[114] J.M. Silverman. Reflections on the verification of the security of an operating system kernel. In *Proc. 9th ACM Symposium on Operating Systems Principles*, pages 143–154, Bretton Woods, NH, October 1983. (ACM Operating Systems Review, Vol 17, No. 5).

[115] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, December 1988.

[116] R.E. Strom. Mechanisms for compile-time enforcement of security. In *Proceedings 10th Symposium on Principles of Programming Languages*, pages 276–284, Austin, TX, January 1983.

[117] F.W. von Henke and D.C. Luckham. A methodology for verifying programs. In *Proceedings, International Conference on Reliable Software*, pages 156–164, Los Angeles, CA, April 1975. IEEE Computer Society.

[118] Leslie J. Waguespack, Jr. and Sunil Badlani. Software complexity assessment: An introduction and annotated bibliography. *ACM Software Engineering Notes*, 12(4):52–71, October 1987.

[119] Elaine J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, April 1982.

[120] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[121] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

[122] C. Wilson and L.J. Osterweil. Omega—a data flow analysis tool for the C programming language. In *Proceedings COMPSAC*, pages 9–18, 1982.

[123] Jeannette M. Wing. A study of 12 specifications of the library problem. *IEEE Software*, 5(4):66–76, July 1988.

[102] K.A. Redish and W.F. Smyth. Evaluating measures of program quality. *Computer Journal*, 30(3):228–232, June 1987.

[103] G-C. Roman. A taxonomy of current issues in requirements engineering. *IEEE Computer*, 18(4):14–21, April 1985.

[104] W. W. Royce. Managing the development of large software systems. In *Proceedings WESCON*, August 1970.

[105] John Rushby. Quality measures and assurance for AI software. Technical Report SRI-CSL-88-7R, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1988. Also available as NASA Contractor Report 4187.

[106] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. Technical Report SRI-CSL-91-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1991. Also available as NASA Contractor Report 4384.

[107] John Rushby and Friedrich von Henke. Formal verification of the interactive convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1989 (Revised August 1991). Also available as NASA Contractor Report 4239.

[108] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *SIGSOFT '91: Software for Critical Systems*, New Orleans, LA, December 1991. (To appear.).

[109] John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.

[110] P.A. Scheffer, A.H. Stone III, and W.E. Rzepka. A case study of SREM. *IEEE Computer*, 18(4):47–54, April 1985.

[111] Richard W. Selby, Victor R. Basili, and F. Terry Baker. Cleanroom software development: An empirical evaluation. *IEEE Transactions on Software Engineering*, SE-13(9):1027–1037, September 1987.

[112] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, July 1988.

[113] Vincent Shen. Editor's introduction to "Quality Time" department. *IEEE Software*, 4(5):84, September 1987.

[89] Bertrand Meyer. On formalism in specifications. *IEEE Software*, 2(1):6–26, January 1985.

[90] R.A. De Millo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[91] Harlan D. Mills, Michael Dyer, and Richard Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987.

[92] M. Moriconi and D.F. Hare. The PegaSys system: Pictures as formal documentation of large programs. *ACM Transactions on Programming Languages and Systems*, 8(4):524–546, October 1986.

[93] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability—Measurement, Prediction, Application.* McGraw Hill, New York, NY, 1987.

[94] David R. Musser. Abstract data type specification in the AFFIRM system. *IEEE Transactions on Software Engineering*, SE-6(1):24–32, January 1980.

[95] G.J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760–768, September 1978.

[96] Peter Naur. Formalization in program development. *BIT*, 22:437–453, 1982.

[97] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, SE-14(6):868–874, June 1988.

[98] L.J. Osterweil and L.D. Fosdick. DAVE—a validation error detection and documentation system for Fortran programs. *Software—Practice and Experience*, 6:473–486, October–December 1976.

[99] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[100] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[101] K.A. Redish and W.F. Smyth. Program style analysis: A natural by-product of program compilation. *Communications of the ACM*, 29(2):126–133, February 1986.

[76] Richard A. Kemmerer. Testing formal specifications to determine design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[77] Richard A. Kemmerer. Verification assessment study final report. Technical Report C3-CR01-86, National Computer Security Center, Ft. Meade, MD, 1986. 5 Volumes. US distribution only.

[78] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[79] J.C. Knight and N.G. Leveson. An empirical study of failure probabilities in multi-version software. In *Digest of Papers, FTCS 16*, pages 165–170, Vienna, Austria, July 1986. IEEE Computer Society.

[80] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[81] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–293. Pergamon, New York, NY, 1970.

[82] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[83] Nancy G. Leveson. Software safety: Why, what and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[84] Nancy G. Leveson and John C. Knight. On N-Version programming. *ACM Software Engineering Notes*, 15(1):24–35, January 1990.

[85] N.G. Leveson. Software safety in computer controlled systems. *IEEE Computer*, 17(2):48–55, February 1984.

[86] N.G. Leveson and P.R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[87] Robert Mandl. Orthogonal Latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–01058, October 1985.

[88] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[62] William E. Howden. An evaluation of the effectiveness of symbolic testing. *Software—Practice and Experience*, 8:381–397, 1978.

[63] William E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, March 1980.

[64] William. E. Howden. Software validation techniques applied to scientific programs. *ACM Transactions on Programming Languages and Systems*, 2(3):307–320, July 1980.

[65] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[66] William E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, SE-12(10):997–1005, October 1986.

[67] D.C. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, February 1987.

[68] D.C. Ince and S. Hekmatpour. An empirical evaluation of random testing. *Computer Journal*, 29(4):380, August 1986.

[69] Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[70] T.C. Jones. *Programming Productivity.* McGraw Hill, New York, NY, 1986.

[71] D. Kapur and D.R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, February 1987.

[72] Deepak Kapur and Mandayam Srivas. Computability and implementability issues in abstract data types. *Science of Computer Programming*, 10:33–63, 1988.

[73] M. Karr and D.B. Lovemann III. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, May 1978.

[74] Joseph K. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. Software complexity measurement. *Communications of the ACM*, 29(11):1044–1050, November 1986.

[75] Chris F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, May 1987.

[49] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian K. Reid, editor, *Proceedings of 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[50] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, June 1988.

[51] S. German. Automating proofs of the absence of common runtime errors. In *Proceedings, 5th ACM Symposium on the Principles of Programming Languages*, pages 105–118, Tucson, AZ, January 1978.

[52] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[53] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[54] M.H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, NY, 1977.

[55] Warren Harrison and Curtis R. Cook. A note on the Berry-Meekings style metric. *Communications of the ACM*, 29(2):123–125, February 1986.

[56] Ian Hayes, editor. *Specification Case Studies*. Prentice-Hall International (UK) Ltd., Hemel Hempstead, UK, 1987.

[57] Ian J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.

[58] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, September 1981.

[59] S. Henry and D. Kafura. The evaluation of software systems' structure using quantitative software metrics. *Software—Practice and Experience*, 14(6):561–573, June 1984.

[60] Paul N. Hilfinger. An Ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, April 1988.

[61] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-4(1):70–73, January 1977.

[36] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings, Menlo Park, CA, 1986.

[37] P. Allen Currit, Michael Dyer, and Harlan D. Mills. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, SE-12(1):3–11, January 1986.

[38] C.G. Davis and C.R. Vick. The software development system. *IEEE Transactions on Software Engineering*, SE–3(1):69–84, January 1977.

[39] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[40] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, July 1977.

[41] *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[42] Thomas Downs. An approach to the modeling of software testing with some applications. *IEEE Transactions on Software Engineering*, SE-11(4), April 1985.

[43] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–443, April 1984.

[44] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P.J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[45] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, March 1976.

[46] Michael E. Fagan. Advances in software inspection. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[47] M. A. Fischler and O. Firschein. A fault-tolerant multiprocessor architecture for real-time control applications. In *First Annual Symposium in Computer Architecture*, pages 151–157, December 1973.

[48] M. A. Fischler, O. Firschein, and D.L. Drew. Distinct software: An approach to reliable computing. In *Second Annual USA-Japan Computer Conference*, pages 27–4–1–27–4–7, Tokyo, Japan, August 1975.

[24] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[25] T.A. Budd, R.A. De Millo, R.J. Lipton, and F.G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings, 7th ACM Symposium on the Principles of Programming Languages*, Las Vegas, NV, January 1980.

[26] J.R. Burch, E.M. Clarke, K.L McMillan, D.L. DIll, and L.J. Hwang. Symbolic model checking 2^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[27] J. Celko, J.S. Davis, and J. Mitchell. A demonstration of three requirements language systems. *SIGPLAN Notices*, 18(1):9–14, January 1983.

[28] S. Cha, N.G. Leveson, T.J. Shimeall, and J.C. Knight. An empirical study of software error detection using self-checks. In *Digest of Papers, FTCS 17*, pages 156–161, Pittsburgh, PA., July 1987. IEEE Computer Society.

[29] Fun Ting Chan and Tsong Hueh Chen. AIDA—a dynamic data flow anomaly detection system for Pascal programs. *Software—Practice and Experience*, 17(3):227–239, March 1987.

[30] M. Cheheyl et al. Verifying security. *Computing Surveys*, 13(3):279–339, September 1981.

[31] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Digest of Papers, FTCS 8*, pages 3–9, Toulouse, France, June 1978.

[32] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[33] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[34] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, NY, 1981.

[35] Robert F. Cmelik and Narain H. Gehani. Dimensional analysis with C++. *IEEE Software*, 5(3):21–27, May 1988.

[11] T.E. Bell, D.C. Bixler, and M.E. Dyer. An extendable approach to computer–aided software requirements engineering. *IEEE Transactions on Software Engineering*, SE–3(1):49–59, January 1977.

[12] Jean-François Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[13] Gerald M. Berns. Assessing software maintainability. *Communications of the ACM*, 27(1):14–23, January 1984.

[14] R.E. Berry and B.A.E. Meekings. A style analysis of C programs. *Communications of the ACM*, 28(1):80–88, January 1985.

[15] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[16] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[17] Barry W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, January 1984.

[18] Barry W. Boehm. Verifying and validating software requirements. *IEEE Software*, 1(1):75–88, January 1984.

[19] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[20] Barry W. Boehm, Terence E. Gray, and Thomas Seewaldt. Prototyping versus specifying: A multiproject experiment. *IEEE Transactions on Software Engineering*, SE-10(3):290–303, May 1984.

[21] R.S. Boyer, B. Elspas, and K.N Levitt. SELECT: A formal system for testing and debugging programs by symbolic execution. In *Proceedings, International Conference on Reliable Software*, pages 234–245, Los Angeles, CA, April 1975. IEEE Computer Society.

[22] Susan .S. Brilliant and John C. Knight ans Nancy G. Leveson. The consistent comparison problem in N-Version software. *IEEE Transactions on Software Engineering*, 15(11):1481–1485, November 1989.

[23] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

Bibliography

[1] Maryam Alavi. An assessment of the prototyping approach to information systems development. *Communications of the ACM*, 27(6):556–563, June 1984.

[2] M.W. Alford. A requirements engineering methodology for real–time processing requirements. *IEEE Transactions on Software Engineering*, SE–3(1):60–69, January 1977.

[3] M.W. Alford. SREM at the age of eight; the distributed computing design system. *IEEE Computer*, 18(4):36–46, April 1985.

[4] T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding. An evaluation of software fault tolerance in a practical system. In *Digest of Papers, FTCS 15*, pages 140–145, Ann Arbor, MI, June 1985. IEEE Computer Society.

[5] T. Anderson and P.A. Lee. *Fault-Tolerance: Principles and Practice (Second, revised edition)*. Springer Verlag, Wien and New York, 1990.

[6] T. Anderson and R.W. Witty. Safe programming. *BIT*, 18:1–8, 1978.

[7] Dorothy M. Andrews and Jeoffrey P. Benson. An automated program testing methodology and its implementation. In *Proceedings, 5th International Conference on Software Engineering*, pages 254–261, San Diego, CA, March 1981.

[8] Geoff Baldwin. Implementation of physical units. *SIGPLAN Notices*, 22(8):45–50, August 1987.

[9] V.R. Basili and B.T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[10] Farokh B. Bastani and S. Sitharama Iyengar. The effect of data structures on the logical complexity of programs. *Communications of the ACM*, 30(3):250–259, March 1987.

specifications early in the life-cycle deserve special attention: rapid prototyping and some of the techniques of formal verification may be especially useful here. For really critical requirements, formal verification of those properties should be considered; conventional testing can be used to ensure that the less-critical requirements are satisfied.

> "The goal of practical usefulness does not imply that the verification of a program must be made independent of creative effort on the part of the programmer ...such a requirement is utterly unrealistic."

In reality, a verification system assists the human *user* to develop a convincing argument for his program by acting as an implacably skeptical colleague who demands that all assumptions be stated and all claims justified. The requirement to explicate and formalize what would otherwise be unexamined assumptions is especially valuable. Speaking from substantial experience, Shankar [112] observes:

> "The utility of proof-checkers is in clarifying proofs rather than in validating assertions. The commonly held view of proof-checkers is that they do more of the latter than the former. In fact, very little of the time spent with a proof-checker is actually spent proving theorems. Much of it goes into finding counterexamples, correcting mistakes, and refining arguments, definitions, or statements of theorems. A useful automatic proof-checker plays the role of a devil's advocate for this purpose."

Our own experience supports this view. We have recently undertaken the formal verification of an important and well-known algorithm for clock-synchronization and have discovered that the published journal proof [82] contains major flaws [108, 107]. It was the relentless skepticism of our formal verification environment that led us to this discovery, but our belief in the correctness of our current proof owes as much to the increased understanding of the problem that we obtained through arguing with the theorem prover as it does to the fact that the theorem prover now accepts our proofs.

Another element in De Millo, Lipton and Perlis' parody that is far from the truth is their implicit assumption that formal verification is something that is done *after* the program has been developed. In reality, formal verification is practised as a component of a development methodology: the verification and the program (or specification) are developed together, each generating new problems, solutions, and insights that contribute to the further development of the other. Formal verification can be made easier if the property being verified is achieved by direct and simple means. Thus, in addition to helping the user build a convincing case for belief in his program, formal verification encourages the programmer to build a more believable, and often better, program in the first place.

