FTIC FILE COPY

Copy 20 of 36 copies

AD-A216 862

FINAL

(2)

IDA MEMORANDUM REPORT M-496

BIBLIOGRAPHY OF TESTING AND EVALUATION REFERENCE MATERIAL

Bill R. Brykczynski
Christine Youngblut

DTIC
ELECTE
JAN 25 1990
S D
D

August 1989

Prepared for
Strategic Defense Initiative Organization (SDIO)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited



INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

90 01 25 011

IDA Log No. HQ 88-33619

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, or (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Memorandum Reports

IDA Memorandum Reports are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Memorandum Reports is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This Memorandum Report is published in order to make available the material it contains for the use and convenience of interested parties. The material has not necessarily been completely evaluated and analyzed, nor subjected to formal IDA review.

Approved for public release; unlimited distribution. Unclassified.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1989	Final

4. TITLE AND SUBTITLE
Bibliography of Testing and Evaluation Reference Material

5. FUNDING NUMBERS
MDA 903 89 C 0003
T-R2-597.21

6. AUTHOR(S)
Bill R. Brykczynski, Christine Youngblut

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Memorandum Report M-496

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Strategic Defense Initiative Organization (SDIO)
Room 1E149, The Pentagon
Washington, D.C. 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT ***(Maximum 200 words)***

The purpose of IDA Memorandum Report M-496, Bibliography of Testing and Evaluation Reference Material, is to present the reference material acquired in the course of developing IDA Paper P-2132, SDS Testing and Evaluation: A Review of the State of the Art in Software Testing and Evaluation with Recommended R&D Tasks. This document comprises four sections: Section 1, Subject Index; Section 2, References; Section 3, Author Index; and Section 4, Abstracts.

14. SUBJECT TERMS
Software Testing and Evaluation (T&E); Software Validation; Software Maintainability; Software Development; Regression Teseting; Software Life Cycle.

15. NUMBER OF PAGES
390

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

IDA MEMORANDUM REPORT M-496

BIBLIOGRAPHY OF TESTING AND EVALUATION REFERENCE MATERIAL

Bill R. Brykczynski
Christine Youngblut

August 1989

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1



INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.21

August 9, 1989

PREFACE

The purpose of IDA Memorandum M-496, *Bibliography of Testing and Evaluation Reference Material*, is to present the reference material acquired in the course of developing IDA Paper P-2132, *SDS Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation With Recommended R&D Tasks*. This document was prepared for the Strategic Defense Initiative Organization (SDIO).

CONTENTS

PREFACE 1
1. SUBJECT INDEX 1
2. REFERENCES 57
3. AUTHOR INDEX 145
4. ABSTRACTS 181

1. SUBJECT INDEX

A1 Certification, *see also* **Security Verification**
structuring a system for, [Good84a]
ADAMAT, [Kell85a], [Kell85b], [Perk87]
analysis and validation of metrics, [Perk86]
AFFIRM, [Gerh80], [Thom81]
applications of,
specification/verification of communication protocols, [Suns77], [Suns82]
comparison with other techniques, [Cheh81], [Mill81b]
example of model formulation and analyzer for PSL/PSA, [Gerh84]
status and future directions, [Kemm86]
underlying formalisms, [Muss79], [Suns77], [Suns82]
ANNA: A Language for Annotating Ada Programs, [Krie80], [Luck84a], [Luck84b]
automated support for, [Luck85]
consistency checking in Ada and ANNA, [Krie83]
implementation of a subset, [Sank85]
transformation approach, [Sank85], [Sank86]
uses of,
comparative testing, [Luck85]
self-checking Ada programs, [Luck85]
semantic specification of Ada packages, [vonH85]
ASSET: A System to Select and Evaluate Tests, [Fran85a], [Fran85b]
example script, [Fran88]
ATTEST,
AID: ATTEST Interface Description Language, [Wint78]
constraint management in, [Dill81]
Abstract Data Types,
applications of,
formal verification, [Flon77]
functional testing, [Boug85a], [Boug86], [Choq86]
performance analysis, [Boot80]
programming, [Gutt75]
prototypes and implementation models, [Belk86]
to simplify modifications, [Lind76]
automated support for, see DAISTS, [Gann80]
specification (and verification) of, [Flon77]
access-right expressions for sequential constraints, [Kieb83]
algebraic techniques, [Gogu78], [Gutt77], [Gutt78b], [Zill74]
automated support for, AFFIRM, DAISTS
logic programming, [Boug86]
proof of correctness of implementations of, [Gutt78a]
axiomatic techniques,
applied to modeling specification languages, [Gerh84]
testing completeness of,
automated support for, [Jalo89]
comparison of specification formalisms, [Emde81]
evaluation of specification techniques, [Lisk75]
hierarchical specification, [Wirs83]
using state machines, [Shan82]
survey of techniques, [Shan82]

