1	0	0	1
ESSENT10	0	1	0	0	1
ESSENT11	0	1	0	0	1
ESSENT12	1	1	0	0	1
ESSENT13	1	1	0	0	1
ESSENT14	0	1	0	0	1
ESSENT15	0	0	0	0	1
ESSENT16	0	1	0	0	1
ESSENT17	0	0	0	0	1
ESSENT18	0	1	0	0	1
ESSENT19	0	0	0	0	1
ESSENT20	1	1	0	1	1
ESSENT21	1	1	0	0	1
ESSENT22	1	1	0	0	1
ESSENT23	0	1	0	1	1
ESSENT24	1	1	0	1	0
ESSENT25	1	1	0	1	0
ESSENT26	0	0	0	0	1
ESSENT27	1	1	0	0	1
ESSENT28	1	1	0	1	0
ESSENT29	1	1	0	1	1
ESSENT30	1	1	0	1	1
ESSENT31	1	1	0	1	1
ESSENT32	0	1	0	0	1
ESSENT33	0	1	0	0	1
ESSENT34	0	1	0	0	1
ESSENT35	0	1	0	0	1
ESSENT36	0	0	0	0	1
ESSENT37	0	1	0	0	1
ESSENT38	1	0	0	0	1
ESSENT39	1	0	0	0	1
ESSENT40	0	1	0	0	1

	MO.1(3)	MO.1(7)	SD.2(1)	GE.1(1)	FS.1(2)
ESSENT41	0	1	0	1	1
ESSENT42	0	1	0	0	1
ESSENT43	0	1	0	0	1
ESSENT44	1	1	0	1	1
ESSENT45	1	0	0	0	1
ESSENT46	0	1	0	0	1
ESSENT47	0	1	0	0	1
ESSENT48	0	1	0	0	1
ESSENT49	0	1	0	0	1
ESSENT50	0	1	0	0	1
ESSENT51	0	1	0	0	1
ESSENT52	0	1	0	0	1
ESSENT53	0	1	0	0	1
ESSENT54	1	0	0	0	1
ESSENT55	1	1	0	0	1
ESSENT56	0	1	0	0	1
ESSENT57	1	1	0	0	1
ESSENT58	1	1	0	0	1
ESSENT59	0	1	0	0	1
Total	28	48	1	11	56

Modularity, MO = MO.1(3) & MO.1(7)

MO.1(3)	28/ 59 =	0.47
MO.1(7)	48/ 59 =	0.81
DS	76/118 = DR 0.64	SO 0.64

Maintainability = MO & SD

MO	76/118 =	0.64 x 0.5 = 0.32
SD	1/ 59 =	0.02 x 0.5 = 0.01
DS	77/177 = DR 0.44	SO 0.33 WSO 0.33

Expandability = GE & MO & SD

GE	11/ 59 =	0.19 x 0.6 = 0.11
MO	76/118 =	0.64 x 0.2 = 0.13
SD	1/ 59 =	0.02 x 0.2 = 0.00
DS	88/236 = DR 0.37	SO 0.28 WSO 0.24

Reusability = FS & GE & MO & SD

FS	56/ 59 =	0.95 x 0.3 = 0.28
GE	11/ 59 =	0.19 x 0.3 = 0.06
MO	76/118 =	0.64 x 0.2 = 0.13
SD	1/ 59 =	0.02 x 0.2 = 0.00
DS	144/295 = DR 0.49	SO 0.45 WSO 0.47

The two modules of critical code had the RADC measures shown in the following tables.

	MO.1(3)	MO.1(7)	SD.2(1)	GE.1(1)	FS.1(2)
CRITIC01	1	1	0	0	1
CRITIC02	1	1	0	0	1
Total	2	2	0	0	2

Modularity, MO = MO.1(3) & MO.1(7)

MO.1(3)	2/ 2 =	1.00	
MO.1(7)	2/ 2 =	1.00	
DS	4/ 4 = DR 1.00	SO 1.00	

Maintainability = MO & SD

MO	4/ 4 =	1.00 x 0.5 = 0.50	
SD	0/ 2 =	0.00 x 0.5 = 0.00	
DS	4/ 6 = DR 0.67	SO 0.50	WSO 0.50

Expandability = GE & MO & SD

GE	0/ 2 =	0.00 x 0.6 = 0.00	
MO	4/ 4 =	1.00 x 0.2 = 0.20	
SD	0/ 2 =	0.00 x 0.2 = 0.00	
DS	4/ 8 = DR 0.50	SO 0.33	WSO 0.20

Reusability = FS & GE & MO & SD

FS	2/ 2 =	1.00 x 0.3 = 0.30	
GE	0/ 2 =	0.00 x 0.3 = 0.00	
MO	4/ 4 =	1.00 x 0.2 = 0.20	
SD	0/ 2 =	0.00 x 0.2 = 0.00	
DS	6/ 10 = DR 0.60	SO 0.50	WSO 0.50

BIBLIOGRAPHY

Akiyama, Fumio, "An Example of Software System Debugging", Proceedings IFIP Congress, 1971, pp. 353-358.