Overall, the conclusion to be drawn from experimental and other data seems to be that all testing methods have their merits, and these tend to be complementary to each other. For the purpose of enhancing reliability, random testing is a clear winner. For the purpose of finding bugs, anomaly detection and walk-throughs should be used in combination with functional and structural testing (Howden [66] describes such an integrated approach). Techniques for evaluating requirements and other

technique depends on the extent to which errors are correlated between the two versions. Recently, experiments have been performed to examine this hypothesis in the context of N-Version programming[6] [79, 80, 44, 84]; the results indicate that errors do tend to be correlated to some extent. Additional problems can arise with numerical software, due to the character of finite-precision arithmetic [22].

The efficiency of the debugging process can be evaluated by seeding a program with known errors—this is called mutation testing. If ten errors are seeded, and debugging reveals 40 bugs, including 8 of those that were seeded, we may conclude that 10 ($= \frac{40}{8}(10-8)$) bugs (including two seeded ones) remain. The weakness in this approach is that it is highly questionable whether the seeded bugs reflect the characteristics of the natural bug population: being introduced by simple modification to the program code, they are unlikely to reflect the behavior of subtle errors committed earlier in the life-cycle. A counter-argument is that test sets that detect simple errors will often catch subtle errors too (this is called "coupling" [90]) and mutation testing provides a systematic technique for establishing the coverage of a test set [25, 65].

For some very critical applications, formal verification is recommended as the assurance mechanism of choice [41]. Some authors have cast doubt on the value of this approach [39, 96], arguing that use of truly formal specification and verification tends to overload the user with intricate detail, and that the soundness of the whole process rests on the correctness of the underlying verification environment. Since this will typically be a very large and complex program in its own right, located at the limits of the state of the art, its own correctness should be regarded with more than usual skepticism. The thrust of this argument is that formal verification moves responsibility away from the "social process" that involves human scrutiny, towards a mechanical process with little human participation. We believe this concern unfounded and based on a mistaken view of how a mechanical verification environment is used. De Millo, Lipton and Perlis [39] claim that:

> "The scenario envisaged by the proponents of verification goes something like this: the programmer inserts his 300-line input/output package into the verifier. Several hours later, he returns. There is his 20,000-line verification and the message 'VERIFIED'."

This is a parody of the scenario actually envisaged by the proponents of verification. In a paper published several years earlier [117], von Henke and Luckham indicated the true nature of this scenario when they wrote:

[6] N-version programming is an adaptation for software of the modular redundancy techniques used to provide reliable and fault tolerant hardware. N (typically $N = 3$) separate modules perform the computation and their results are submitted to a majority voter. Since all software faults are *design* faults, the redundant software modules must be separately designed and programmed. N-Version programming is advocated by Avizienis [31] (see also [47, 48] for a slightly different perspective), but has been criticized by Leveson and others (see the discussion in [5]).

discuss this issue and identify a historical evolution of concerns, starting with debugging in the 1950s. They propose that the concern for 1988 and beyond should be the use of testing as a fault prevention mechanism, based on the detection of faults at the earliest possible stage in the life-cycle.

Of the systematic testing strategies, Howden's data [64] ([62] is similar, but based on a smaller sample of programs) provides evidence that functional testing finds about twice as many bugs as structural testing; furthermore, several of the bugs found by structural testing would be found more easily or more reliably by other methods, so that the ratio three to one probably more accurately reflects the superiority of functional over structural testing.

Howden's data also shows that static testing (particularly automated anomaly detection) found more bugs than dynamic testing—however, the two methods were complementary, dynamic testing tending to find the bugs that static testing missed, and vice versa. Myers [95] presented similar data, showing that code walk-throughs were about as effective as dynamic testing at locating errors in PL/1 programs.

The efficacy of anomaly detection is likely to increase with the degree of redundancy present in the programming language—indeed, the presence of redundancy may often allow errors to be detected at compile-time that would otherwise only be located at run-time. (Compare Lisp with Ada, for example: almost any string with balanced parentheses is a syntactically valid Lisp program and any deficiencies will only be discovered in dynamic testing, whereas writing a syntactically correct Ada program is quite demanding and many errors will be caught at the compilation stage.) There have been proposals to increase the degree of redundancy in programs in order to improve early error-detection. One of the most common suggestions is to allow physical units to be attached to variables and constants [73, 8]. Expressions may then be subject to dimensional analysis in order to prevent a variable representing length being added to one representing time, or assigned to one representing velocity.[5] Other proposals would render it impossible to read from an empty message buffer, or to use a variable that has not yet been given a value [116].

All forms of dynamic testing assume that it is possible to detect errors when they occur. This may not always be the case—as, for example, when the "correct" value of a result may be unknown. Weyuker [119] first identified the problem of "untestable" programs, and proposed two methods for alleviating the difficulty they pose. The first is to use simplified test cases for which it is possible to compute the "correct" answers. The second, and more interesting method suggested by Weyuker is "dual coding": writing a second—probably simpler but less efficient—version of the program to serve as a check on the "real" one. The efficacy of this

[5] Strong typing, present in any modern programming language, provides some protection of this sort—preventing booleans being added to integers, for example—but the use of physical units and dimensional analysis represents a capability beyond the normal typing rules. The data abstraction facilities of a modern language such as C++ or Ada can provide this capability, however [35, 60].

respond gracefully to errors. The purpose of a rapid prototype is to allow early experience with, and direct testing of, the main aspects of the system's proposed functionality—thereby allowing much earlier and more realistic appraisals of the system's requirements specifications.

An experimental comparison of a prototyping versus the conventional approach to software development [20] found that both approaches yielded approximately equivalent products, though the prototyping approach required much less effort (45% less) and generated less code (chiefly due to the elimination of marginal requirements). The products developed incrementally were easier to learn and use, but the conventionally developed products had more coherent designs and were easier to integrate. (Another experimental evaluation of prototyping is described by Alavi [1].)

Viewed as testing vehicles for evaluating and refining requirements specifications, rapid prototypes fit neatly into the standard life-cycle model of software engineering. A more radical approach that has much in common with rapid prototyping is incremental software development. Here, the complete software system is made to run quite early in the development phase, even if it does nothing useful except call dummy subprograms. Then it is fleshed out, with the subprograms being developed in their turn to call dummy routines at the next level down, and so on until the system is complete. The advantages claimed for this approach [23] are that it necessitates top-down design, allows easy backtracking to reconsider inappropriate decisions, lends itself to rapid prototyping, and has a beneficial impact on morale. See also Boehm's "Spiral" Model of system development [19].

5.4 Discussion of Testing

Much attention has been focused on systematic testing strategies—especially structurally based ones. However, there is evidence that, if increasing reliability (rather than finding the maximum number of bugs) is the goal, then random testing is much superior to other methods. Currit, Dyer and Mills [37] report data from major IBM systems which shows that random testing would be 30 times more effective than structural testing in improving the reliability of these systems. The reason for this is the enormous variation in the rate at which different bugs lead to failure: one third of all the bugs had a MTTF of over 5000 years (and thus have no effect on overall MTTF), and a mere 2% of the bugs accounted for 1000 times more failures than the 60% of bugs that were encountered least often.[4]

Interpretation of data such as these requires a context that establishes the *purpose* of testing—is it to find bugs or to improve reliability? Gelperin and Hetzel [50]

[4]The thirty-fold improvement of random over structural testing is simply estimated by the calculation $2 \times 1000/60$.

5.3.5 Testing of Specifications

Conventional formal specification languages are optimized for ease and clarity of expression and are not directly executable. Furthermore, high-level specifications are often deliberately partial—they indicate what is required of any implementation, but do not provide enough information to uniquely characterize an acceptable implementation. Nonetheless, it is highly desirable to subject such specifications to tests and scrutiny in order to determine whether they accurately capture their intended meaning.

If direct execution is infeasible for the specification technique chosen, indirect testing methods must be used. As noted above, a formal specification defines properties that are required to be true of any implementation. In addition to the properties S that have been specified in this way, there may be additional properties A that are desired but not mandated, or that are believed to be subsumed by S, or that are to be added in a later, more detailed, specification. Tests of formal specifications consist of attempts to check whether these intended relationships between the given S and various sets of properties A do, in fact, hold. Thus, to ensure whether the property A is subsumed by S, we may try to establish the putative theorem $S \supset A$. Independently of additional properties A, we may wish to ensure that the specification S is consistent (i.e., has a model)—since otherwise $S \supset A$ is a theorem for all A.

Depending on the formal specification language and verification environment available, examinations such as those described above may be conducted by attempting to prove putative theorems, by symbolic evaluation, or by rapid prototyping. Kemmerer [76] describes the latter two alternatives. An important special case is the checking of specifications for consistency with a notion of "multilevel security." This activity, which is a requirement for certain types of system [41], seeks to demonstrate that a fairly concrete specification of a system is consistent with an abstract specification of security [30].

5.3.6 Rapid Prototyping

As we have noted several times, errors made early but detected late in the life-cycle are particularly costly and serious. This applies especially to missed or inappropriate requirements—yet such faults of omission are especially difficult to detect at an early stage. Systematic review will often detect inconsistent, or ambiguous requirements, but missing requirements generate no internal inconsistencies and often escape detection until the system is actually built and tried in practice.

A rapid prototype is one that simulates the important interfaces and performs the main functions of the intended system, while not necessarily being bound by the same performance constraints. Prototypes typically perform only the mainline tasks of the application, but make no attempt to handle the exceptional cases, or

of detail, should be the focus of mathematical verification. Indeed, a whole hierarchy of specifications may be verified in stepwise fashion from a highly abstract, but intuitively understandable one, down to a very detailed one that can be used as the basis for coding. If the original, abstract specification, is studied and understood by the user, and agreed by him to represent his requirements, then such hierarchical verification (provided it is performed without error) accomplishes the validation of the detailed specification. This approach is attractive in circumstances where the properties of interest are difficult to validate directly—as in the case of ultra-high reliability (where failure probabilities on the order of 10^{-9} per day may be required, but are unmeasurable in practice) [106], and security (which requires that *no possible* attack should be able to defeat the protection mechanisms) [114].

5.3.4 Executable Specifications

Specification languages provide mechanisms for saying *what* must be accomplished, not *how* to accomplish it. As a result, specifications cannot usually be executed. Programming languages on the other hand, reverse these concerns and provide many mechanisms for stipulating *how* something is to be accomplished. As a result, programs generally execute very efficiently, but are inperspicuous. Recently, however, *logic* programming languages have emerged that blur the distinction between specification and programming languages. By employing a more powerful interpreter (essentially a theorem prover, though generally referred to as an "inference engine"), logic programming languages allow the programmer to concentrate more on the *what*, and less on the *how* of his program. Dually, these languages can be regarded as *executable* specification languages. The obvious merit of executable specification languages is that they permit specifications to be tested and validated directly in execution.

Prolog, the best known logic programming language [34], contains many compromises intended to increase its efficiency in execution that detract from its merit as a specification language.[3] Languages based on equations, however, offer considerable promise. OBJ [49], developed by Goguen and his coworkers in the Computer Science Laboratory of SRI, is the best developed and most widely known of these. In addition to a cleaner logical foundation than Prolog, OBJ has sophisticated typing and parameterization features that contribute to the clarity and power of its specifications. (Being based on equational logic, OBJ could also exploit the completeness and consistency checks described in Section 5.3.2 (page 46), although the present version of the system does not do so.)

[3] Stickel's Prolog Technology Theorem Prover (PTTP) [115], which provides correct first-order semantics—but with Prolog-like efficiency when restricted to Horn clauses—overcomes some of these disadvantages.

In addition to the *analysis* and *extraction* tools described above, *generation and display* tools provide two-dimensional graphical displays for R-Nets, and functional and analytical *simulators* support validation of performance, accuracy, and functional requirements. Both types of simulator automatically generate a PASCAL program corresponding to the R-Net structure. Each ALPHA becomes a call to a PASCAL procedure—which is generally written by the requirements engineer and associated with the corresponding ALPHA as an attribute.

The REVS simulation tools attempt to generate input data from the description of the data provided by the user. The user can also declare *artificial* data, i.e., data not required to be generated by the system when deployed. Typically, artificial data will be more abstract than the data actually applied to the system in operation. In addition to functionality, REVS supports simulation for the validation of performance and accuracy constraints. The latter are evaluated using "rapid prototypes" of the critical algorithms to be used in practice.

5.3.2 Completeness and Consistency of Specifications

Among the properties of specifications that are generally considered desirable, completeness and consistency rank highly. Informally, completeness means that the specification gives enough information to totally determine the properties of the object being specified; consistency means that it does not specify two contradictory properties.

For a semi-formal specification language, it may be possible to give some precepts for the construction of complete and consistent specifications, and it may be feasible to check adherence to these precepts mechanically. With formal specification languages, however, rather more may be possible. For quantifier-free equational logic—which logic has been found very suitable for the specification of abstract data types [94]—there is a formal notion of "sufficient completeness" that can be checked mechanically [53], and a sufficient test for consistency is that the Knuth-Bendix algorithm [81] should terminate without adding the rule `true -> false` [71]. Kapur and Srivas [72] discuss other important properties of such specifications and describe appropriate tests. Meyer [89] provides some interesting examples of flawed specifications that have appeared in the literature, while Wing [123] describes 12 different specifications for a single problem and discusses some of the incompletenesses and ambiguities found therein.

5.3.3 Mathematical Verification of Specifications

As we explained in Section 5.2.3 (page 35), mathematical verification demonstrates consistency between two different descriptions of a program. Often, one of these descriptions is the program itself—so that a program is verified against its specification. However, it is perfectly feasible that two specifications, at different levels

meaning. Only dataflow concepts have meaning. Thus, in the conventional sense, RSL is not an extensible language.

Using RSL, a user (called the *requirements engineer*) is encouraged to identify significant units of processing, each of which is viewed as a single input/single output ALPHA (inputs and outputs can have structure—rather like PASCAL records—so the restriction to single inputs and outputs is not as severe as it might appear). The definition of an ALPHA consists of declarations of its input and output, declarations of any *files* the ALPHA will write to and a brief natural language description of the transformations it effects on the input data.

ALPHAs are connected together with OR, AND, SELECT and FOR_EACH nodes into what can be viewed as a dataflow graph.[2] The intention is to express system requirements in terms of how significant units of processing are connected with each other, and the kind of data that flows along the interconnections.