Acceptance Testing,
debugging techniques, [Laue79]
evaluating readiness for, [Bowe79]
example of,
for Ada compilers, [Amor89]
management using coverage measures, [Basi84a]
methods, [CSC78], [Cele81]
based on internal program state behavior data, [Prot88]
screening criteria for military software, [Pari76]
supported by reliability measurement, [Thom80]
Ada,
acceptance testing,
example for compilers, [Amor89]
animation of programs, [Feld89]
automated support for, Development Environments
interactive Ada, [Stan83]
programming-in-the-large, [Wolf85a]
run-time environments, problems with, [Bend89]
using DIANA trees as an internal form, [Rose84], [Rose85a]
communication protocols,
Remote Entry Call/Remote Procedure Call, [Schu81]
compiler validation, [Will89]
debugging, [Cont85]
AdaTAD, [Fain85], [Fain86], [Lind88c]
VIPS: Visual and Interactive Programming Support, [Isod87]
annotation based *see also* ANNA, TSL
capabilities needed, [Brin85]
graphical debugger, [Mora85]
graphics-oriented animation tool, [Feld89]
knowledge-based *see also* STAD,
reconstructing execution host/target, [Tayl82b]
saving traces for, [LeDo85]
symbolic debugger, [DiMa85], [Maug85], [Maye89]
definition of, [HONE80]
AVA: Annotated Verifiable Ada, [Smit88]
EEC formal definition, [Mayf86]
axiomatic semantics for exceptions, [Luck80b]
definitions of modularity and their application, [Katz87]
denotational semantics, [Mear83]
description of language using Petri Nets, [Mand85]
problems with task semantics, [Germ84]
virtual machine for, [Grov80]
experiments in, [Agre86], [Basi82b], [Basi84d], [Godf87]
impact on reliability, [Goel88]
lessons learned, [Basi85h], [Brop87]
fault-tolerance,
correspondent computing, [Lee89a]
formal verification of, [Luck80a], [Luck80b], [McGe82]
current issues, [Mayf85], [Mayf86], [Roby85]
proof system for tasks, [Barr82], [Gert84], [Mear81]
using symbolic execution, [Dill87], [Dill88a], [Dill88b], [Harr88c]

specification and proving for exceptions, [Luck80c]
interface control *see also* AdaPIC Toolset,
measurement of,
Partial Metrics System for metric-driven design, [Reyn86], [Reyn87], [Reyn89]
automated support for,
metric analysis, experience with, [Ande88]
characterization of an Ada development, [Basi85h]
metrics for, [Basi83a], [Gann83], [Gann85], [Gann86]
Cyclomatic Complexity,
ACE metric and tool, [Taus87a], [Taus87b], [Taus88]
Software Science for designs, [Szul84]
analysis and validation of, [Perk86]
automated support for *see also* ADAMAT,
classification of, [Basi84c]
example of WIS metrics, [Dela88]
structure and maintainability, [Katz86]
complexity, [Leac87]
software structure,
based on DQL: DIANA Query Language, [Byrn89]
performance evaluation,
of compilers, [Shaw89]
benchmarks for real-time system compilers, [Goel89]
of programs, [Stan83]
of run-time environments, [Bend89]
performance analyzer, [Maye89]
supported by model simulation, [Lee89b]
programmer errors, [Good86b]
quality assurance,
planning for Ada development with 2167, [Bark89]
reproducible testing, [Tai85b]
reuse,
Ada Software Repository, [Conn87]
metric analysis of, [Leac89]
Moorehouse object-oriented reuse library, [Jone89]
RLF: Reusability Library Framework project,
domain modeling, [Sold89]
hypertext for taxonomies of packages, [Lato89]
reusability analysis and measurement, [Romb88g], [Romb88h]
run-time environments,
evaluation and selection of, [Lefk89]
run-time monitoring, [Helm84b]
ART: relational translator and interpreter, [DiMa85]
ATEST: Ada Test and Analysis Tools, [Maye89]
based on TSL specifications, [Helm84a], [Helm85]
detection of errors and evasive actions, [Helm83]
transformation and monitoring, [Germ82a], [Germ82b], [Germ84], [Helm83]
statement probes, [Prob82c]
simulation,
TASKIT: Tasking Ada Simulation Kit), [Ange89]
specification languages *see also* SADMT, ANNA, TSL
ADAM: Ada-based language for multiprocessing, [Luck81]