Albrecht, A. J., "Measuring Application Development Productivity", Proceedings of the IBM Application Development Symposium, 1979.

Albrecht, Allan J., "Function Points Help Managers Assess Applications", Computerworld, August 26, 1985.

Albrecht, Allan J. and John E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983, pp. 639-648.

Amster, S. J. et al., "An Experiment in Automatic Quality Evaluation of Software", Proceedings of the Symposium on Computer Software Engineering, New York Polytechnic, New York, NY, 1976.

Arthur, J., "Metrics: Tools to Aid Software Code Quality", Computerworld, Volume 17, No. 26, June 27, 1983.

Atwood, M. E. et al., "Annotated Bibliography on Human Factors in Software Development", ARI Technical Report, P-79-1, Science Applications, Inc., Englewood, CO, June 1979.

Bailey, C. T. and W. L. Dingee, "A Software Study Using Halstead Metrics", ACM Performance Evaluation Review - SIGMETRICS, 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, March 1981, pp. 189-197.

Baker, Albert L. and Stuart H. Zweben, "A Comparison of Measures of Control Flow Complexity", Proceedings of COMPSAC 1979, IEEE, 1979.

Baker, Albert L. and Stuart H. Zweben, "The Use of Software Science in Evaluating Modularity Concepts", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979, pp. 110-120.

Basili, V. R. and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory", ACM SIGMETRICS (1981 ACM Workshop/Symposium Measurement and Evaluation of Software Quality), Volume 10, March 1981, pp. 95-106.

Basili, V. R., R. W. Selby, Jr., and T. Y. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983.

Basili, Victor R. and H. Dieter Rombach, "Implementing Quantitative SQA: A Practical Model", IEEE Software, Volume 4, No. 5, September 1987, pp. 6-9.

Basili, Victor R. and David H. Hutchens, "An Empirical Study of a Syntactic Complexity Family", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983, pp. 664-672.

Basili, Victor R. and Robert W. Reiter, Jr., "Evaluating Automatable Measures of Software Development", Department of Computer Science, University of Maryland, College Park, MD, 1979.

Behrens, C., "Measuring the Productivity of Computer Systems Development Activities with Function Points", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983, pp. 648-652.

Bell, D. E. and J. E. Sullivan, Further Investigations into the Complexity of Software, MTR-2874, MITRE, Bedford, MA, 1974.

Boehm, B. W. et al., Proceedings of the TRW Symposium on Reliable, Cost-Effective, Secure Software, Los Angeles, CA, March 20-21, 1974.

Boehm, B. W. et al., Characteristics of Software Quality, North-Holland Publishing Company, New York, NY, 1978.

Bowen, Thomas P., Gary B. Wigle, and Jay T. Tsai, Specification of Software Quality Attributes, RADC-TR-85-37, RADC, Griffiss Air Force Base, NY, Volumes I, II, and III, February 1985.

Brainerd, Walter S., Charles H. Goldberg, and Jonathan L. Gross, FORTRAN 77 Fundamentals and Style, Boyd & Fraser Publishing Co., Boston, MA, 1985.

Browne, Jim C., "A Proposal for Structural Models of Software Systems", Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, Springer-Verlag, Inc., New York, NY, 1981, pp. 208-210.

Buckley, Fletcher J., "Standard Set of Useful Software Metrics is Urgently Needed", Computer, Volume 22, No. 7, July 1989, pp. 88-89.

Bulut, N. and M. H. Halstead, "Impurities Found in Algorithm Implementations", SIGPLAN Notices, Volume 9, No. 3, March 1974, pp. 9-11.

Campbell, D. T. and J. C. Stanley, Experimental and Quasi-Experimental Designs for Research, Houghton Mifflin Company, Boston, MA, 1963.

Card, David N., "Major Obstacles Hinder Successful Measurement", IEEE Software, Volume 5, No. 6, November 1988, pp. 82 and 86.

Carver, D. L., "Criteria for Estimating Module Complexity", Journal of Systems Management, August 1986.

Chen, Edward T., "Program Complexity and Programmer Productivity", IEEE Transactions on Software Engineering, Volume SE-4, No. 3, May 1978, pp. 187-194.

Cote, V. et al., "Software Metrics: An Overview of Recent Results", The Journal of Systems and Software, Volume 8, No. 2, March 1988, pp. 121-131.

Coulter, Neal S., "Software Science and Cognitive Psychology", IEEE Transactions on Software Engineering, Volume SE-9, No. 2, March 1983, pp. 166-171.

Coupal, Daniel and Pierre Robillard, "Factor Analysis of Source Code Metrics", The Journal of Systems and Software, Volume 12, No. 3, July 1990, pp. 263-270.