OR nodes resemble a PASCAL *case* statement and indicate conditions under which each output path will be followed. The conditions attached to a path through an R-Net identify the conditions under which an ALPHA is invoked and in which an output message must be produced in response to an input message.

The AND node indicates that all of the paths following it are to be executed. Execution of the paths can be in any order or with any degree of parallelism permitted by the hardware base. It is intended that any file read in any of the parallel paths is not written by any other path, otherwise the behavior of the system would be indeterminate. In RSL, this constraint on files is called the *independence* property.

There is no *goto* statement in RSL. To produce other than loopless programs, RSL provides the FOR_EACH node. This takes a set of data items as argument and indicates that the path following it is to be executed once for each of the items in the set; the order is not specified.

Multiple R-Nets can be linked together using EVENT nodes, and VALIDATION_POINTs can be attached to paths in an R-Net in order to specify performance and accuracy requirements.

Several tools have been constructed to support requirements descriptions written in RSL. Collectively, these constitute the "Requirements Engineering Validation System" (REVS). The most basic component of REVS is the RSL translator, which analyzes RSL requirements definitions and generates entries in a central database called the Abstract System Semantic Model (ASSM). Information in the ASSM may be queried, and checked for consistency using the "Requirements Analysis and Data Extraction" system (RADX). RADX generates reports that trace source document requirements to RSL definitions, identify data items with no source or sink, unspecified attributes, useless entities and so on. Other checks ensure, among other things, that the paths following an AND satisfy the independence property.

[2]Strictly, it is incorrect to refer to RSL descriptions as dataflow programs since they can produce side-effects to files.

Again due to its origins in ballistic missile defense, SREM is very much concerned with the constraints of accuracy and performance. It therefore makes provision for traceable, testable, performance and accuracy constraints to be attached to requirements specifications.

In addition to support for analyzing the problem and checking understanding, SREM's tools perform "internal" completeness and consistency checks (i.e., checks that are performed relative to the requirements definition itself, without reference to external reality). These checks ensure, for example, that all data has a source, and that there are no dangling items still"to be done."

The paradigm underlying SREM is that system requirements describe the necessary processing in terms of all possible responses (and the conditions for each type of response) to each input message across each interface. This paradigm is based on a graph model of computation: requirements are specified as *Requirement Networks*, or *R-Nets*, of processing steps. Each R-Net is a tree of *paths* processing a given type of stimulus.

R-nets are expressed in the Requirements Statement Language (RSL), the language of SREM: an RSL requirements definition is a linear representation of a two-dimensional R-net. Requirements definitions in RSL are composed of four types of primitives:

- *Elements* in RSL include the types of data necessary in the system (DATA), the objects manipulated by the system being described (MESSAGES), the functional processing steps (ALPHAs), and the processing flow descriptions themselves.

- *Relationships* are mathematical relations between elements. For example, the relationship of DATA being INPUT to an ALPHA. Generally, a complementary relationship is defined for each basic relationship: for example, an ALPHA INPUTS DATA.

- *Attributes* are properties of objects, such as the ACCURACY and INITIAL_VALUE attributes of elements of type DATA. A set of values (names, numbers, or text strings) may be associated with each attribute. For example, the set of values associated with INITIAL_VALUE is the set of initial values allowed for data items.

- *Structures* model the flows through the processing steps (ALPHAs) or the flows between places where accuracy or timing requirements are stated (VALIDATION_POINTs).

RSL is described as an extensible language. What this means is that the user can declare new elements, relationships, and attributes; *however, they do not have*

important component of this is *traceability*—items in the specification should have clear antecedents in earlier specifications or statements of system objectives.

Feasibility : A specification is feasible to the extent that the life-cycle benefits of the system specified exceed its life-cycle costs. Thus feasibility involves more than verifying that a system satisfies functional and performance requirements. It also implies validating that the specified system will be sufficiently maintainable, reliable, and human-engineered to keep a positive life-cycle balance sheet.

Testability : A specification is testable to the extent that one can identify an economically feasible technique for determining whether or not the developed software will satisfy the specification.

Among the methodologies that aim to satisfy these criteria TRW's SREM [2, 3, 11, 38] is representative, and is described in the following section.

5.3.1.1 SREM

SREM (Software Requirements Engineering Methodology) was the product of a program undertaken by TRW Defense and Space Systems Group as part of a larger program sponsored by the Ballistic Missile Defense Advanced Technology Center (BMDATC) in order to improve the techniques for developing correct, reliable BMD software. Early descriptions of SREM include [2, 11]; descriptions of subsequent extensions for distributed systems can be found in [3], and accounts of experience using SREM are given in [18, 27, 110].

Owing to its genesis in the problems of software for ballistic missile defense, SREM adopts a system paradigm derived from real-time control systems. Such systems are considered as "stimulus-response" networks: an "input message" is placed on an "input interface" and the results of processing—the "output message" and the contents of memory—are extracted from an "output interface" [2]. Furthermore, the requirements for the system are understood in terms of the processing steps necessary to undertake the required task. The essence of a SREM requirements definition is therefore a dataflow-like description (called an *R-net*) of the processing steps to be performed and the flow of data (messages) between them.

SREM recognizes that requirements engineering is concerned with more than just writing a description of what is required—it is first necessary to analyze the problem in order to discover just what *is* required, and it is constantly necessary to check one's understanding of the problem and its evolving description against external reality. Accordingly, SREM allows behavioral simulation studies in order to verify that the system's interfaces and processing relationships behave as required. In addition, there is provision for *traceability* of all decisions back to source.

- The documentation is likely to be inadequate, exacerbating the problem of determining why certain decisions were made and assessing the impact of decisions. It is also difficult to keep the documentation current as changes are made.

Not only are errors in requirements and specifications expensive to correct, they are also among the most frequent of all errors – one study [9] found that 30% of all errors could be attributed to faulty statement or understanding of requirements and specifications. Worse, it appears that errors made in these early stages are among those most likely to lead to *catastrophic failures* [83].

These problems indicate the need for methodologies, languages, and tools that address the earliest stages in the software life-cycle. The aims of such *requirements engineering* are to see that the *right* system is built and that it is built *correctly*. Since systems have many dimensions, there are several facets to the question of what constitutes the right system. Roman [103] divides these facets of system requirements into two main categories: *functional* and *non-functional* (these latter are also called *constraints*). Functional requirements capture the nature of the interaction between the system and its environment—they specify what the system is to *do*. Non-functional requirements restrict the types of system solutions that should be considered. Examples of non-functional requirements include security, performance, operating constraints, and cost.

Functional requirements can be expressed in two very different ways. The *declarative* (or *non-constructive*) approach seeks to describe *what* the system must do without any indication of *how* it is to do it. This style of requirement specification imposes little structure on the system and leaves maximum freedom to the system designer—but, since it says nothing about how the system is to work, it provides little basis for checking non-functional constraints. The *procedural* approach to the specification of functional requirements, on the other hand, aims to describe what the system must do in terms of an outline design for accomplishing it. This approach appeals to many engineers who find it most natural to think of a requirement in terms of a mechanism for accomplishing it.

Both functional and non-functional requirements definitions, and declarative and procedural specifications, should satisfy certain criteria. Boehm [18] identifies four such criteria: namely, completeness, consistency, feasibility, and testability.

Completeness : A specification is complete to the extent that all of its parts are present and each part is fully developed. Specifically, this means: no TBDs ("To Be Done"), no nonexistent references, no missing specification items, and no missing functions.

Consistency : A specification is consistent to the extent that its provisions do not conflict with each other or with governing specifications and objectives. An

5.3 Testing Requirements and Specifications

So far we have explicitly considered only the testing of finished *programs*, but there is much to be said for the testing of specifications and requirements also. In the first place, testing a program against its specification is of little value if the specification is wrong; secondly, the cost of repairing faults increases dramatically as the number of life-cycle stages between its commission and its detection increase. As we noted in Section 2 (page 4), it is relatively simple, quick, and cheap, to correct an error in a requirements statement if that error is discovered during review of that statement, and before any further stages have begun; and it is also fairly simple, quick, and cheap to correct a coding error during testing. It is, however, unlikely to be either simple, quick, or cheap to correct an error in requirements that is only discovered during system test. Major redesign may be required, and wholesale changes necessitated. Any attempt to correct the problem by a "quick fix" is likely to generate even more problems in the long run.

For these reasons, testing and evaluation of requirements, specifications, and design documents may be considered a very wise investment. Of the testing methods we have described, only structured walk-throughs are likely to be feasible if the requirements and specification documents are informal, natural-language texts. If requirements and specifications are presented in some semi-formal design language, then limited anomaly detection and mathematical verification may be feasible, and possibly simulated execution also. If fully formal requirements and/or specifications are available, then quite strong forms of anomaly detection, mathematical verification, and even dynamic testing may be feasible.

In the following sections, we will briefly touch on some of these topics.

5.3.1 Requirements Engineering and Evaluation

Many studies of the software life-cycle have concluded that its early stages are particularly crucial. It is in these early stages that the overall requirements for a system are identified and the basic design of the system is specified. Errors or misapprehensions made at these stages can prove ruinously expensive to correct later on. Recent studies (e.g., [9, 18]) have shown that errors due to faulty requirements are between 10 and 100 times more expensive to fix at the implementation stage than at the requirements stage. There are two main reasons for the high cost of modifying early decisions late in the life-cycle:

- The changes often have a widespread impact on the system, requiring many lines of code to be modified. Furthermore, it can be difficult to identify all of the code requiring attention, resulting in the modification being performed incorrectly.

hazards as possible, but does imply that additional procedures may be necessary to ensure system safety.

The goal of SFTA is to show that the logic contained in the software design will not produce system safety failures, and to determine environmental conditions which could lead to the software causing a safety failure. The basic procedure is to suppose that the software has caused a condition which the hazard analysis has determined will lead to catastrophe, and then to work backward to determine the set of possible causes for the condition to occur.

The root of the fault tree is the event to be analyzed, i.e., the "loss event." Necessary preconditions are described at the next level of the tree with either an AND or an OR relationship. Each subnode is expanded in a similar fashion until all leaves describe events of calculable probability or are incapable of further analysis for some reason. SFTA builds software fault trees using a subset of the symbols currently in use for hardware systems. Thus hardware and software fault trees can be linked together at their interfaces to allow the entire system to be analyzed. This is extremely important since software safety procedures cannot be developed in a vacuum but must be considered as part of overall system safety. For example, a particular software error may cause a mishap only if there is a simultaneous human and/or hardware failure. Alternatively, environmental failure may cause the software error to manifest itself. In many previous safety mishaps, e.g., the nuclear power plant failure at Three Mile Island, the safety failure was actually the result of a sequence of interrelated failures in different parts of the system.

Fault tree analysis can be used at various levels and stages of software development. At the lowest level the code may be analyzed, but it should be noted that higher levels of analysis are important and can and will be interspersed with the code level. Thus the analysis can proceed and be viewed at various levels of abstraction. It is also possible to build fault trees from a program design language (PDL) and to thus use the information derived from the trees early in the software life cycle. When working at the code level, the starting place for the analysis is the code responsible for the output. The analysis then proceeds backward deducing both how the program got to this part of the code and determining the current values of the variable (current state).

An experimental application of SFTA to the flight and telemetry control system of a spacecraft is described by Leveson and Harvey [86]. They report that the analysis of a program consisting of over 1250 lines of Intel 8080 assembly code took two days and discovered a failure scenario that could have resulted in the destruction of the spacecraft. Conventional testing performed by an independent group prior to SFTA had failed to discover the problem revealed by SFTA.

respects the structural properties given in its specification. One advantage of this style of verification is that it can be performed with complete formality, but without burdening the user with details. The PegaSys system developed in the Computer Science Laboratory of SRI [92] supports the use of *pictures* as *formal* specifications of system structure, and hides all the details of theorem proving from the user.

5.2.4 Fault-Tree Analysis

Reliability is not the same as safety, nor is a reliable system necessarily a safe one. Reliability is concerned with the incidence of *failures*; safety concerns the occurrence of accidents or *mishaps*—which are defined as unplanned events that result in death, injury, illness, damage to or loss of property, or environmental harm. Whereas system failures are defined in terms of system services, safety is defined in terms of external consequences. If the required system services are specified incorrectly, then a system may be unsafe, though perfectly reliable. Conversely, it is feasible for a system to be safe, but unreliable. Enhancing the reliability of software, though desirable and perhaps necessary, is not sufficient for achieving *safe* software.

Leveson [85, 86, 83] has discussed the issue of software safety at length and proposed that some of the techniques of system safety engineering should be adapted and applied to software. First, it is necessary to define some of the terms used in system safety engineering. *Damage* is a measure of the loss in a mishap. A *hazard* is a condition with the potential for causing a mishap; the *severity* of a hazard is an assessment of the worst possible damage that could result, while the *danger* is the probability of the hazard leading to a mishap. *Risk* is the combination of hazard severity and danger. *Software Safety* is concerned with ensuring that software will execute in a system context without resulting in unacceptable risk. One class of techniques for software safety is concerned with design principles that will reduce the likelihood of hazardous states; another is concerned with methods for analyzing software in order to identify any unduly hazardous states. An example of the latter is "Software Fault Tree Analysis" (SFTA). The description below is adapted from that in [86].

SFTA is an adaptation to software of a technique that was developed and first applied in the late 60's in order to minimize the risk of inadvertent launch of a Minuteman missile. The first step, as in any safety analysis, is a hazard analysis of the entire system. This is essentially a listing and categorization of the hazards posed by the system. The classifications range from "catastrophic," meaning that the hazard poses extremely serious consequences, down to "negligible" which denotes that the hazard will not seriously affect system performance. Once the hazards have been determined, fault tree analysis proceeds. It should be noted here that in a complex system, it is possible, and perhaps even likely, that not all hazards can be predetermined. This fact does not decrease the necessity of identifying as many

were hired to add self-checks to programs containing a total 60 known faults (there were 8 different programs, each was given to three students). Only 9 out the 24 self-checking programs detected any faults at all; those that did find faults found only 6 of the 60 known faults, but they also discovered 6 previously unknown faults (in programs which had already been subjected to one million test-cases). Sadly, 22 new faults were introduced into the programs in the process of adding the self-checks.