specifying tasking using patterns of behavior, [Meld88]
test drivers,
GET Test Environment Generator, [Bess87]
TBGEN Test Bed Generator, [Pout87]
based on UATL: Universal Ada Test Language, [Zeig89]
testing and analysis of,
ASAP: Ada Static source code Analyzer Program, [Doub87]
ATEST: Ada Test and Analysis Tools, [Maye89]
data flow analysis *see also* STAD,
mutation analysis,
automated support for, [Appe88]
reference manual for mutant operators, [Bows87]
safety analysis using fault-tree analysis, [Leve83c]
automated support for, [Cha88]
standards checking,
based on DIANA intermediate form, [Byrn89]
static concurrency analysis, [Tayl83a]
automated support for, [Wamp85]
supported by Petri nets, [Mura89], [Shat88]
using TIG and TICG models, [Long88]
structural testing,
automated support for, [Basi86d], [Wu87c]
requisite support tools, [Tayl86a]
coverage monitors,
Test Coverage Monitor/Bottleneck Finder, [Pout87]
symbolic execution, [Knig85b], IOGen
CASEX: Concurrent Ada Symbolic EXecutor, [Harr88a]
symbolic testing techniques, [Clar86b]
AdaPIC Toolset, [Wolf86c]
Algebraic Program Testing, [Howd76a], [Howd78b]
comparison with other techniques, [Zeil84]
for concurrent systems, [Avru83]
probabilistic approach, [DeMi77]
Algol,
debugging tools for Algol W, [Satt72], [Satt75]
numerical algorithms testbed, [Henn78]
run-time monitoring,
Algol68 numerical algorithms testbed, [Henn76a]
Alphard,
support for formal verification, [Wulf76]
Animation of Programs,
PegaSys: Programming Environment for the Graphical Analysis of SYStems, [Mori83]
demonstration system for Ada, [Feld89]
system for algorithm animation, [Bent87]
using Smalltalk, [Lond85]
using VDM and Prolog, [Bloo86]
Arcadia, [Wolf86b]
determining requirements for persistent object capability,
PGRAPHITE model, [Wile88]
environment architecture, [Oste86b], [Tayl86b], [Tayl88]
object management, [Oste86a]

research objectives, approaches, status, [Tayl86b], [Tayl87], [Tayl88]
support for process programming, [Tayl88]
Arithmetic Fault Detection, *see also* **Perturbation Testing**
Artificial Intelligence,
expert systems,
ARROWSMITH-P for management, [Basi85g]
testing strategy for, [Hite88]
heuristics,
analysis of nature and power of, [Pear84]
characterization of methods, [Pear84]
heuristic search for error detection, [Andr81]
knowledge-based testing environment,
for kernel system calls of UNIX systems, [Pesc85]
support for debugging, [Hara83], STAD, [Shap81]
LAURA to debug student programs, [Adam80]
PROUST, [John84]
model of fault localization process, [Sedl83]
support for domain modeling, [Sold89]
Assertions,
automated support for,
input/output assertion verifier, [Siya80]
temporal assertions, [Lamp83]
uses of,
break-point assertions for debugging,
ALADDIN for assembly language, [Fair79]
error detection,
combined with heuristic search algorithms, [Andr81]
during design *see also* DACC,
formal verification *see also* Inductive Assertion [Kell76],
testing programs against formal specification, [Majo83]
verification of program execution, [Chen76], PET
Automata Theoretic, [Chow78]
Availability Estimation and Measurement,
modeling systems with hardware/software faults, [Land77]
using data from design/code inspections, [Gaff88]
Axiomatic Proof Techniques, [Chan79], [Hoar75], [Owic75], [Owic76]