Crawford, S. G., A. A. McIntosh, and D. Pregibon, "An Analysis of Static Metrics and Faults in C Software", The Journal of Systems and Software, Volume 5, No. 1, February 1985, pp. 37-48.

Currans, Nancy, "A Comparison of Counting Methods for Software Science and Cyclomatic Complexity", Pacific Northwest Software Quality Conference.

Curtis, B., "Measurement and Experimentation in Software Engineering", Proceedings of the IEEE, IEEE, New York, NY, Volume 68, No. 9, September 1980, pp. 1144-1157.

Curtis, B. et al., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979.

Denicoff, Marvin, "Software Metrics: Paradigms and Processes", Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, Springer-Verlag, Inc., New York, NY, 1981, pp. 205-207.

Deutsch, Michael S., "Application of an Automated Verification System" Software Verification and Validation Realistic Project Approaches, Prentice Hall, Englewood Cliffs, NJ, 1982.

Dickson, et al., "Quantitative Analysis of Software Reliability", Proceedings of 1972 Reliability and Maintainability Symposium, Annals of Assurance Science, IEEE, Catalog No. 72CH0577-7R, pp. 148-157.

Drummond, S., "Measuring Applications Development Performance", Datamation, February 1985, pp. 103-108.

Duncan, Otis Dudley, Introduction to Structural Equation Models, Academic Press, Inc., New York, NY, 1975, pp. 1-168.

Dunsmore, H. E., "Software Metrics: An Overview of an Evolving Methodology", Information Processing and Management, Volume 20, No. 1-2, 1984, pp. 183-192.

Ejiogu[1], Lem C., "A Simple Measure of Software Complexity", Computerworld, Volume 18, No. 14, April 2, 1984.

Ejiogu[2], Lem O., "Software Structure: Its Characteristic Polynomials", SIGPLAN Notices, Volume 19, No. 12, December 1984, pp. 1-7.

Ejiogu, Lem O., "The Critical Issues of Software Metrics", Sigplan Notices, Volume 22, No. 3, March 1987, pp. 1-6.

Ejiogu, Lem O., "A Unified Theory of Software Metrics", Softmetrix, Inc., Chicago, IL, 1988, pp. 232-238.

Ejiogu, Lem O., "Beyond Structured Programming: An Introduction to the Principles of Applied Software Metrics", Structured Programming, Springer-Verlag, Inc., New York, NY, 1990.

Elshoff, James L., "A Numerical Profile of Commercial PL/I Programs", GMR-1927, General Motors Corporation Research Laboratories, Warren, MI, September 1975.

Elshoff, James L., "Measuring Commercial PL/I Programs Using Halstead's Criteria", SIGPLAN Notices, Volume 11, No. 5, May 1976, pp. 38-46.

Elshoff, James L., "The PEEK Measurement Program", GMR-4208, General Motors Corporation Research Laboratories, Warren, MI, November 15, 1982.

Elshoff, James L., "Characteristic Program Complexity Measures", GMR-4446R, General Motors Corporation Research Laboratories, Warren, MI, December 5, 1983.

Elwell, D. and N. VanSuetendael, Software Quality Metrics, DOT/FAA/CT-91/1, U.S. Department of Transportation, Federal Aviation Administration, 1991.

Evangelist, W. M., "Software Complexity Metric Sensitivity to Program Structuring Rules", The Journal of Systems and Software, Volume 3, No. 3, September 1983.

Farr, Leonard and Henry J. Zagorski, "Quantitative Analysis of Programming Cost Factors: A Progress Report", Economics of Automatic Data Processing. ICC Symposium Proceedings 1965 Rome, North-Holland, Amsterdam, Holland, 1965, pp. 167-180.

Federal Aviation Regulations, Part 25 Airworthiness Standards: Transport Category Airplanes, Subpart F, Section 25.1309.

Ferdinand, A. E., "A Theory of System Complexity", International Journal of General Systems, Volume 1, No. 1, 1974.

Feuche, Mike, "Attention is Being Generated by Complexity Metrics Tools", MIS Week, Volume 9, No. 9, February 29, 1988.

Feuer, Alan, R. and Edward B. Fowlkes, "Some Results from an Empirical Study on Computer Software", Bell Telephone Laboratories, 1979.

Fisher, R. A., "The Design of Experiments", Oliver and Boyd, London, United Kingdom, 1935.

Fitzsimmons, Ann B., "Relating the Presence of Software Errors to the Theory of Software Science", Proceedings of the Eleventh Hawaii International Conference on Systems Sciences, Western Periodicals, 1978.

Fitzsimmons, Ann B. and Tom L. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys, Volume 10, No. 1, March 1978, pp. 1-18.

Funami, Y. and M. H. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data", Proceedings of the MRI 24th International Symposium: Software Engineering, Polytechnic Press, New York, NY, 1976

Gaffney, Jr., John E., "Software Metrics: A Key to Improved Software Development Management", Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, Pittsburgh, PA, March 12 and 13, 1981, pp. 211-220.