5.2.3.2 Verification of Limited Properties

A common and familiar example of executable assertions is the "range check" generally compiled into array subscripting operations. Though a valuable safety net, these checks can be very expensive when they appear in the inner loops of a program. An approach which "turns the tables" on the relationship between formal verification and executable assertions is to *prove* that subscripting errors, and other simple forms of run-time error, cannot occur [51]. This is an example of an important variation on the application of formal verification.

Conventionally, the goal of formal verification is understood to be "proof of correctness." We have been careful to make a more careful and accurate statement—namely, that it provides a demonstration of consistency between a program and its specification—but we have implicitly assumed that the specification concerned is one that provides a full description of the functionality required of the program. This need not be the case, however. The methods of formal verification can be applied equally well when the specification is a limited, weak, or partial one: instead of proving that the program does "everything right," we can attempt to prove only that it does *certain* things right, or even that it is does *not* do certain things wrong. In fact, the properties proved of a program need not be functional properties at all, but can be higher order (e.g., "security"), or structural.

The limited properties to be proved may be chosen because of their tractability—i.e., because formal verification is among the most cost-effective ways of ensuring those properties—or because of their importance. The absence of array subscripting errors is an example of the first class; security exemplifies the second. In particular, security is an example of a "critical property": a property considered so important that really compelling evidence must attest to its realization. What constitutes a critical property is something that can only be determined by the customer (and the law!), but it will generally include anything that could place human life, national security, or major economic assets at risk.

Yet another variation on formal verification is to prove properties about the structural properties of programs. For example, a specification may assert that the program should have a certain structure, or that a certain structural relationship should exist among some of its components (e.g., one may use the other, but not vice-versa). Formal verification of these properties guarantees that the program

stead of concentrating on *what* he wants to say, he has to spend all his effort on *how* to say it—with the consequent danger that he may fail to say it correctly, and so fall into the second kind of error. Similarly, the user may have to spend considerable ingenuity, not in proving his theorems directly, but in persuading a mechanical theorem prover to prove them for him. This problem is compounded by the fact that most verification systems do not allow the user to reason about programs directly (as he would do if performing the proof by hand), but reduce the question of consistency to a (generally large) number of (generally lengthy) formulas called "verification conditions" that rob the user of much of his intuition concerning the program's behavior. This latter difficulty should not affect the reliability of the process (provided the theorem prover is sound), but will adversely affect its economics. SRI's EHDM system attempts to overcome these difficulties by providing a very expressive specification language and powerful underlying logic (based on multi-sorted higher order logic) together with the ability to reason about programs directly (using the Hoare, or relational, calculus) [109].

5.2.3.1 Executable Assertions

The task of proving consistency between a program and its specification is generally broken down into more manageable steps by embedding assertions at suitable points in the program text and proving that the assertions will always be satisfied when the locus of control passes these points during execution. An interesting alternative to *proving* the assertions à priori is to *test* them during execution and to halt the program with an error message if any assertion fails. This can permit a fairly simple proof of consistency between a program and the weakened specification "X or fail," where "X" was the original specification. Variations on this theme include the use of executable assertions during conventional dynamic testing [7], and in a dynamic variant of anomaly detection [29]. In the former case, the presence of the assertions allows the testing process to probe the "inner workings" of the program, rather than merely its input-output behavior (recall our discussion of symbolic execution in Section 5.1.4 on page 31). In the latter case, instrumenting the program with executable assertions allows data flow anomaly detection to be performed without requiring a data flow analyzer for the programming language concerned. Of course, this approach can only perform anomaly detection on paths actually executed.

More radical techniques that have much in common with executable assertions include "provably safe" programming [6], and the "recovery block" approach to software fault tolerance—in which an "acceptance test" (effectively an executable assertion) governs the invocation of alternative program components to replace those that have failed [5]. An experiment by Anderson [4] showed promise for recovery blocks (70% of software failures were eliminated, and MTBF was increased by 135%), but found that acceptance tests are hard to write. In another study [28], 24 students

argument is provided to justify the claim that the component satisfies its specification. If a verification is difficult or unconvincing, the program is revised and simplified so that the argument for its correctness becomes more perspicuous.

Mathematical verification subsumes structural testing in the Clean Room methodology, although functional testing is still performed. However, the major testing effort performed in the Clean Room is random testing for the purposes of statistical quality control. Software specifications in the Clean Room methodology include not only a description of the functions to be supported by the software, but a probability distribution on scenarios for its use. The Clean Room methodology then prescribes a testing procedure and a method for computing a certified statistical quality measure for the delivered software.

The mathematical verification technique used in the Clean Room methodology is called "functional verification" and is different from that employed in "classical" mathematical verification. The "rigorous" techniques of Jones [69], and of the Oxford school [56] are based on more conventional forms of mathematical verification than the Clean Room methodology, and are also more formal, but share much of their motivation with the Clean Room approach. An empirical evaluation of the Clean Room methodology has recently been reported by Selby *et al* [111].

Beyond the techniques described above come the *totally* formal methods. These generally employ the assistance of an automated specification and verification environment: the sheer quantity of detail entailed by complete formality is likely to render verification less, rather than more, reliable unless mechanical assistance is employed. Formal specification and verification environments generally provide a formal specification language, a programming language, a means of reducing the problem of establishing consistency between a program and its specification to a putative theorem in some formal system, and a mechanical theorem prover for seeking, or checking, proofs for such theorems. Three or four systems of this type have been developed [77].

All verification methods are vulnerable to two classes of error. First is the possibility of a flaw in the verification itself—the demonstration of consistency between the program and its specification may be flawed, and the two descriptions may, in fact, be inconsistent. The second class of error arises when the verification is sound, but irrelevant, because the specification does not reflect the actual user requirements (i.e., the system may satisfy the verification process, but fail validation)

It is in order to avoid the first class of error that truly formal, and mechanically assisted, formal verification is generally recommended. There are, however, drawbacks to this approach. Firstly, in order to be amenable to mechanization, rather restrictive specification and programming languages, built on a very elementary formal system (usually a variation on first-order predicate calculus), must usually be employed. Because of their lack of convenient expressiveness, such languages and logics may make it difficult for the user to say what he really intends—so that in-

that it remains constructive and purposeful. As the design and its implementation become understood, the attention shifts to a conscious search for faults. A checklist of likely errors may be used to guide the fault finding process.

One of the main advantages of structured walk-throughs over other forms of testing is that it does not require an executable program, nor even formal specifications—it can be applied early in the design cycle to help uncover errors and oversights *before* they become entrenched.

5.2.3 Mathematical Verification

Mathematical verification is the demonstration of consistency between two different descriptions of a program. Usually, one description is the program code itself and the other its specification, though the method can equally well be applied to two specifications at different levels of detail. This terminology is entirely consistent with the notion of "verification" defined in Section 2.1 (page 4), but the presence of the adjective "mathematical" qualifies this particular style of verification as a mathematical activity, in which the two program descriptions are treated as formal, mathematical texts and the notion of "consistency" that is to be demonstrated between them is also a formal, mathematical one (for example, that of "theory interpretation" or of a homomorphism).

Mathematical verification can be performed at various levels of formality. As stated above, *mathematical* verification means that the process is grounded on formal, mathematical principles. But just as conventional mathematicians do not reduce everything to the level of *Principia Mathematica*, so it is entirely reasonable to perform mathematical verification "rigorously," but informally—that is to say, in the style of a normal mathematical demonstration.[1]

There are many worthwhile points along the spectrum of formality in mathematical verification. At the most informal end is the "Clean Room" methodology espoused by Mills [91]. Though informal in the sense that the process is performed manually at the level of ordinary mathematical discourse, the *process* itself is highly structured and formalized. Its goal is to prevent faults getting into the software in the first place, rather than to find and remove them once they have got in. (Hence the name of the methodology—which refers to the dust-free environment employed in hardware manufacturing in order to eliminate manufacturing defects.)

The two cornerstones of the Clean Room methodology are mathematical verification and statistical quality control. The first requires that precise, formal, specifications are written for all program components and that a detailed mathematical

[1] What we are calling *mathematical* verification is often called *formal* verification; we have chosen our terminology to avoid having to talk about "informal" formal verification which, if it is not an oxymoron, is undoubtedly a solecism. Our usage also avoids confusion with some notions of "formal" verification that are anything but mathematical—the adjective "formal" being used in this case to refer to a highly structured *process* of verification.

of anomalies that are likely to indicate faults include supplying a constant as an actual parameter to a procedure call in which the corresponding formal parameter is modified, and subscripting an array with a loop index whose bounds exceed those declared for the array.

Control flow anomalies include unreachable sections of program, and loops with no exit. These circumstances indicate certain errors; other control flow anomalies may merely indicate "bad style"—for example, jumping into the middle of a loop.

Among the most effective of techniques for anomaly detection are those based on data flow analysis. For example, if it can be determined that the value of a program variable may be used before it has been given a value (i.e., if there is a path to a *use* occurrence that does not pass a *def* occurrence), then it is very likely that a fault is present. Dually, *def* occurrences that do not lead to a subsequent *use* occurrence are also suspect, as are paths that have two *def* occurrences of a variable, with no intervening *use* occurrence.

Information flow analysis is related to data flow analysis, but is rather more sophisticated in tracing the influence between program variables and statements. For example, in the program fragment

```
if x = 0 then y := 1 endif
```

there is no *data* flow from x to y, but the value of x certainly influences the subsequent value of y, and so there is considered to be a flow of *information* from x to y. Information flow analysis is used routinely in computer security verification [40, 114]; its application to more general analysis and anomaly detection is promising [12].

Automated tools have been developed to perform detection of anomalies of the types described above for programs written in a variety of languages [98, 122].

5.2.2 Structured Walk-Throughs

Structured walk-throughs are a method for the manual inspection of program designs and code. The method requires far more than mere "eyeballing": it is highly structured and somewhat grueling—its intensity is such that no more than two two-hour sessions per day are recommended.

As first described by Fagan [45] (see [46] for an update), four participants are required—the *Moderator*, the *Designer*, the *Coder/Implementor/*, and the *Tester*. If a single person performed more than one role in the development of the program, substitutes from related projects are impressed into the review team. The review team scrutinizes the design or the program in considerable detail: typically, one person (usually the coder) acts as a "reader" and describes the workings of the program to the others, "walking through" its code in a systematic manner so that every piece of logic is covered at least once, and every branch is taken at least once. Intensive questioning is encouraged, but it is the Moderator's responsibility to ensure

The average size of the test sets needed to achieve this coverage was 12.8, with the largest being 112. Since the test sets were randomly generated, they were far from optimal, and contained subsets that provided the same coverage as the whole set. Using zero-one integer linear programming, the minimum such subsets were found for each test set. These were found to average only 3.1 in size, with the test set of size 112 being reduced to only 10. Thus, the suggestion is to generate random test data until adequate coverage is achieved relative the chosen structural testing criterion (or until further tests result in little increased coverage). Coverage is measured by instrumenting the program under test. The randomly generated test set is then reduced to minimum size using zero-one integer linear programming, and the test results obtained using this subset are examined.

5.2 Static Testing

Dynamic testing is an important method of *validation*; static testing tends to address the complementary problem of *verification*. Static testing subjects the program text (and its accompanying requirements and specification documents) to scrutiny and review in order to detect inconsistencies and omissions. The scrutiny may operate within a single level, in order to detect internal inconsistencies (this is the basis of anomaly detection, discussed in the next section), or across two levels. In the latter case, the purpose is to establish that the lower level specification (or program) fully and exclusively implements the requirements of its superior specification. Everything in a lower level specification should be *traceable* to a requirement in a higher level specification; conversely, every requirement should ultimately be realized in the implementation.

5.2.1 Anomaly Detection

The idea behind anomaly detection is to look for features in the program that probably indicate the presence of a fault. For example, if a programmer declares a variable name but never otherwise refers to it, he is guilty of carelessness at the very least. More ominously, this situation may indicate that the programmer anticipated a need for the variable early in the programming effort, but later forgot to deal with the anticipated circumstance—in which case the existence of a genuine fault may have been detected. Anomaly detection refers to the process of systematically searching for "suspicious" quirks in the syntactic structure of the program, or in its control or data flow.

At the syntactic level, the example given above is typical of a very fruitful technique: searching for identifiers that are declared but not used. The dual problem of identifiers that are used but not identified usually violates the definition of the programming language concerned and will be caught by its compiler. Other examples

indicates the paths he wishes to explore and the symbolic execution system asserts the necessary truth of the appropriate test predicates.

The output produced by a symbolic execution system consists of the symbolic values accumulated in its variables through execution of a selected path. The programmer can compare these values with those expected. This is very convenient and appropriate for some computations, less so for others. For example, Howden [61] cites a subroutine to compute the sine function using the Maclaurin series $sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$ Two iterations round the main loop in the subroutine yield the symbolic value `X-X**3/6+X**5/120` for the variable `SUM` in which the result is accumulating. This provides much more useful information than would the output values for a couple of isolated points.

Another use for symbolic execution is to help partition the input domain by the execution paths invoked. This is helpful in the generation of test data to satisfy path coverage criteria. Symbolic execution systems and their applications are described by several authors [61, 78]; one of the first such systems was developed in the Computer Science Laboratory of SRI [21].

5.1.5 Automated Support for Systematic Testing Strategies

Given a systematic test selection criterion, the question naturally arises: how does one generate test data satisfying the criterion? For functional testing, there seems little alternative to generating the data by human inspection: generally the requirements and design documents from which functional test data are derived are not formal and not amenable to mechanical analysis. However, it is feasible that automatic test data selection could be performed once human inspection had performed the classification of the input domain: that is to say, human skill would be used to identify the primitive classifications, but the generation of combinations of elements of each classification would be performed mechanically.