BASIC,
program testing assistant, [Chan84]
Backtracking Techniques,
proving correctness from control structure abstraction, [Gerh76b]
Bibliography on, [Perr83], [Youn89b]
SEL literature, [SEL82]
automated support tools, [DeMi87a]
formal verification, [Bryk89], [Lond75], [Yeh77]
intermittent assertion, [Grie79]
invariant assertion, [Grie79]
proving correctness of programs, [Lond70]
measurement, [Bryk89]
metrics, [Cook82]
software quality, [Boeh78]

testing and analysis, [Bryk89], [DeMi87a], [Perr88], [Yeh77], [Youn89b]
mutation analysis, [Guin87]
Boundary Value Analysis, [Selb86]
Boyer-Moore Computational Logic, [Boye79], [Boye81]
comparison with other techniques, [Crai88a], [Kauf87a], [Kauf87b]
examples of, [Russ83]
FM8501 verified microprocessor, [Hunt85], [Hunt87]
Goedels' incompleteness theorem, [Shan87]
RSA Public Key Encryption Algorithm, [Boye84b]
Turing completeness of pure Lisp, [Boye83]
verified assembler, [Moor88]
verified operating system kernel, [Bevi87]
theorem prover, [Boye79], [Boye80], [Boye84a], [Boye88]
Branch Testing,
based on concept of essential branches, [Chus87]
automated support for, [Chus87]
comparison with other techniques, [Howd77c], [Ntaf81a]
problems and methods, [Chus87], [Huan75]
types of errors found and resource costs, [Gann79]

COBOL,
automated support for,
mutation analysis, CMS1 system, [Hank80]
test data generator, [Saud62]
errors, error-proneness, error diagnosis, [Lite76]
program analysis for, [Al-J82]
code-based model for predicting path faults, [Rogg80]
software science analysis of programs, [Shen80]
Cause-Effect Graphing, [Elme73]
Change Data,
applications of,
evaluation of development, [Basi82a], [Weis85c]
evaluation of requirements,
examples from A-7E requirements, [Frye81]
Chief Programmer Teams, [Bake81]
impact on quality, [Bake72a]
part of an overall development methodology, [Bake72b]
Classification of,
automated support tools, [Reif75], [Reif79b]
testing and analysis tools, [Mill77b]
cost models, [Ducl82]
data flow, [Fosd76a]
errors, [Beiz83], [JohnXX], [Ostr84], [RADC76a]
error complexity (measure of detectability), [Naka89]
formal verification methods, [Mili84]
heuristic methods, [Pear84]
measurement,
complexity measures, [Gors80]
metrics, [Basi86c]
for Ada, [Basi84c]
module cohesion, [Emer84]

productivity factors, [Vosb84]
reliability models, [Rama82]
program structure types, [Turn80]
testing and analysis techniques, [Youn89c]
dynamic analysis techniques, [Howd81c]
fault-based technques, [More88]
reliability validation techniques, [Triv80]
Cleanroom Software Development, [Dyer81b], [Dyer81c], [Dyer82c]
certifying model, [Curr83]
certifying reliability, [Dyer82b]
engineering software under statistical quality control, [Mill87a]
evaluation of, [Selb85], [Selb87b]
project management data, [Dyer81a]
software validation, [Dyer83]
statistical quality control, [Dyer85b]
statistical testing for, [Dyer82a]
Code Reading and Inspections, [Faga74], [Faga76]
advances in, [Faga86]
comparison with other techniques, [Hetz76], [Hwan81], [Selb86]
fault detection effectiveness/cost faults, [Basi85b]
evaluation of, [Selb85]
experiments in, [Cail79], [Myer78a]
applications of a probability-based model, [Jeli73]
indicators of quality inspections, [Buck81]
uses of,
estimating software availability, [Gaff88]
investigating program correctness, [Brit88]
mechanism for error reduction rates, [Faga76]
quality assurance, [Bark89]
role in the software life cycle, [Basi86b]
Cohesion,
applications of,
measuring the design process, [Crui80]
discriminant metric for classification of, [Emer84]
for generation of hierarchical system descriptions,
based on data bindings, [Selb88a], [Selb88b]
Communicating Sequential Processes, [Hoar85]
ECSP,
Concurrent Debugger, [Baia85], [DeFr85], [Late84]
static analysis of interprocess communication,
automated support for, [Baia84]
calculus for total correctness, [Hoar81]
parallel composition, [Hoar78]
proof systems for, [Apt80], [Apt83a], [Levi80], [Levi81], [Zhou81]
static analysis of, [Apt83b]
Communication Protocols, [Sari88a]
certification of, [Bart80]
design of,
computer-aided design tool for testing, [Barb88]
design validation using executable specifications,
automated support for, [Jard83]