Gaffney, Jr., John E. and Richard R. Reynolds, "Specifying Performance Requirements for a Degradable System - The Case of an Unmanned Weather Station", presented at CMG XIV International Conference on Computer Performance Evaluation, Crystal City, VA, December 1983.

Gaffney, Jr., John E., "Estimating the Number of Faults in Code", IEEE Transactions on Software Engineering, Volume SE-10, No. 4, July 1984, pp. 459-464.

Gaffney, Jr., John E., "The Impact on Software Development Costs of using HOL's", IEEE Transactions on Software Engineering, Volume SE-12, No. 3, March 1986.

Gaffney, Jr., John E. and Charles F. Davis, An Approach to Estimating Software Errors and Availability, SPC-TR-88-007, Software Productivity Consortium, Herndon, VA, March 1988.

Gibson, Virginia R. and James A. Senn, "System Structure and Software Maintenance Performance," Communications of the ACM, March 1989, Volume 32, No. 3, pp. 347-358.

Gilb, T., Software Metrics, Winthrop, Inc., Cambridge, MA, 1977.

Gordon, Ronald D. and Maurice H. Halstead, "An Experiment Comparing FORTRAN Programming Times with the Software Physics Hypothesis", AFIPS Conference Proceedings - National Computer Conference, Volume 45, 1976, pp. 935-937.

Gordon[1], Ronald D., "Measuring Improvements in Program Clarity", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979, pp. 79-90.

Gordon[2], Ronald D., "A Qualitative Justification for a Measure of Program Clarity", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979, pp. 121-128.

Gorla, Narasimhaiah, Alan C. Benander, and Barbara A. Benander, "Debugging Effort Estimation Using Software Metrics", IEEE Transactions on Software Engineering, Volume 16, No. 2, February 1990, pp. 223-231.

Gould, John D., "Some Psychological Evidence on How People Debug Computer Programs", International Journal of Man-Machine Studies, Volume 7, pp. 151-181.

Gremillion, L. L., "Determinants of Program Repair Maintenance Requirements", Communications of the ACM, Volume 27, No. 8, pp. 826ff.

Halstead, Maurice H., "Natural Laws Controlling Algorithm Structure", SIGPLAN Notices, Volume 7, No. 2, 1972.

Halstead, Maurice H. and P. M. Zislis, Experimental Verification of Two Theorems of Software Physics, CSD-TR-97, Purdue University, Lafayette, IN, June 12, 1973.

Halstead, Maurice H., Elements of Software Science, Elsevier North Holland Publishing Company, Inc., New York, NY, 1977, pp. 19-26.

Halstead, Maurice H., "Advances in Software Science", Advances in Computers, Volume 18, NY Academic, 1979.

Harrison[1], W., "MAE: A Syntactic Metric Analysis Environment", The Journal of Systems and Software, Volume 8, 1988, pp. 57-62.

Harrison[2], W., "Using Software Metrics to Allocate Testing Resources", Journal of Management Information Systems, Volume 4, No. 4, Spring 1988.

Harrison, W. A., "Applying McCabe's Complexity Measure to Multiple-Exit Programs", Software - Practice & Experience, Volume 14, No. 10, October 1984.

Harrison, W. and C. Cook, "Are Deeply Nested Conditionals Less Readable?", The Journal of Systems and Software, Volume 6, 1986, pp.335-341.

Harrison, W. and C. Cook, "Insights On Improving the Maintenance Process Through Software Measures", Oregon State University, Computer Science Department, Corvallis, OR, March 1990.

Harrison, Warren and Curtis Cook, "A Reduced Form For Sharing Software Complexity Data", Oregon State University, Computer Science Department, Corvallis, OR, 1983.

Harrison, W. and K. Magel, "A Complexity Measure Based On Nesting Level", The Journal of Systems and Software, Volume 6, 1986, pp. 335-341.

Harrison, W. et al., "Applying Software Complexity Metrics To Program Maintenance", Computer, Volume 15, No. 9, September 1988.

Hays, William L. and Robert L. Winkler, Statistics, Holt, Rinehart, and Winston, Inc., New York, NY, 1970.

Henry, S. and D. Kafura, "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics", Software - Practice and Experience, Volume 14, No. 6, June 1984.

Henry, S., D. Kafura, and K. Harris, "On the Relationships Among Three Software Metrics", 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, March 1981.

IEEE Computer Society, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.1-1988, April 1989.

IEEE Computer Society, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.2-1988, June 1989.

Jones, Capers, "Building a Better Metric", Computerworld, Volume 22, No. 25, June 20, 1988, pp. 38-39.

Jones, T. C., "Measuring Programming Quality and Productivity", IBM Systems Journal, Volume 17, No. 1, 1978, pp. 39-63.

Kafura, Dennis and Geereddy R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance", IEEE Transactions on Software Engineering, Volume SE-13, No. 3, March 1987, pp 335-343.