Unlike functional testing, structural testing is ideally suited to automation. The program text is a formal object and well suited to systematic exploration. Unfortunately, considerable computation is needed to calculate the input that will exercise a particular path: symbolic execution is generally employed in order to derive the "path predicates" associated with the path, then solutions must be found to the systems of numerical inequalities that they induce. Consequently, most test case generators are severely limited in the class of programs and/or in the class of test criteria that they support. Ince [67] provides a modern survey of the field and also suggests that systematic use of randomly generated test data could provide very good coverage at low cost. The idea, which is elaborated in a short note [68], starts from the observation that relatively small sets of random test data seem to provide quite good coverage [43]. For example, with ten programs ranging in size from 12 to 102 branches, random test data achieved an average branch coverage of 92.80% [68].

values $+k$ and $-k$, where k is a very large value, are likely to suggest themselves. But if the function is used in the context

```
if x > -2 then f(x) endif,
```

the test value $-k$ will not exercise f at all, and faults manifest by negative values for x, for example, will remain undiscovered. Given its *context*, appropriate test values for f might be $-2 \pm \epsilon$ and $+k$.

If a formal specification is available for a program, it may be possible to derive a functional testing strategy from that specification in a highly systematic fashion [57]. Other systematic functional testing strategies are described by Mandl [87] and by Ostrand and Balcer [99].

5.1.4 Symbolic Execution

The model of testing described so far assumes that test data are presented to the program and the results produced are compared with those expected. In practice, however, programmers do not merely examine the final output of the program, but often instrument or modify the program under test so that traces of its control flow and of the intermediate values of its variables are generated during execution. These traces greatly assist the programmer in determining whether the actual behavior of the program corresponds to that intended. Since they provide a peek into the inner workings of the program, traces often yield much more insight than the single datum points provided by tests that only consider input-output values.

Symbolic Execution constitutes a systematic technique for generating information about the inner workings of a program. The idea is to allow program variables to take symbolic values and to evaluate the functions computed by program statements symbolically also. Symbolic execution bears a similar relationship to conventional execution as algebra does to arithmetic. Consider, for example, the following (Fortran) program fragment from a subroutine to compute the sine of an angle:

```
I=3
TERM=TERM*X**2/(I*(I-1))
SUM=SUM+(-1)**(I/2)*TERM
```

We can compute the final values of variables `I, TERM` and `SUM`, given the initial assignments `TERM=SUM=X` to be `I=3`, `TERM=X**3/6`, and `SUM=X-X**3/6`. The process performed, statement by statement, is to substitute the current symbolic values into each variable in the right hand side of each assignment statement, simplify the resulting expression, and let this become the new symbolic value of the variable on the left hand side. In a full symbolic execution system, it is necessary to be able to carry the computation forward through branches. Typically, the programmer

some subsequent *c-use* should be included. The *all-c-uses/some-p-uses* criterion is defined dually.

The *all-uses* criterion is a comprehensive one but, like the *all-paths* criterion, may require an excessive, or infinite, number of test cases. The *all-defs* criterion seems an attractive compromise from this point of view. Systematic test selection criteria should surely draw on both data and control flow perspectives—so a very reasonable criterion would seem to be one that encompassed both *all-defs* and *all-edges*. Rapps and Weyuker [100] showed that these are independent criteria—neither implies the other. This tends to confirm the belief that both are important, but complicates test selection since two very different criteria are involved. It would be nice if a single criterion could be found that would include both *all-defs* and *all-edges*. Rapps and Weyuker [100] showed that *all-p-uses/some-c-uses* has this property and recommended its use. Ntafos [97] provides a comprehensive comparison of the relative coverage of these and other structural testing strategies. Clarke [33] provides a more formal examination.

5.1.3.2 Functional Testing

Functional testing selects test data on the basis of the *function* of the program, as described in its requirements, specification, and design documents. Generally speaking, the detailed characteristics of the program itself are not considered when selecting test data for functional testing, though general aspects of its design may be.

Functional testing treats the program as a "black box" which accepts input, performs one of a number of possible functions upon it, and produces output. Based on the relevant requirements documents, classifications and groupings are constructed for the input, output, and function domains. Test data are then constructed to exercise each of these classifications, and combinations thereof. Typically, an attempt is made to select test data that lie well inside, just inside, and outside each of the input domains identified. For example, if a program is intended to process "words" separated by "whitespace," we might select test data that consists of zero words, one word, several words, and a huge number of words. Similarly, we would select words consisting of but a single letter, a few letters, and very many letters—not to mention words containing "illegal" letters. The "whitespace" domain would be explored similarly.

Functional testing for a given program may take many forms, depending on which of its requirements or design documents are used in the analysis. Howden [63] argues that for maximum effectiveness, functional testing should consider the "high-level" design of a program, and the context in which functions will be employed. For example, if a particular function $f(x)$ has the domain $x \in (-\infty, \infty)$, then the test

exercised. In terms of the control flow graph of the program, the first of these criteria requires that all *nodes* must be visited during testing; the second requires that all *edges* must be traversed, and properly includes the first (unless there are isolated nodes—which surely represent errors in their own right).

Test data that simply visits all nodes, or traverses all edges, may not be very effective: not many faults will be so gross that they will be manifest for *all* executions of the offending statement. Generally, faults are manifest only under certain conditions, determined by the context—that is to say, the values of the accessible variables—in which the offending statement is executed. Context is established in part by the execution path taken through the control flow graph; the *path* testing criterion requires test data that will exercise all possible paths through the program.

There are (at least) two problems with the all-paths criterion. Firstly, a path does not uniquely establish an execution context: two different sets of input values may cause the same execution path to be traversed, yet one set may precipitate a failure while the other does not. Secondly, any program containing loops has an infinite number paths. Some further equivalence partitioning is therefore needed. One plausible strategy is to partition execution paths through each cycle in the control flow graph into those involving zero, one, and many iterations.

A more sophisticated strategy considers the *data flow* relationships in the program. Data flow analysis, which was first studied systematically for its application in optimizing compilers, considers how values are given to variables, and how those values are used. Each occurrence of a variable in a program can be classified as a *def*, or as a *use*: a *def* occurrence (for example, an appearance in the left hand side of an assignment statement) assigns a value to a variable; a *use* occurrence (for example, an appearance in the right hand side of an assignment statement) makes use of the value of a variable. *Use* occurrences may be further distinguished as *c-uses* (the value is used in a computation that assigns a value to a variable) and *p-uses* (the value is used in a predicate that influences control flow). For example, in the program fragment

```
if x = 1 then y := z endif,
```

x has a *p-use*, z a *c-use*, and y a *def*.

Rapps and Weyuker [100] proposed a family of path selection criteria based on data flow considerations. The *all-defs* criterion requires that for every *def* occurrence of, say, variable x, the test data should cause a path to be traversed from that *def* occurrence to some *use* occurrence of x (with no intervening *def* occurrences of x). The *all-p-uses* criterion is similar but requires paths to be traversed from that *def* occurrence to every *p-use* occurrence of x that can be reached without intervening *def* occurrences of x. Definitions for the criteria *all-c-uses*, and *all-uses* are similar. A hybrid criterion is *all-p-uses/some-c-uses*—this is the same as *all-p-uses*, except that if there are no *p-uses* of x subsequent to a given *def* occurrence, then a path to

because it captures the basic idea underlying all attempts to create thorough test strategies, namely the search for test criteria that are both reliable and valid, yet economical in the sense that they can be satisfied by relatively small test sets.

Unfortunately, there are several problems with the practical application of these definitions [121]. First of all, the concepts of reliability and validity are not independent. A test selection criterion is valid if, given that the program is faulty, at least one test set satisfying the criterion is unsuccessful. Therefore, if a test selection criterion is *invalid*, all test sets that satisfy it must be successful. Hence they will all give the same result, and so the criterion is reliable. Thus, *all* test selection criteria are either valid or reliable (or both) [121]. (Note also that if F is correct, then *all* criteria are both reliable and valid for F.)

Next, the concepts of validity and reliability are relative to a single, given program. A criterion that is valid and reliable for program F may not be so for the slightly different program F'. Furthermore, since F' may result from correcting a bug in F, the reliability and validity of a test selection criterion may not be preserved during the debugging process.

A plausible repair to this deficiency is to construct modified definitions which quantify over all programs. Thus, a test selection criterion is said to be *uniformly valid* if it is valid for all programs F, and *uniformly reliable* if it is reliable for all programs F. Unfortunately, a criterion is uniformly reliable if and only if it selects a single test and uniformly valid if and only if the union of the test sets that satisfy it is the entire input space D. Hence a criterion is uniformly reliable and valid if and only if it selects the single test set D. Thus we see that the notions of uniform reliability and validity are unhelpful [121]. A more recent attempt to axiomatize the notion of software test data adequacy is described by Weyuker [120].

In theory, the construction of a reliable and valid test selection criterion for a program is equivalent to proving the formal correctness of that program—a very hard problem. In practice, constructing such a criterion is virtually impossible without some knowledge, or at least some assumptions, about the faults that it may contain. Thus, the search for a theoretical basis for thorough test selection has shifted from the original goal of demonstrating that *no* faults are present, to the more modest goal of showing that *specified classes* of faults are not present. In practice, the goal is often reduced to that of finding as many faults as possible. This is accomplished by systematic exploration of the state space based on one (or both) of two strategies known as structural testing, and functional testing, respectively.

5.1.3.1 Structural Testing

Structural testing selects test data on the basis of the program's *structure*. As a minimum, test data are selected to exercise all *statements* in the program; a more comprehensive criterion requires that all outcomes of all decision points should be

5.1.3 Thorough Testing

The motivation behind random testing is find and remove the most costly faults (generally interpreted as those that most frequently lead to operational failure) as quickly and as cheaply as possible. Test selection without replacement and equivalence partitioning may be used to enhance the testing compression factor and hence the cost-effectiveness of the process. The motivation behind what we will call *thorough testing*, on the other hand, is to find *all* the faults in a program (as efficiently as possible). The theoretical framework for thorough testing was established by Goodenough and Gerhart in a landmark paper [52].

Let F denote the program being considered, and D its input space. If $d \in D$ then $\text{OK}(d)$ denotes that F behaves correctly when given input d. We say that the *test set* $T \subseteq D$ is *successful* if F behaves correctly on all its members, that is $\text{SUCCESSFUL}(T) \stackrel{\text{def}}{=} (\forall t \in T : \text{OK}(t))$. A *correct* program is one that behaves correctly throughout its input space—i.e., one that satisfies $\text{SUCCESSFUL}(D)$. A *faulty* program is one that is not correct—i.e., one that satisfies $\neg\text{SUCCESSFUL}(D)$. A *thorough* test set is one which, if successful, guarantees that the program is correct, that is

$$\text{THOROUGH}(T) \stackrel{\text{def}}{=} \text{SUCCESSFUL}(T) \supset \text{SUCCESSFUL}(D).$$

If C is a *test selection criterion* and the test set T *satisfies* C, then we write $\text{SATISFIES}(T, C)$. We would like test criteria to be *reliable* and *valid*. A criterion is reliable if all test sets that satisfy it give the same result:

$$\begin{aligned}\text{RELIABLE}(C) \quad \stackrel{\text{def}}{=} \quad & (\forall T_1, T_2 \in D : \text{SATISFIES}(T_1, C) \wedge \text{SATISFIES}(T_2, C) \\ & \qquad \supset \text{SUCCESSFUL}(T_1) = \text{SUCCESSFUL}(T_2)).\end{aligned}$$

A criterion is valid if it is capable of revealing all faulty programs. That is, if F is faulty, then there should be some T that satisfies C and fails on F:

$$\text{VALID}(C) \stackrel{\text{def}}{=} \neg\text{SUCCESSFUL}(D) \supset (\exists T : \text{SATISFIES}(T, C) \wedge \neg\text{SUCCESSFUL}(T)).$$

Given these definitions, Goodenough and Gerhart were able to state the *Fundamental Theorem of Testing*:

$$\begin{aligned}(\exists T, C : & \text{RELIABLE}(C) \wedge \text{VALID}(C) \wedge \text{SATISFIES}(T, C) \wedge \text{SUCCESSFUL}(T)) \\ & \supset \text{SUCCESSFUL}(D).\end{aligned}$$

In words, this says that if a successful test set satisfies a reliable and valid criterion, then the program is correct. Another way of stating this result is that a test is thorough if it satisfies a reliable and valid test criterion.

This result is called "Fundamental," not because it is profound (indeed, its proof is a trivial consequence of the definitions, and is omitted for that reason), but

so the expected total time to cover the input space during operation is $\sum_{k=1}^{|I|} \frac{p_k}{p_{\min}} \tau_k$. Since only one execution of each input is required during test (without replacement), the execution time to cover the input space during test is only $\sum_{k=1}^{|I|} \tau_k$, and so the testing compression factor is given by:

$$C = \frac{\sum_{k=1}^{|I|} \frac{p_k}{p_{\min}} \tau_k}{\sum_{k=1}^{|I|} \tau_k}. \tag{5.1}$$

If all τ_k are equal, then (5.1) simplifies to

$$C = \frac{1}{|I| \cdot p_{\min}}.$$

If p_k is assumed to be inversely proportional to k, then it can be shown that as $|I|$ grows from 10^3 to 10^9, C grows slowly from just less than 10 to a little over 20.

If a value for the testing compression factor can be calculated or estimated using the methods given above, then the reliability formulas of Section 3.1 (page 8) can be employed to extrapolate from the testing to the operational phase if execution times are multiplied by C (alternatively, failure intensities can be *divided* by C) in order to convert observations made during test to those expected during operation.

The pure random test selection strategy described above assumes that all failures have equal cost. In practice, some failures may be more expensive than others; some may be unconscionable (perhaps endangering human life). In these cases, the random test strategy may be modified in order to attempt to identify and provide early tests of those inputs which might provoke especially costly failures. The technique of software fault-tree analysis may be useful in identifying such critical inputs [86, 83]–see Section 5.2.4.

5.1.2 Regression Testing

The desirability of test selection without replacement is considerably diminished if we admit the possibility of imperfect debugging. Under the assumption that debugging is imperfect, the repair that is initiated following each failure may not only fail to remove the fault that caused it, but may actually introduce *new* faults. These faults may cause failure on inputs that previously worked satisfactorily. Under a *regression testing* regime, previously tested inputs are repeated whenever a repair (or other modification) is made to the program. Under *strict* regression testing, *all* previous tests are repeated; under less strict regimes, some subset of those tests (usually including all those that provoked failure) are repeated. Models of the software testing process under various assumptions are discussed by Downs [42].

or cases that they believe the designers may have overlooked. This tactic may be successful in finding bugs, but because those bugs are manifested by rare or unusual inputs, they may not be encountered during normal operation and may therefore contribute little to the incidence of operational failures.