formal methods, [Boch80]
perturbation technique for reachability analysis, [Zafi80]
automated support for, [Zafi80]
production rules to prevent errors, [Zafi80]
types of errors and their effects, [Zafi80]
for Ada,
Remote Entry Call/Remote Procedure Call, [Schu81]
security analysis,
Interrogator, [Mill87b]
specification (and verification) of, [Boch78], [Sari84b], [Sari88a], [Zhou81]
CEDAR Programming Environment, [Fern85]
Escort for integrated v&v and simplification, [Waka89]
XESAR sliding window protocol example, [Rich87b]
automated testing of, [Ural84]
state transition approach,
automated support for *see also* AFFIRM,
semi-automatic implementation of protocols, [Boch87b]
using Gypsy, [DiVi82]
using Lotos, [ISO87c], [Najm87]
using logic interpreter SLOG, [Choq85]
using symbolic execution, [Bran78]
via projections, [Lam84]
testing and analysis, [Rubi82], [Sabn85]
based on finite state machines, [Holz82], [Waka89]
automated support for, [Holz82]
dynamic analysis, [Sari88c]
based on finite state transition model,
limiting non-determinancy, [Jard83]
conformance testing for ISO-OSI protocols, ISO-OSI [Stee86]
using automaton models, [Kato86]
using checking sequences, [Heng87]
error detection with multiple observers, [Dsso85]
reachability analysis, [Zhao86]
RGA: Reachability Graph Analyzer, [Morg86], [Morg87]
test generation, [Sari88a]
based on finite state machines, [Sari82], [Sari84a], [Sari84b], [Sari87], [Sari88b]
CONTEST-FSM tool, [Forg87]
931, [Uyar86]
using T-, U-, D- and W- methods,
evaluation of fault coverage, [Sidh89]
testing methods based on formal specifications, [Dsso86]
trace analysis, [Boch88], [Sari88a]
for conformance and arbitration testing, [Boch87a]
Compiler-Based Testing,
example for a record-oriented text editor, [McMu83]
static analysis,
ECSP interprocess communication, [Baia84]
using input-output specifications, [Haml77a], [Haml77b]
Compiler Techniques, [Aho86]
for code analysis and generation, [Payt82]
optimization for asynchronous multiprocessor, [Hibb82]

smart recompilation, [Tich86]
to support symbolic debugging, [John79]
type checking for separately compiled parts, [Levy84]
Compiler Testing,
acceptance testing,
example for Ada compiler, [Amor89]
automated support for,
symbolic interpretation, [Same76]
syntax machine to generate random test cases, [Hanf70]
test generator based on grammars, [Bazz82], [Cele80], [Hous77]
performance evaluation, [Shaw89]
benchmarks for Ada real-time compilers, [Goel89]
specification and verification, [Pola81]
Compiler Verification,
for micro Gypsy, [Youn86c]
Complexity, [INFO76]
computational complexity,
Blum axioms and complexity gaps, [Boro72]
applications of, [Pipp78]
inductive inference, [Angl76]
examples of,
of specific control flow measures, [Howa85]
theory of, [Hart71], [Pipp78]
framework for research, [Rabi77]
process complexity,
of analyzing synchronization structure, [Tayl83b]
of modeling information systems, [Mart70]
of temporal logic, [Sist88]
programming complexity,
impact of programming factors, [Duns78a], [Duns78b]
measures for, [Duns77], [Duns78a], [Duns78b]
relation with problem complexity, [Wood79b]
testing complexity, [Tai80]
program complexity, [Bell74], [McTaXX]
as integral part of development, [McCl76]
contributing factors, [Gors80], [McCl78a], [Unde63]
experiment in, [Zoln76]
program structure, [Gree76], [Piwo82], [Schn77c]
results from a Delphi Survey, [Zoln81]
control of, [Dijk76b], [McCl78a], [McCl78b]
impact on,
error characteristics, [Basi82d], [Grem84]
error detectability, [Gree76]
maintainability, [Vess83], [Wake88]
programmer productivity, [Chen78a]
relationship with,
information content, [Shoo79]
psychological complexity, [Evan83b]
statistical language theory and, [Laem78]
psychological complexity, [Love77b]
of programs, [Weis74]