Keller, Steven E., "Using Metrics to Specify Software Quality", Ada Quality Quarterly, Systems Division of Dynamics Research Corporation, Andover, MA, 1989.

Kelvin, W. T., "Popular Lectures and Addresses, 1981-1984", Workshop on Quantitative Software Models for Reliability, Complexity, and Cost, IEEE, New York, NY, 1980.

Kernighan, B. W. and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, New York, NY, 1974.

Kitchenham, B. A., "Measures of Programming Complexity", ICL Technical Journal, Volume 2, No. 3, 1981, pp. 298ff.

Krause, K. W., R. W. Smith, and M. A. Goodwin, "Optimal Software Test Planning through Automated Network Analysis", Proceedings IEEE Computer Software Reliability Symposium, 1973.

Krause, K. W., R. W. Smith, and M. A. Goodwin, Optimal Software Test Planning through Automated Network Analysis, TRW-SS-73-01, TRW Engineering and Integration Division, Redondo Beach, CA, April 1973, pp. 1-5.

Kreitzberg, Charles B. and Ben Shneiderman, FORTRAN Programming: A Spiral Approach, Second Edition, Harcourt Brace Jovanovich, Inc., 1982.

Kruger, Gregory A., "Project Management Using Software Reliability Growth Models", Hewlett-Packard Journal, Volume 39, No. 3, June 1988, pp. 30-35.

Lasky, Jeffrey A., Alan R. Kaminsky, and Wade Boaz, Software Quality Measurement Methodology Enhancements Study Results, Final Technical Report, RADC-TR-89-317, RADC, Griffiss Air Force Base, NY, January 1990.

Lassez, J.-L., D. Van Der Knijff, J. Shepherd, and C. Lassez, "A Critical Examination of Software Science", The Journal of Systems and Software, Volume 2, December 1981, pp. 105-112.

Lennselius, Bo, Claes Wohlin, and Ctirad Vrana, "Software Metrics: Fault Content Estimation and Software Process Control", Microprocessors and Microsystems, Volume 11, No. 7, September 1987, pp. 365-375.

Lew, Ken S., Tharam S. Dillon, and Kevin E. Forward, "Software Complexity and Its Impact on Software Reliability", IEEE Transactions on Software Engineering, Volume 14, No. 11, November 1988, pp. 1645-1655.

Li, H. F. and W. K. Cheung, "An Empirical Study of Software Metrics", IEEE Transactions on Software Engineering, Volume SE-13, No. 6, June 1987.

Lind, Randy K. and K. Vairavan, "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort", IEEE Transactions on Software Engineering, Volume 15, No. 5, May 1989, pp. 649-653.

Littlewood, B. (edited by), Software Reliability - Achievement and Assessment, Blackwell Scientific Publications, Oxford, United Kingdom, 1987.

Love, L. T. and A. B. Bowman, "An Independent Test of the Theory of Software Physics", SIGPLAN Notices, Volume 11, November 1976, pp. 42-49.

Low, Graham C. and D. Ross Jeffery, "Function Points in the Estimation and Evaluation of the Software Process", IEEE Transactions on Software Engineering, Volume 16, No. 1, January 1990, pp. 64-71.

Mannino, Phoebe, Bob Stoddard, and Tammy Sudduth, "The McCabe Software Complexity Analysis as a Design and Test Tool", Texas Instruments Technical Journal, Volume 7, No. 2, March-April 1990, pp. 41-53.

McAuliffe, Daniel, "Measuring Program Complexity", Computer, Volume 21, No. 6, June 1988, pp. 97-98.

McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, Volume SE-2, No. 4, 1976.

McCabe, Thomas J., Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, U.S. Department of Commerce - National Bureau of Standards Special Publication #500-99, Washington, DC, 1982.

McCabe, Thomas J., "The Cyclomatic Complexity Metric", Hewlett Packard Journal, April 1989.

McCabe, Thomas J. and Charles W. Butler, "Design Complexity Measurement and Testing", Communications of the ACM, December 1989, pp. 1415-1425.

McCall, Jim A., Paul K. Richards, and Gene F. Walters, Factors in Software Quality, RADC-TR-77-369, Volumes I, II, and III, RADC, Griffiss Air Force Base, NY, November 1977.

McClure, Carma L., "A Model for Program Complexity Analysis", Proceedings of the Third International Conference on Software Engineering, IEEE, New York, NY, 1978, p. 149-157.

Millman, P. and B. Curtis, A Matched Project Evaluation of Modern Programming Practices (Management Report), RADC-TR-80-6, Volume 1 of 2, RADC, Griffiss Air Force Base, NY, 1980.

Millman, P. and B. Curtis, A Matched Project Evaluation of Modern Programming Practices (Scientific Report), RADC-TR-80-6, Volume 2 of 2, RADC, Griffiss Air Force Base, NY, 1980.