5.1.1 Random Test Selection

This argument may be developed into a case for *random* testing. If individual failures are all assumed to have similar cost, then the cost of unreliability is proportional to failure intensity. Since the cost of testing is primarily influenced by the execution time taken to perform the tests themselves, the optimum test strategy is one which yields the greatest reduction in failure intensity for a given amount of execution time devoted to testing. This argues for concentrating on finding and eliminating those faults that lead to failures on commonly occurring inputs—and therefore suggests the strategy of selecting test cases randomly, with a probability distribution equal to that expected to occur during operation. This approach has the advantage that it essentially duplicates the operational profile and therefore yields failure intensity data that approximates that which would have been found if the program had been released into operation at that time. The failure intensity data obtained during testing should therefore provide accurate estimates for the reliability formulas developed in Section 3.1 (page 8).

A criticism of the random testing strategy is that it is wasteful: the *same* input may be tested several times. However, it is usually easy to record each input and to avoid repeating tests already performed. By reference to the classic sampling problems of statistics, such methods are often called "test selection without replacement." If tests are selected and performed without replacement, then the probability of selecting any particular input value becomes uniform. However, the *order* in which inputs are selected will still follow their expected probability of occurrence in operation.

Although random test selection without replacement should be more cost-effective than a purely random strategy, such tests do not follow the expected operational profile and therefore do not provide an accurate estimate of the failure intensities to be expected in practice. It is possible to take account of this divergence between the testing and the operational profiles as follows.

Define C, the *testing compression factor* to be the ratio of the execution time required to cover the entire input space during operation, to that required during test. Let $|I|$ be the size (cardinality) of the input space, p_k the probability of occurrence of input k during operation, $p_{\min}$ the probability of occurrence of the *least* likely input, and let τ_k be the execution time required for input k. If the entire input space is covered during operation, then the least likely input must be executed at least once. Hence the expected frequency of execution of input k is $p_k/p_{\min}$ and

Chapter 5

Testing

Testing subjects software to scrutiny and examination under the controlled conditions of a "test phase" before it is released into its "operational phase." The purpose of testing is to discover faults and thereby prevent failures in the operational phase. Discussion of the efficiency of a test strategy must relate the actual cost of testing to the avoided costs of the operational failures that are averted.

Testing can take two forms: *static*, and *dynamic*, though the unadorned term "testing" generally refers only to the latter. Static testing is that which depends only on scrutiny on the program text (and possibly that of its specifications and requirements also). Dynamic testing, on the other hand, observes the behavior of the program *in execution*. We will describe dynamic testing first, then three variants of static testing: anomaly detection, code walk-throughs, and formal verification.

5.1 Dynamic Testing

The input space of any interesting program is so large (if not infinite) that it is infeasible to examine the behavior of the program on all possible inputs. Given that only a fraction of the total input space can be explored during testing, systematic testing strategies attempt to maximize the likely benefit while minimizing the cost of testing. Two tactics underlie most systematic testing strategies: the first is to reduce cost by partitioning inputs into groups whose members are *likely* to have very similar behavior, and then testing only one member from each group (this is called "equivalence partitioning"); the second is to maximize benefit by selecting test data from among inputs that are considered to have an above average likelihood of revealing faults. The second of these tactics is not universally admitted to be a good idea: the divergence of opinion occurs between those who are primarily interested in enhancing reliability (actually, in reducing the cost of unreliability), and those who are interested in finding bugs. Concentration on input states believed to be fault-prone often leads testers to examine the boundaries of expected ranges of values,

Effort metrics are rather less controversial and easier to validate than complexity metrics—their purpose, if not their effectiveness, is clear: to enable the cost and duration of a programming task to be estimated in advance. However, the efficacy of existing effort metrics seems dubious at best. The COCOMO model, for example, seems to perform very poorly when applied to projects other than those used in its own validation. Kemerer [75], for example, reported an *average* error of over 600%, and also found that the Basic COCOMO mode outperformed the more complex Intermediate and Detailed modes. Conte *et al.* [36] suggest that the COCOMO model is *too* complicated, and involves too many parameters that require estimation. Kemerer's data [75] suggests that a much simpler "function count" method, similar to the Software Science metric, actually performs better than COCOMO.

Kearney *et al.* also criticise the fact that complexity measures are developed without regard to the intended *application* of the measure. There is a significant difference between *prescriptive* uses of such metrics (using them to "score" programmer's performance), and merely *descriptive* uses. A much greater burden of proof attends the former use, since we must be sure that techniques that improve the score really do improve the program. Kearney *et al.* find much to criticize in the methodology and interpretation of experiments that purport to demonstrate the significance and merit of selected complexity measures, and thereby cast serious doubt on the wisdom of using complexity metrics in a prescriptive setting.

Criticism such as that levied by Kearney and his colleagues against much of the work in complexity metrics is acknowledged by the more thoughtful practioners. Shen, for example, introducing a magazine column dedicated to metrics [113], writes:

> "Metrics researchers in the 1980's are generally less optimistic than their colleagues in the 1970's. Even though the pressure to find better metrics is greater because of the greater cost of software, fewer people today are trying to formulate combinations of complexity metrics that they relate to some definition of productivity and quality. Instead they set very narrow goals and show whether these goals are met using focussed metrics ...one narrow goal would be to test software thoroughly. An appropriate metric might be some measure of test coverage."

The issue addressed in this report is software quality; size and complexity metrics are of interest only in so far as they contribute to the estimation or improvement of software quality. As far as estimation is concerned, metrics research does not yet seem able to provide accurate predictions for the properties of economic interest (number of faults, cost to maintain or modify) from the measurable properties of the program itself. The most accurate and best validated predictive techniques seem to be the crudest—for example 20 faults per KLOC for a program entering unit test [93, page 121]. Even these techniques are likely to need considerable calibration and re-validation when used in new environments.

The application of software metrics to improving (as opposed to predicting) the quality of software is even more questionable. Given their lack of substantiation, the use of such metrics prescriptively seems highly ill-advised [74]. Their least controversial application might be as components of "program comprehension aids." By these, we mean systems that assist a programmer to better comprehend his program and to master its complexities—especially those due to scale. Examples of such aids range from simple prettyprinters and cross-reference generators, to more semantically oriented browsers. Descriptive complexity metrics might assist the programmer in identifying parts of the program worthy of restructuring or simplification in much the same way that profiling tools help to identify the places where optimization can be applied most fruitfully.

the original Intermediate model. For all cost drivers, a "nominal" value corresponds to a multiplier of 1.00.

Not all programs are written from scratch; the effort to develop a program that includes a substantial quantity of reused code should be less than one of comparable total size that consists of all new code. Various modifications have been suggested to accommodate this factor; the simplest is to treat the total size of the program (S_e) as a linear combination of the number of lines of new (S_n) and "old" code (S_o):

$$S_e = S_n + kS_o,$$

where k is an appropriate constant—a value of $k = 0.2$ has been found reasonable [36], though this could vary if the "old" code requires significant adaptation.

4.4 Discussion of Software Metrics

The attempt to measure and quantify the properties and characteristics of programs is a laudable one—and one on which considerable effort has been expended (see, for example, the survey and bibliography by Waguespack and Badlani [118]). However, the substance and value of most of this work is open to serious question. Kearney *et al.* marshal the arguments against the standard complexity measures in their critique [74]. Firstly, they observe that existing complexity measures have been developed in the absence of a theory of programming behavior—there is no comprehensive model of the programming process that provides any intellectual support for the metrics developed. Any reasonable theory of programming behavior would consider not only the program, but also the programmer (his skill and experience), the programming environment, and the task that the program is to accomplish. Yet existing complexity measures consider only the program itself, and ignore its context. Furthermore, most complexity measures examine only the surface features of programs—their lexical and syntactic structure—and ignore the deeper semantic issues. Indeed, most metrics research seems stuck in the preoccupations of the 1960's: it equates complexity with control flow (c.f. the "structured programming" nostrums of the time) and seems unaware that the serious work on programming methodology over the last 15 years has been concerned with the deeper problems of hierarchical development and decomposition, and with the issues of data structuring, abstraction, and hiding. Some more recent work does attempt to address these issues—for example, Bastani and Iyengar [10] found that the perceived complexity of data structures is strongly determined by the relationship between the concrete data representation and its more abstract specification. They conjectured that a suitable measure for the complexity of a data structure is the length of the mapping specification between its abstract and concrete representations—i.e., a *semantic* measure.

Rating	Multiplier
Very Low	0.75
Low	0.88
Nominal	1.00
High	1.15
Very High	1.40

Figure 4.1: Multipliers for the Reliability Cost Driver in the COCOMO Model

this approach is the "Delphi" technique in which several people prepare independent estimates and are then told how their estimates compare with those of the others. (In some variants, they discuss their estimates with each other). Next, they are allowed to modify their estimates and the process is repeated until the estimates stabilize. Often the estimates converge to a very narrow range, from which a consensus value may be extracted.

A composite cost estimation technique (which combines the use of expert judgment with statistical data fitting) is the COCOMO (COnstructive COst MOdel) of Boehm [17, 16]. There are three "levels" of the COCOMO model; the most successful seems to be the (modified) Intermediate Level. There are also three "modes" to the COCOMO model—for simplicity we will pick just one (the "semidetached" mode). This variant of the COCOMO predicts development effort E in man-months by the equation

$$E = 3.0 \times S^{1.12} \times \prod_{i=1}^{16} m_i$$

where S is the estimated KLOC and each m_i is the multiplier assigned to a particular "cost driver." Development time (in months) can be obtained from the COCOMO effort measure by the equation $T = 2.5E^{0.35}$. Each of the 16 cost drivers is evaluated by experts on a five point descriptive scale, and a numerical value for the corresponding multiplier is then extracted from a table. The table for the "reliability" cost driver is given in Figure 4.1. The 16 cost-drivers used in the (modified) Intermediate COCOMO model are: required software reliability, data base size, product complexity, execution time constraint, main storage constraint, virtual machine volatility, computer turnaround time, analyst capability, applications experience, programmer capability, virtual machine experience, programming language experience, modern programming practice, use of software tools, required development schedule, and volatility of requirements. It is the presence of the last cost driver (volatility of requirements) that distinguishes the modified Intermediate COCOMO model from

4.3 Cost and Effort Metrics

Cost and effort metrics attempt to measure, or predict, the eventual size of a program, the cost required to construct it, the "effort" needed to understand it, and other such interesting and important attributes.

Halstead's "Software Science" postulates several composite metrics that purport to measure such attributes. Halstead hypothesized that the length of a program should be a function of the numbers of its distinct operands and operators. That is, it should be possible to predict N, the length of a program, from η_1 and η_2—the numbers of its distinct operators and operands, respectively. Halstead encoded a relationship between these quantities in his famous "length equation":

$$\widehat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2.$$

Going further, Halstead defined the "potential volume" V^*, of a program as the volume of the program of minimal size that accomplishes the same purpose as the original. The "difficulty" D of a program is then defined by $D = V/V^*$, and its "level" L is defined as the reciprocal of difficulty. It is then but a small step to hypothesize that the effort required to implement a program should be related to both its length and its difficulty, and to define the "effort" E required to implement a program by $E = D \times V (= V^2/V^*)$. Halstead claimed that E represented "elementary mental discriminations" and, based on the suggestion of a psychologist, J. Stroud, that the human mind is capable of making only between 5 and 20 elementary mental discriminations per second, he then proposed the equation $T = E/18$, where T is the time required to construct a program in seconds.

The practical application of these last few formulae depend on a method for computing or estimating the potential volume, V^*, for a program. Halstead proposed that potential volume could be *defined* by

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*),$$

where η_2^* is the number of input/output operands, and *estimated* by

$$V^* = \frac{2\eta_2}{\eta_1 N_2} \times V.$$

Using this estimation, we obtain

$$D = \frac{\eta_1 N_2}{2\eta_2} \quad \text{and} \quad E = V \times \frac{\eta_1 N_2}{2\eta_2}.$$

In contrast to the scientific pretensions of Software Science, a widely practised method for predicting the time and effort required to construct a software project is simply to ask the opinions of those experienced in similar projects. A refinement of

of "diamonds" (DC) minus one (for the exit node). Thus $e = n + DC - 1$. Hence

$$\begin{aligned} \nu &= e - n + 2 \\ &= (n + DC - 1) - n + 2 \\ &= DC + 1. \end{aligned}$$

Thus, the highfalutin cyclomatic complexity measure turns out to be no different than the elementary decision count!

Related to syntactic complexity measures are "style" metrics [13, 14, 55, 102, 101]. These seek to score programs on their adherence to coding practices considered superior, in some sense. The details of these metrics vary according to the individual preference of their inventors. Generally, marks are added for instances of "good" style such as plenty of comments, and the use of symbolic constants, and deducted for instances of "bad" style such as explicit `goto`'s, and excessive module length. Some program style analysis systems also perform limited anomaly detection (e.g., variables declared but not used) and incorporate these into their scores. We discuss anomaly detection separately in Section 5.2.1 (page 33).

4.2.2 Measures of Data Complexity

The simplest data complexity metrics simply count the number of variables used by a program. Halstead's η_2 (the number of distinct operands) is a slightly more elaborate measure of the same type. Such metrics only measure the total number of different variables that are used in the program; they do not indicate how many are "live" (i.e., must be actively considered by the programmer) in any one place. A simple definition states that a variable is "live" in all statements lexically contained between its first appearance and its last. It is then easy to compute the number of variables that are live at each statement, and hence $\overline{LV}$—the *average* number of variables that are live at each statement.

Another approach to quantifying the complexity of data usage is to measure the extent of inter-module references. The work of Henry and Kafura [59, 58] is representative of this type. The *fan-in* of a module may be defined as the number of modules that pass data to the module, either directly or indirectly; similarly the *fan-out* may be defined as the number of modules to which the present module passes data, either directly or indirectly. The complexity of the interconnections to the module are then defined as $(\text{fan-in} \times \text{fan-out})^2$. Henry and Kafura then relate the overall complexity of a module within a program to both its length and the complexity of its interconnections by the definition: complexity = $\text{SLOC} \times (\text{fan-in} \times \text{fan-out})^2$.