Mohanty, Siba N., "Models and Measurements for Quality Assessment of Software", Computing Surveys, Volume 11, No. 3, September 1972, pp. 251-275.

Moranda, P. B., "Is Software Science Hard?", (in Surveyors' Forum) ACM Computer Surveys, Volume 10, No. 4, December 1978, pp. 503-504.

Murine[1], Gerald E., "The Application of Software Quality Metrics", Phoenix Conference on Computers and Communications, IEEE, Carlsbad, CA, 1983.

Murine[2], Gerald E., "Improving Management Visibility Through the Use of Software Quality Metrics", Proceedings, Seventh International Computer Software and Applications Conference, Chicago, IL, November 7-11, 1983.

Murine[3], Gerald E., "On Validating Software Quality Metrics", 4th Annual Phoenix Conference, Phoenix, AZ, March 1985.

Murine[4], Gerald E., "The Role of Software Quality Metrics in the Software Quality Evaluation Process", Proceedings, Ninth International Computer Software and Applications Conference, Chicago, IL, October 1985.

Murine, Gerald E., "Software Measurement Techniques", Proceedings, Measurement Science Conference, Long Beach, CA, January 1988.

Murine, Gerald E. and Rita Cerv, "The New Science of Software Quality Metrology", Proceedings, Measurement Science Conference, Irvine, CA, January 1986.

Musa, John D., "Faults, Failures, and a Metrics Revolution", IEEE Software, Volume 6, No. 2, March 1989, pp. 85 and 91.

Myers, G. J., Software Reliability, John Wiley and Sons, Inc., New York, NY, 1976.

Myers, G. J., "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, Volume 12, No. 10, 1977.

Myers, G. J., Composite/Structured Design, Van Nostrand Reinhold Company, New York, NY, 1978.

Myers, G. J., The Art of Software Testing, John Wiley and Sons, Inc. New York, NY, 1979.

Ntafos, Simeon C., "A Comparison of Some Structural Testing Strategies", IEEE Transactions on Software Engineering, Volume 14, No. 6, June 1988, pp. 868-874.

Ohba, M., "Software Reliability Analysis Models", IBM Journal of Research and Development, Volume 28, No. 4, July 1984, pp. 428-443.

Oldehoeft, R. R., "A Contrast Between Language Level Measures", IEEE Transactions on Software Engineering, Volume SE-3, No. 6, November 1977, pp. 476-478.

Ottenstein, Linda M., "Quantitative Estimates of Debugging Requirements", IEEE Transactions on Software Engineering, Volume SE-5, No. 5, September 1979, pp. 504-514.

Ottenstein, Linda M., "Predicting Numbers of Errors Using Software Science", ACM Performance Evaluation Review - SIGMETRICS, 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, March 1981, pp. 157-167.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems Into Modules", Communications of the ACM, Volume 15, No. 12, December 1972.

Perlis, A. J., F. G. Sayward, and M. Shaw (Editors), Software Metrics: An Analysis and Evaluation, MIT Press, Cambridge, MA, 1981.

Perrone, Giovanni, "Program Measures Complexity of Software", PC Week, Volume 5, No. 13, March 29, 1988.

Perry, W. E., "Lack of Measurements Plague IS Assessments", Information Systems News, May 2, 1983.

Pierce, Patricia, Richard Hartley, and Suellen Worrells, Software Quality Measurement Demonstration Project II Final Technical Report, RADC-TR-87-164, RADC, Griffiss Air Force Base, NY, October 1987.

Pippenger, Nicholas, "Complexity Theory", Scientific American, June 1978, pp. 114-124.

Prather, R. E., "Theory of Program Testing - An Overview", Bell System Technical Journal, Volume 62, No. 10, Part 2, December 1983.

Prather, R. E., "An Axiomatic Theory of Software Complexity Measure", The Computer Journal, Volume 27, No. 4, November 1984.

Quantitative Software Management Inc., "Hanscom AFB gets a Handle on Vendor Performance", QSM Perspectives, McLean, VA, Fall 1990.

Radio Technical Commission for Aeronautics, Software Considerations in Airborne Systems and Equipment Certification, Document No. RTCA/DO-178A, March 22, 1985.

Ramamurthy, Bina and Austin Melton, "A Synthesis of Software Science Measures and the Cyclomatic Number", IEEE Transactions on Software Engineering, Volume 14, No. 8, August 1988, pp. 1116-1121.

Reynolds, R. G., "Metrics to Measure the Complexity of Partial Programs", The Journal of Systems and Software, Volume 4, No. 1, April 1984.

Robillard, Pierre N. and Germinal Boloix, "The Interconnectivity Metrics: A New Metric Showing How a Program is Organized", The Journal of Systems and Software, Volume 10, No. 1, July 1989, pp. 29-39.

Schneidewind, N. F. and H. M. Hoffman, "An Experiment in Software Error Data Collection and Analysis", IEEE Transactions on Software Engineering, Volume SE-5, No. 3, May 1979, pp. 276-286.