4.2 Complexity Metrics

Two similarly sized programs may differ considerably in the effort required to comprehend them, or to create them in the first place. *Complexity* metrics attempt to quantify the "difficulty" of a program. Generally, these metrics measure some aspect of the program's control flow—there being some agreement that complicated control flow makes for a complex, hard-to-understand, program.

4.2.1 Measures of Control Flow Complexity

The simplest complexity metric is the *decision count*, denoted DC which can be defined as the number of "diamonds" in a program's flow chart. A good approximation to DC can be obtained by counting the number of conditional and loop constructs in a program—and this can be reduced to the purely lexical computation of adding the number of `if`, `case`, `while` and similar keywords appearing in the program.

An objection to this simple scheme is that it assigns a different complexity to the program fragment

```
if A and B then X endif
```

than it does to the semantically equivalent fragment

```
if A then if B then X endif endif.
```

This objection can be overcome by defining DC to be the number of *elementary predicates* in the program. Both the examples above contain two elementary predicates: A and B.

Much the best-known of all syntactic complexity measures is the *cyclomatic complexity* metric of McCabe [88]. This metric, denoted ν, is given by $\nu = e - n + 2$, where e is the number of edges and n the number of nodes in the control flow graph of the program. The cyclomatic complexity of a program is equivalent to the maximal number of linearly independent cycles in its control flow graph. (Actually, in the control flow graph modified by the addition of an edge from its exit point back to its entry point.) Clearly, the simplest control flow graphs (i.e., those corresponding to linear sequences of code) have $e = n - 1$ and hence $\nu = 1$. Motivated by testing considerations, McCabe suggested that a value of $\nu = 10$ as a reasonable upper limit on the cyclomatic complexity of program modules.

Despite their different origins, and the apparently greater sophistication of cyclomatic complexity, DC and ν are intimately related. Each "rectangle" in a program's flow chart has a single outgoing edge (except the exit node, which has none); each "diamond" has two outgoing edges. Therefore, the total number of edges in the flow chart is equal to the total number of nodes in the flow chart (n), plus the number

obvious objection to these measures is that they do not account for the different "densities" of different programming languages (e.g., a line of APL is generally considered to contain more information, and to require more effort to write, than a line of Cobol), and they do not account for the fact that different lines in the same program, or lines written by different people, may have very different amounts of information on them. A straightforward attempt to overcome the latter objection (and perhaps the former also) is to count syntactic tokens rather than lines.

An early, controversial, and influential system of this type was the "Software Science" of Halstead [54]. Software Science classifies tokens as either *operators* or *operands*. Operators correspond to the control structures of the language, and to system and user-provided procedures, subroutines, and functions. Operands correspond to variables, constants, and labels. The basic Software Science metrics are then defined as follows:

η_1: the number of distinct operators,

η_2: the number of distinct operands,

N_1: the total number of operators, and

N_2: the total number of operands.

The *length* of a program is then defined as $N = N_1 + N_2$. It is a matter of taste, if not dispute, how the multiple keywords of iterative and conditional statements should be counted (e.g., does one count each of `while do`, and `endwhile` as three separate operators, or as a single "while loop" operator). Similarly controversial is the question whether tokens appearing in declarations should be counted, or only those appearing in imperative statements (opinion currently favors the first alternative). The Software Science metric N may be converted to SLOC by the relationship SLOC $= N/c$, where c is a language-dependent constant. For FORTRAN, c is believed to be about 7.

Software Science derives additional metrics from the basic terms. The *vocabulary* is defined as $\eta = \eta_1 + \eta_2$. Clearly it requires $\log_2 \eta$ bits to represent each element of the vocabulary in a uniform encoding, so the number of bits required to represent the entire program is roughly $V = N \log_2 \eta$. The metric V is called the *volume* of a program and provides an alternative measure for the size of a program.

An alternative attempt to quantify the size of a program in a way that is somewhat independent of language, and that may get closer to measuring the semantic, rather than the purely syntactic or even merely lexical, content of a program is one based on function count: that is, a count of the number of *procedures* and *functions* defined by the program. Lisp programs are often described in this way—for example, a medium sized Lisp application might contain 10,000 function definitions. For some languages (e.g., Ada), the number of *modules* might be a more natural or convenient measure than function count.

Chapter 4

Size, Complexity, and Effort Metrics

In the previous chapter, we have seen that à priori estimates for the reliability of a program depend on estimates for the number of faults it contains and for its initial failure intensity. It is plausible to suppose that these parameters may themselves be determined by the "size" and "complexity" of the program. Accordingly, considerable effort has been expended in the attempt to define and quantify these notions.

Whereas measurements of the static properties of completed programs (e.g., size and complexity) may help in predicting some aspects of their behavior in execution (e.g., reliability), similar measurements of "requirements" or "designs" may help to predict the "effort" or cost required to develop the finished program.

In this chapter we will examine metrics that purport to measure the size, and complexity of programs, and those that attempt to predict the cost of developing a given piece of software.

4.1 Size Metrics

The *size* of a program is one of its most basic and measurable characteristics. It seems eminently plausible that the effort required to construct a piece of software is strongly influenced, if not fully determined, by its final size, and that its reliability will be similarly influenced.

The crudest measure of the size of a piece of software is its *length*, measured by the number of lines of code that it contains. A line of code is counted as any non-blank, non-comment line, regardless of the number of statements or fragments of statements on the line. The basic unit of program length is the "SLOC"—a single "Source Line Of Code." A variant is the "KLOC"—a thousand lines of code. An

$$
\begin{aligned}
&= 18.53 \ln 68 \\
&= 18.53(4.22) \\
&= 78 \text{ CPU-hours.}
\end{aligned}
$$

3.2 Discussion of Software Reliability

Software reliability modeling is a serious scientific endeavor. It has been pursued diligently by many of those with a real economic stake in the reliability of their software products—for example, the manufacturers of embedded systems (where repair is often impossible), and those with enormously stringent reliability objectives (for example, the manufacturers of telephone switching equipment).

The "Basic Execution Time Model" described here has been validated over many large projects [93] and has the virtue of relative simplicity compared with many other models. A similar model, the "Logarithmic Poisson Model," has been less intensively applied, but may be preferred in some circumstances. Both of these models use execution-time as their time base. This is one of the primary reasons for their greater accuracy over earlier models, which used man-hours, or other human-oriented measures of time. That execution time should prove more satisfactory is not surprising: the number of failures manifested should surely be most strongly determined by the amount of exercise the software has received. Converting from a machine-oriented view of time to a human-oriented view is often necessary for the application of the results obtained from the model; ways of doing this are described by Musa *et al.* [93].

There are several circumstances that can complicate the application of reliability models. Reliability is concerned with counting failures, and prediction is based on collecting accurate failure data during the early stages of a project. These data may be unreliable, and predications based upon them may be inaccurate, if any of the following circumstances obtain:

- There is ambiguity or uncertainty concerning what constitutes a failure,
- There is a major shift in the operational profile between the data gathering (e.g., testing) phase and the operational phase, or
- The software is undergoing continuous change or evolution.

We note that these circumstances which cause difficulty in the application of reliability modeling, also characterize much AI-software development. We will return to this issue in the second part of the report.

of code—will lead to failure: particular circumstances may be needed to "trigger" the bug). For the purposes of estimation, we may assume that fault encounters are linearly related to the number of instructions executed (i.e., to the duration of execution and to processor speed), and that failures are linearly related to fault encounters. Since processor speed is generally given in *object* instructions per second, we will also assume that the number of object instructions I_o in a compiled program is linearly related to the number of source lines I_s. Thus, if Q is the *code expansion ratio* (i.e., I_o/Is), and R_o is the number of object instructions executed in unit time, then the number of source lines executed in unit time will be given by $R_s = R_o/Q$. Now each source line executed exposes ω_0/I_s faults. Thus $\omega_0 \times R_s/I_s$ faults will be exposed in unit time. If each fault exposure leads to K failures, we see that the initial failure intensity is given by $\lambda_0 = K \times \omega_0 \times R_s/I_s$. Substituting the previously derived expressions for ω_0 and R_s, we obtain:

$$\lambda_0 = D \times K \times \frac{R_o}{Q}. \tag{3.14}$$

In order to complete the estimation of λ_0, we need values for three new parameters: R_o, Q, and K. The first of these is provided by the computer manufacturer, while the second is a function of the programming language and compiler used. A table of approximate expansion ratios is given by Jones [70, page 49]. The most difficult value to estimate is the fault exposure ratio K. Experimental determinations of K for several large systems yield values ranging from 1.41×10^{-7} to 10.6×10^{-7}—a range of 7.5 to 1, with an average of 4.2×10^{-7} [93, page 122].

As an example, of the application of these estimates, consider a 20,000 line program entering the system test phase. Using $D = 6.01$ and $B = 0.955$ as before, and assuming a code expansion ratio Q of 4, a fault exposure ratio K of 4.2×10^{-7} failures per fault, and a 3 MIPS processor, we obtain:

$$\begin{aligned}
\lambda_0 &= D \times K \times \frac{R_o}{Q} \\
&= \frac{6.01}{10^3} \times \frac{4.2}{10^7} \times \frac{3 \times 10^6}{4} \\
&= 1.893 \times 10^{-3} \text{ failures per CPU-sec,} \\
&= 6.8 \text{ failures per CPU-hour.}
\end{aligned}$$

Given these estimates of the parameters ν_0 and λ_0, we may proceed to calculate how long it will take to reduce the fault intensity to an acceptable level—say 0.1 failures per CPU-hour.

$$\begin{aligned}
\delta t &= \frac{\nu_0}{\lambda_0} \ln \frac{\lambda_P}{\lambda_F} \\
&= \frac{126}{6.8} \ln \frac{6.8}{0.1}
\end{aligned}$$

Development Phase	Faults/K source lines
Coding	99.50
Unit test	19.70
System test	6.01
Operation	1.48

Figure 3.1: Fault Density in Different Phases of Software Development

Prior to operational data becoming available, however, we can only attempt to *predict* values for the parameters of the model, using characteristics of the program itself, and the circumstances of its development. The total number of failures can be predicted as the number of faults inherent in the program, divided by the fault-reduction factor: $\nu_0 = \omega_0/B$. Dozens of techniques have been proposed for estimating the number of faults in a program based on static characteristics of the program itself. These are generally related to some notion of "complexity" of programs and are described in detail in the following chapter. For the purposes of exposition, we will use the simplest (yet one of the best) such measures: the length of the program. There is quite good evidence that faults are linearly related to the length of the source program. If we let I_s denote the length of the program (measured by lines of *source* code), and D the fault density (in faults per source line), then we have

$$\omega_0 = I_s \times D \tag{3.12}$$

and

$$\nu_0 = \frac{I_s \times D}{B}. \tag{3.13}$$

Results from a large number of experiments to determine values of D are summarized in Figure 3.1 (taken from [93, page 118]). Experimental determinations of the fault-reduction factor range from 0.925 to 0.993, with an average of 0.955 (again, from [93, page 121]). Thus, an à priori estimate for the number of failures to be expected from a 20,000 line program entering the system test phase is given by

$$\begin{aligned} \nu_0 &= \frac{I_s \times D}{B} \\ &= \frac{20 \times 6.01}{0.955} \\ &= 126 \text{ failures.} \end{aligned}$$

The initial failure intensity λ_0 depends upon the number of faults in the program, the rate at which faults are encountered during execution, and the ratio between fault encounters and failures (not all fault encounters—i.e., execution of faulty pieces

$$\begin{aligned}\mu(10) &= 100\left(1 - e^{-\frac{10}{100}10}\right) \\ &= 100\left(1 - e^{-1}\right) \\ &= 100(1 - 0.368) \\ &= 63 \text{ failures.}\end{aligned}$$

Additional formulae can be derived to give the *incremental* number of failures ($\delta\mu$) or elapsed time (δt) to progress from a known present failure intensity (λ_P), to a desired future goal (λ_F):

$$\delta\mu = \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F), \text{ and} \tag{3.10}$$

$$\delta t = \frac{\nu_0}{\lambda_0}\ln\frac{\lambda_P}{\lambda_F}. \tag{3.11}$$

Using the same example as before (90 faults, fault-reduction factor 0.9, initial failure intensity 10 per CPU-hour), we can ask how many additional failures may be expected between a present failure intensity of 3.68 per CPU-hour, and a desired intensity of 0.000454 per CPU-hour:

$$\begin{aligned}\delta\mu &= \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F) \\ &= \frac{100}{10}(3.68 - 0.000454) \\ &= 10(3.68) \\ &= 37 \text{ failures.}\end{aligned}$$

Similarly we may inquire how long this may be expected to take:

$$\begin{aligned}\delta t &= \frac{\nu_0}{\lambda_0}\ln\frac{\lambda_P}{\lambda_F} \\ &= \frac{100}{10}\ln\frac{3.68}{0.000454} \\ &= 10\ln 8106 \\ &= 10(9) \\ &= 90 \text{ CPU-hours.}\end{aligned}$$

The reliability model just developed is determined by two parameters: the initial fault intensity, λ_0, and the total number of failures expected in infinite time, ν_0. In order to apply the model, we need values for these two parameters. If the program of interest has been in operation for a sufficient length of time that accurate failure data are available, then we can *estimate* the values of the two parameters. Maximum likelihood or other methods of statistical estimation may be used, and confidence intervals may be used to characterize the accuracy of the estimates.

where $\nu_0 = \omega_0/B$ is the total expected number of failures, and

$$G_a(t) = 1 - e^{-B\int_0^t z_a(x)\,dx}$$

is the cumulative distribution function of the time to remove a fault. Similarly

$$\lambda(t) = \nu_0 g_a(t)$$

where $g_a(t)$ is the probability density function associated with $G_a(t)$.

The consequent modifications to the important equations (3.4–3.6) are simple: merely replace ω_0 (the number of faults) by ν_0 (the total number of failures expected). Thus we obtain

$$\lambda(t) = \lambda_0 e^{-\frac{\lambda_0}{\nu_0}t}, \tag{3.7}$$

$$\mu(t) = \nu_0\left(1 - e^{-\frac{\lambda_0}{\nu_0}t}\right), \text{ and} \tag{3.8}$$

$$\lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{\nu_0}\right). \tag{3.9}$$

As an example of the application of these formulae, consider a system containing 90 faults and whose fault-reduction factor is 0.9. The system may be expected to experience 100 failures. Suppose its initial failure intensity was 10 failures per CPU-hour and that it has now experienced 50 failures. Then the present failure intensity should be given by

$$\begin{aligned}\lambda(\mu) &= \lambda_0\left(1 - \frac{\mu}{\nu_0}\right)\\ \lambda(50) &= 10\left(1 - \frac{50}{100}\right)\\ &= 5 \text{ failures per CPU-hour.}\end{aligned}$$

In addition we may ask what the failure intensity will be after 10 CPU-hours and how many failures will have occurred by that time. For failure intensity we have:

$$\begin{aligned}\lambda(t) &= \lambda_0 e^{-\frac{\lambda_0}{\nu_0}t}\\ \lambda(10) &= 10e^{-\frac{10}{100}10}\\ &= 10e^{-1}\\ &= 3.68 \text{ failures per CPU-hour,}\end{aligned}$$

and for failures we have

$$\mu(t) = \nu_0\left(1 - e^{-\frac{\lambda_0}{\mu_0}t}\right)$$

2. The number of faults in the program initially is a Poisson random variable with mean ω_0.

3. The hazard rate for all faults is the same for all faults, namely $z_a(t)$.

Under these additional assumptions, it can be shown that

$$\mu(t) = \omega_0 F_a(t) \tag{3.1}$$

where $F_a(t)$ is the per-fault failure probability, and

$$\lambda(t) = \omega_0 f_a(t) \tag{3.2}$$

where $f_a(t)$ is the per-fault failure density.

The final assumption in the derivation of the model is that the per-fault failure density has an exponential distribution. That is,

$$f_a(t) = \phi e^{-\phi t}. \tag{3.3}$$

(Note this implies that the per-fault hazard rate is constant, that is $z_a(t) = \phi$). Substituting (3.3) into (3.2), we obtain

$$\lambda(t) = \omega_0 \phi e^{-\phi t}.$$

Letting λ_0 denote $\lambda(0)$, we obtain $\phi = \lambda_0/\omega_0$ and hence

$$\lambda(t) = \lambda_0 e^{-\frac{\lambda_0}{\omega_0} t}. \tag{3.4}$$

Similarly, from (3.1) we obtain

$$\mu(t) = \omega 0 \left(1 - e^{-\frac{\lambda_0}{\omega_0} t}\right). \tag{3.5}$$

Some additional manipulation allows us to express failure intensity, λ, as a function of failures observed, μ:

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\omega_0}\right). \tag{3.6}$$

The assumption that the fault responsible for each failure is identified and removed immediately the failure occurs is clearly unrealistic. The model can be modified to accommodate imperfect debugging by introducing a *fault reduction factor* B. This attempts to account for faults that cannot be located, faults found by code-inspection prior to causing failure, and faults introduced during debugging, by assuming that, on average, each failure leads to the removal of B faults ($0 \leq B \leq 1$). (Conversely, each fault will lead to $1/B$ failures.) It can then be shown that

$$\mu(t) = \nu_0 G_a(t)$$

and reliability models are based on the mathematics of random or stochastic processes. Because failures generally provoke (attempted) repair, the number of faults in a program generally changes over time, and so the probability distributions of the components of a reliability model vary with time. That is to say, reliability models are based on *nonhomogeneous* random processes.

A great many software reliability models have been developed. These are treated systematically in the first textbook to cover the field, which has just been published [93]. The most accurate and generally recommended model is the "Basic Execution Time Model." We outline the derivation of this model below.

3.1 The Basic Execution Time Reliability Model

The starting point for this derivation (and that of most other reliability models) is to model the software failure process as a NonHomogeneous Poisson Process (NHPP)—a particular type of Markov model.

Let $M(t)$ denote the number of failures experienced by time t. We make the following assumptions:

1. No failures are experienced by time 0, that is $M(0) = 0$,
2. The process has independent increments, that is the value of $M(t+\delta t)$ depends only on the present value of $M(t)$ and is independent of its history,
3. The probability that a failure will occur during the half-open interval $(t, t+\delta t]$ is $\lambda(t).\delta t + o(\delta t)$, where $\lambda(t)$ is the *failure intensity* of the process.
4. The probability that more than one failure will occur during the half-open interval $(t, t+\delta t]$ is $o(\delta t)$.

If we let $P_{m(t)}$ denote the probability that $M(t)$ is equal to m, that is:

$$P_{m(t)} = \text{Prob}[M(t) = m]$$

then it can be shown that $M(t)$ is distributed as a Poisson random variable. That is:

$$P_{m(t)} = \frac{\mu(t)^m}{m!}\, e^{-\mu(t)}$$

where

$$\mu(t) = \int_0^t \lambda(x)\, dx\,.$$

Next we make some further assumptions:

1. Whenever a software failure occurs, the fault that caused it will be identified and removed instantaneously. (A more realistic assumption will be substituted later).

0.83 for 8 hours when employed by a hacker. To give an idea of the reliabilities demanded of flight-critical aircraft systems (in which software components are increasingly important), the FAA requires the probability of catastrophic failure to be less than 10^{-9} per 10-hour flight for a life-critical civil air transport flight control system; the US Air Force requires the probability of mission failure to be less than 10^{-7} per hour for military aircraft.

From the basic notion of reliability, many different measures can be developed to quantify the occurrence of failures in time. Some of the most important of these measures, and their interrelationships are summarized below:

Reliability, denoted by $R(t)$, is the probability of failure-free operation up to time t.

Failure Probability, denoted by $F(t)$, is the probability that the software will fail prior to time t. Reliability and failure probability are related by $R(t) = 1 - F(t)$.

Failure Density, denoted by $f(t)$, is the probability density for failure at time t. It is related to failure probability by $f(t) = \frac{d}{dt}F(t)$. The probability of failure in the half-open interval $(t, t + \delta t]$ is $f(t).\delta t$.

Hazard Rate, denoted by $z(t)$, is the conditional failure density at time t, given that no failure has occurred up to that time. That is, $z(t) = f(t)/R(t)$. Reliability and hazard rate are related by

$$R(t) = e^{-\int_0^t z(x)\,dx}.$$

An important special case occurs when the hazard rate is a constant ϕ. In this case the failure density has an exponential distribution $f(t) = \phi e^{-\phi t}$, the failure probability is given by $F(t) = 1 - e^{-\phi t}$ and the reliability is given by $R(t) = e^{-\phi t}$.

Mean Value Function, denoted by $\mu(t)$, is the mean *number* of failures that have occurred by time t.

Failure Intensity, denoted by $\lambda(t)$, is the number of failures occurring per unit time at time t. This is related to the mean value function by $\lambda(t) = \frac{d}{dt}\mu(t)$. The number of failures expected to occur in the half-open interval $(t, t + \delta t]$ is $\lambda(t).\delta t$.

Failure intensity is the measure most commonly used in the quantification of software reliability. [1] Because of the complexity of the factors influencing the occurrence of a failure, the quantities associated with reliability are random variables,

[1] Mean Time To Failure (MTTF), denoted by Θ, is not employed to the extent that it is in hardware reliability studies (probably because the models used in software reliability tend not to give rise to closed form expressions for MTTF). This measure is related to reliability by $\Theta = \int_0^\infty R(x)\,dx$.

Chapter 3

Software Reliability

Software reliability is concerned with quantifying how well software functions to meet the needs of its customer. It is defined as the probability that the software will function without failure for a specified period of time. "Failure" means that in some way the software has not functioned according to the customer's requirements. This broad definition of failure ensures that the concept of reliability subsumes most properties generally associated with quality—not only correctness, but also adequacy of performance, and user-friendliness. Reliability is a user-oriented view of software quality: it is concerned with how well the software actually works. Alternative notions of software quality tend to be introspective, developer-oriented views that associate quality with the "complexity" or "structure" of the software. Fortunately, software reliability is not only one of the most important and immediate attributes of software quality, it is also the most readily quantified and measured.

Software reliability is a scientific field, and employs careful definition of terms. The two most important are "failure" and "fault." A software *failure* is a departure of the external behavior of the program from the user's requirements. The notion of requirements is discussed in 5.3.1. A software *fault* is a defect in a program that, when executed under certain conditions, causes a failure—that is, what is generally called a "bug." Failure occurrence is affected by two principal factors:

- The number of faults in the software being executed—clearly, the more bugs, the more failures may be expected, and
- The circumstances of its execution (sometimes called the "operational profile"). Some circumstances may be more exacting than others and may lead to more failures.

Software reliability is the probability of failure-free operation of a computer program for a specified time under specified circumstances. Thus, for example, a text-editor may have a reliability of 0.97 for 8 hours when employed by a secretary—but only

Verification is usually a manual process that examines descriptions of the software, while validation depends on testing the software in execution. The two processes are complementary: each is effective at detecting errors that the other will miss, and they are therefore usually employed together. Procurements for mission-critical systems often specify that an *independent* group, unconnected to the development team, should undertake the V&V activity.

There is considerable agreement that the early phases of the life-cycle are particularly important to the successful outcome of the whole process: Brooks, for example observes [23]

> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure, but they are fuzz compared with the conceptual errors in most systems.
>
> "The hardest single part of building a software system is deciding precisely what to build. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."

The more phases of the life-cycle that separate the commission and detection of an error, the more expensive it is to correct. It is usually cheap and simple to correct a coding bug caught during unit test, and it is usually equally simple and cheap to insert a missed requirement that is caught during system requirements review. But it will be ruinously expensive to correct such a missed requirement if it is not detected until the system has been coded and is undergoing integration test. Software Quality Assurance comprises a collection of techniques and guidelines that endeavor to ensure that all errors are caught, and caught early.

2.1 Software Quality Assurance

Software Quality Assurance (SQA) is concerned with the problems of ensuring and demonstrating that software (or, rather, software-intensive systems) will satisfy the needs and requirements of those who procure them. These needs and requirements may cover not only how well the software works *now*, but how well documented it is, how easy to fix if it does go wrong, how adaptable it is to new requirements, and other attributes that influence how well it will continue to satisfy the user's needs in the *future*. In the case of military procurements, a number of standards have been established to govern the practice of various facets of software development: MIL-S-52779A for software program requirements, DOD-STD-1679A and DOD-STD-2167 for software development, DOD-STD-2168 for software quality evaluation, and DOD-STD-7935 for software documentation. Similar standards exist in the civil and international sectors.

One important methodology in SQA is "Verification and Validation" (V&V). *Verification* is the process of determining whether each level of specification, and the final code itself, fully and exclusively implements the requirements of its superior specification. That is, all specifications and code must be *traceable* to a superior specification. *Validation* is the process by which delivered code is directly shown to satisfy the original user requirements.

Chapter 2

Software Engineering and Software Quality Assurance

Before describing specific quality metrics and methods, we need briefly to review the Software Engineering process, and some of the terms used in Software Quality Assurance.

One of the key concepts in modern software engineering is the system life-cycle model. Its premise is that development and implementation are carried out in several distinguishable, sequential phases, each performing unique, well-defined tasks, and requiring different skills. One of the outputs of each phase is a document that serves as the basis for evaluating the outcome of the phase, and forms a guideline for the subsequent phases. The life-cycle phases can be grouped into the following four major classes:

Specification comprising problem definition, feasibility studies, system requirements specification, software requirements specification, and preliminary design.

Development comprising detailed design, coding and unit testing, and the establishment of operating procedures.

Implementation comprising integration and test, acceptance tests, and user training.

Operation and maintenance.

There have been many refinements to this basic model: Royce's *Waterfall* model [104], for example, recognized the existence of feedback between phases and recommended that such feedback should be confined to adjacent phases.

algorithms, including protocols [32]. The use of Binary Decision Diagrams (BDDs) [24] has resulted in a significant increase in speed and, when used in combination with other recent advances, allows massive systems to be checked in reasonable amounts of time [26].

The treatment of most other topics still seems adequate—obviously more recent references would be welcome (and I have added a few where I had them available), but I am unaware of any major breakthroughs that should be brought to the reader's attention. The section on formal methods now seems dated, and recent work, as reported in the September 1990 special issue of IEEE Software, and [15] should be borne in mind. Naturally, I welcome suggestions and advice that will help me keep this document useful in the '90s.

1.2 Acknowledgments

This report comprises Part 1 of the report [105], which was sponsored the National Aeronautics and Space Administration under contract NAS1 17067. The guidance provided by our technical monitors, Kathy Abbott and Wendell Ricks of NASA Langley Research Center, was extremely valuable.

Chapter 1

Introduction

This report is concerned with the software quality and evaluation measures. We consider not only metrics that attempt to measure some aspect of software quality, but also methodologies and techniques (such as systematic testing) that attempt to improve some dimension of quality, without necessarily quantifying the extent of the improvement.

It is now widely recognized that the cost of software vastly exceeds that of the hardware it runs on—software accounts for 80% of the total computer systems budget of the Department of Defense, for example. Furthermore, as much as 60% of the software budget may be spent on maintenance. Not only does software cost a huge amount to develop and maintain, but vast economic or social assets may be dependent upon its functioning correctly. It is therefore essential to develop techniques for measuring, predicting, and controlling the costs of software development and the quality of the software produced.

1.1 Developments Since 1988

This report was originally written in early 1988 as part of a study on quality assurance techniques for AI software. The focus of the original study, and the passage of time, mean that some techniques and concerns are underrepresented here. The areas where the present treatment is most deficient seem to include the following.

CASE tools: These have become much more popular and important over the last few years. The system called SREM can now be seen as an early example of a CASE tool and the section on that system (Section 5.3.1.1 on page 43), gives something of the flavor of these systems. However, modern CASE tools deserve greater attention than is provided here.

Model Checking: Model-checking techniques have become increasingly effective and important for analyzing hardware circuits and properties of distributed

Contents

Abstract

This report comprises chapters 2 to 5 of a report prepared for NASA on the application of software quality and assurance techniques to AI software [105]. The chapters included here provide a review of software quality assurance techniques as applied to conventional software. The techniques covered include software reliability and metrics, static and dynamic testing, and formal specification and verification.

Measures and Techniques for Software Quality Assurance[1]

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

September 1991

[1]This work was performed for the National Aeronautics and Space Administration under contract NAS1 17067.