Selby, Richard W. and Adam A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis", IEEE Transactions on Software Engineering, Volume 14, No. 12, December 1988, pp. 1743-1757.

Shatz, Sol M., "Towards Complexity Metrics for Ada Tasking", IEEE Transactions on Software Engineering, Volume 14, No. 8, August 1988, pp. 1122-1127.

Shen, Vincent Y., Samuel D. Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support", IEEE Transactions on Software Engineering, Volume SE-9, No. 2, March 1983, pp. 155-165.

Shneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishing Co., Cambridge, MA, 1980, pp. 94-120.

Shumskas, Anthony F., "A Layered Software Test and Evaluation Strategy for a Layered Defense System", ITEA Journal, Volume XI, No. 3, 1990.

Siyan, Karanjit S., "Coping with Complex Programs", Dr. Dobb's Journal of Software Tools, Volume 14, No. 3, March 1989.

Society of Automotive Engineers, Fault/Failure Analysis for Digital Systems and Equipment, Aerospace Recommended Practice 1834, August 7, 1986.

Software Engineering Research Review - Quantitative Software Models, Data and Analysis Center for Software (DACS), Griffiss Air Force Base, NY, March 1979.

Sullivan, J. E., "Measuring the Complexity of Computer Software", MTR-2648, MITRE Corporation, Bedford, MA, 1973.

Sunazuka, Toshihiko, Motoei Azuma, and Noriko Yamagishi, "Software Quality Assessment Technology", Proceedings, 8th International Conference on Software Engineering, August 28-30, 1985, pp. 142-148.

Szulewski, Paul A., Mark H. Whitworth, Philip Buchan, and J. Barton DeWolf, "The Measurement of Software Science Parameters in Software Designs", ACM Performance Evaluation Review - SIGMETRICS, 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, March 1981, pp. 89-94.

Taft, Darryl K., "Quality Analysis Aids Ada Programming", Government Computer News, Volume 7, No. 7, April 1, 1988.

Takahashi, Muneo and Yuji Kamayachi, "An Empirical Study of a Model for Program Error Prediction", IEEE Transactions on Software Engineering, Volume 15, No. 1, January 1989, pp. 82-86.

Thayer, T. A., M. Lipow, and E. C. Nelson, "Software Reliability Study", TRW Software Series, TRW-SS-76-03, March 1976.

U.S. Department of the Air Force, Air Force Systems Command Software Management Indicators, AFSCP 800-14, January 20, 1986.

U.S. Department of the Air Force, Air Force Systems Command Software Quality Indicators, AFSCP 800-43, January 31, 1986.

U.S. Department of Defense, Defense System Software Quality Program, Military Standard DOD-STD-2168, August 1, 1979.

U.S. Department of Defense, Defense System Software Development, Military Standard DOD-STD-2167A, June 4, 1985.

U.S. Department of Transportation, Federal Aviation Administration, "System Design and Analysis", Advisory Circular, No. 25.1309-1A, June 1988.

Van Der Knijff, D. J. J., "Software Physics and Program Analysis", The Australian Computer Journal, Volume 10, No. 3, August 1978, pp. 82-86.

Van Der Poel, K. G. and S. R. Schach, "A Software Metric for Cost Estimation and Efficiency Measurement in Data Processing System Development", The Journal of Systems and Software, Volume 3, No. 3, September 1983.

Verilog Products Catalog, Verilog USA Inc., Alexandria, VA, January 1990.

Verilog, Logiscope Automated Code Analyzer, Technical Presentation, December 1989.

Voldman, J. et al., "Fractal Nature of Software - Cache Interaction", IBM Journal of Research and Development, Volume 27, No. 2, March 1983, pp. 164-170.

Walsh, T. J., "A Software Reliability Study Using a Complexity Measure", Proceedings of the National Computer Conference, AFIPS, 1979.

Walters, G. F., "Applications of Metrics to a Software Quality Management (QM) Program", Software Quality Management, Petrocelli, New York, NY, 1979.

Ward, W. T., "Software Defect Prevention Using McCabe's Complexity Metric", Hewlett Packard Journal, April 1989.

Warthman, James L., Software Quality Measurement Demonstration Project I, Final Technical Report, RADC-TR-87-247, RADC, Griffiss Air Force Base, NY, December 1987.

Weyuker, Elaine J., "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, Volume 14, No. 9, September 1988, pp. 1357-1365.

Woodfield, S. N., V. Y. Shen, and H. E. Dunsmore, "A Study of Several Metrics for Programming Effort", Journal of Systems and Software, Volume 2, December 1981, pp. 97-103.

Woodward, Martin R., Michael A. Hennell, and David Hedley, "A Measure of Control Flow Complexity in Program Text", IEEE Transactions on Software Engineering, Volume SE-5, No. 1, January 1979, pp. 45-50.

Yu, Tze-Jie, Vincent Y. Shen, and Hubert E. Dunsmore, "An Analysis of Several Software Defect Models", IEEE Transactions on Software Engineering, Volume 14, No. 9, September 1988, pp. 1261-1270.

Zislis, Paul M., An Experiment in Algorithm Implementation, CSD-TR-96, Department of Computer Science, Purdue University, Lafayette, IN, June 1973.

Zweben, S. H., "A Study of the Physical Structure of Algorithms", IEEE Transactions on Software Engineering, Volume SE-3, No. 3, May 1977, pp. 250-258.

Zweben, Stuart H., Software Physics: Resolution of an Ambiguity in the Counting Procedure, TR 93, Purdue University, Lafayette, IN.

GLOSSARY

ANTAGONISTIC QUALITY FACTORS. Quality Factors with conflicting attributes.

BINARY SEARCH. A searching algorithm in which the search population is repeatedly divided into two equal or nearly equal sections.

BITS. Binary digits.

CODE. The subset of software which exists for the sole purpose of being loaded into a computer to control it.

COMPLEMENTARY QUALITY FACTORS. Quality Factors with interrelated attributes.

CONTROL STRUCTURES. Programming constructs which direct the flow of control.

EQUIVALENCE STATEMENT. A FORTRAN statement which equates two variable names.

HARDWARE. The physical components of a computer.

METRIC. A measure.

MODULE. A unit of code which implements a function.

MONOTONIC FUNCTION. A function in which a certain change in the measure always represents a certain change in the property being measured, where either change is simply an increase or decrease in magnitude.

OBJECT CODE. The translation of source code that is loaded into a computer.

OPERANDS. The variables or constants on which the operators act.

OPERATORS. Symbols which affect the value or ordering of operands.

OPTIMIZING COMPILER. A computer program which, while translating source code into object code, removes inefficiencies from the code.

PROGRAM. A detailed set of instructions for accomplishing some purpose.

QUALITY MEASURE. A repeatable, monotonic relationship relating measures of objects (a set of numbers) to subjective qualities.

SOFTWARE METRIC. A measure of software objects.

SOFTWARE QUALITY FACTOR. Any software attribute that contributes either directly or indirectly, positively or negatively, toward the objectives for the system in which the software resides.

SOFTWARE QUALITY METRIC. (1) A measure that relates measures of the software objects (the symbols) to the software qualities (quality factors). (2) The measure of a software quality factor.

SOFTWARE. Computer programs and the documentation associated with the programs.

SOURCE CODE. Code that can be read by people.

STROUD NUMBER. The total number of elementary mental discriminations that a person makes per second.

SUBROUTINE. A self-contained body of code which can be called by other routines to perform a function.

WELL-BEHAVED FUNCTION. A smooth mathematical relationship.

ACRONYMS AND ABBREVIATIONS

λ	Language Level
$1/E_0$	Average number of discriminations a person is likely to make for each bug introduced into the code.
AC	Advisory Circular
ACT	Analysis of Complexity Tool
ARP	Aerospace Recommended Practice
$\hat{B}$	Number of Bugs (Estimated)
BAT	Battlemap Analysis Tool
CE	Certification Engineer
CFR	Code of Federal Regulations
CR/LF	Carriage Return/Line Feed
CSCI	Computer Software Configuration Item
D	Program Difficulty
DR	Direct Ratio (Average)
DS	Direct Score
E	Programming Effort
EOF	End of File
FAA	Federal Aviation Administration
FAR	Federal Aviation Regulation
FP	Function Point
HOL	High Order Language
I	Intelligence Content
IEEE	Institute of Electrical and Electronics Engineers
IFC	Information Flow Complexity
ISO	International Standards Organization
L	Program Level
$\hat{L}$	Estimated Program Level
$\hat{N}$	Estimated Length
η	Vocabulary of a Program
η_1	Number of Unique Operators
η_2	Number of Unique Operands
η_1^*	Minimum Number of Unique Operators
η_2^*	Number of Different Input and Output Parameters
N	Implementation Length of a Program
N_1	Total Number of Operator Occurrences
N_2	Total Number of Operand Occurrences
N_1^*	Minimum Number of Operators
N_2^*	Minimum Number of Operands
PC	Processing Complexity
PCA	Processing Complexity Adjustment
PROM	Programmable Read-Only Memory
RADC	Rome Air Development Center
RAM	Random-Access Memory
RTCA	Radio Technical Commission for Aeronautics
S	Stroud Number

SAE	Society of Automotive Engineers
SO	Second Order (Average)
SQF	Software Quality Factor
SQM	Software Quality Metrics
SQPP	Software Quality Program Plan
SRS	Software Requirement Specification
SSS	System/Segment Specification
$\hat{T}$	Estimated Programming Time
V	Volume
V^*	Potential Volume
WSO	Weighted Second Order (Average)

☆U.S. GOVERNMENT PRINTING OFFICE: 1992 - 684-061/60053