relationship with program complexity, [Evan83b]
resource complexity,
coordinating personnel activities, [Theb84]
statistical theory, [Shoo77b]
Complexity Measures, *see also* **Software Science, Cyclomatic Complexity** [Sull75]
a characteristic set, [Elsh84]
applications of, [Kear86]
assessing module accessibility/testability, [Moha76b]
cost estimation, [Ducl82]
automated support for,
FORTRANL static source code analyzer, [Li87]
classification of, [Gors80]
comparison of, [Li87]
major properties, [Bake80]
evaluation and validation of, [Elsh84], [Kafu85a], [Zoln76]
code and structure metrics, [Cann85]
factor analysis of dimensionality of metrics, [Muns89]
for assessing maintainability,
effectiveness of subjective/objective measures, [Gibs89]
measurement scales for characteristics, [Harr82]
framework for, [Basi80b]
predictive value, [Davi88]
for Ada *see also* Ada,
problems with, [Kear85]
relationships,
among measures, [Basi83c], [Lind89], [Muns89]
sensitivity to program structuring, [Evan83a], [Evan84c]
with development effort, [Cann85], [Lind89]
with error characteristics, [Basi83c], [Cann85], [Schn79a]
specific measures,
Chapin's measure, [Chap79]
Harrison-Magel's nesting level, [Evan84a]
Information Flow Complexity, [Henr79], [Henr81a], [Kafu88]
evaluation of, [Cann85]
Invocation Complexity, [Kafu88], [McCl78a]
evaluation of, [Cann85]
Program Analysis Complexity Model, [McCl76]
Scope Complexity Ratio, [Harr81b]
Syntactic Interconnection Model, [Wood81c]
evaluation of, [Cann85]
based on information theory, [Berl80], [Chen78a], [Shoo79]
based on nesting level, [Harr81a], [Piwo82]
localization of variables, [Rich76]
control flow measures, [Howa85]
derived metrics slope and r square, [Basi83c]
for clarity measurement, [Gord79a], [Gord79b]
for maintenance,
figure-of-merit for modification complexity, [Yau78]
knot count measure, [Blai85a]
for productivity prediction,
model of programming effort, [Wood81a]

primitive level instructions, [Kitc81]
for project management,
assessment of, [Suno82]
step count, [Suno82]
for unstructuredness, [Wood79a]
hybrid metric with context sensitivity, [Li87]
internal and external module complexity, [Lew88]
knot count measures, [Bake80]
evaluation of, [Bake79a]
program bandwidth, [Lind89]
stability measures, [Yau78], [Yau79]
evaluation of, [Cann85]
sum of the difficulty of constitutent elements, [Bern84]
system measures,
quantifying complexity of clustering partitions, [Bela81]
theoretical limitations, [Leac87]
Computational Testing,
as a debugging aid, [Clar83b]
Computational Work Measurement, [Hell72]
Concurrent Systems,
different ways of using parallelism,
axiomatic proof rules for, [Hoar75]
formal verification of,
nonassertional approach, [Lamp79a]
survey of verification techniques, [Barr85]
models of concurrency, Constrained Expressions, CSP
abstract conceptual model, [Kell76]
geometric models, [Carr82], [Cars84]
notion of synchronization structure,
complexity of analysis, [Tayl83b]
parallel-program model, [Kell76]
relation of parallel/nondeterministic, [Flon78a], [Flon81]
semantics of concurrency, nondeterminism, communication, [Fran79]
specification of,
language based on process interactions,
denotational semantics, [Kahn77]
using temporal assertions, [Lamp83]
synchronization structures,
based on rendezvous,
formalism of, [Tayl83b]
testing and analysis, Static Concurrency Analysis
IN-SYM test for synchronization errors, [Tai85c]
algebraic techniques, [Avru83]
anomaly detection, [Bris79]
automated support for, [Manc83]
combining static and dynamic analysis, [Tayl83c]
data flow analysis, [Tayl80b], [Tayl80c]
automated support for HAL/S programs, [Tayl78b]
error-based testing, [Long88]
examples from RC 4000 multiprogramming system, [Brin73]
of specifications and design, [Tai85a]

structural testing, [Tayl86a]
support for static analysis, [Tayl83b]
timing analysis,
based on path expressions, [Hsie89]
theory of testing, [Weis88a]
Constrained Expressions,
DC DYMOL translator, [Ho79]
behavior generator for, [Aver84]
design language, [Dill86]
for analysis of concurrent/distributed systems, [Dill85], [Dill88c]
automated support for, see ATTEST, [Dill81]
design analysis, [Dill84]
automated support for, [Avru86]
Constraint Logic Programming,
for specification-based testing, [Gerh88b], [Gorl87]
LEONARDO project, [Gerh88a]
Control Flow Analysis, [Carr82], [Wood77]
algorithms for, [Hech77a]
Cost Estimation,
approaches, [Jame77], [McGa84], [Nels66]
conversion cost-estimation techniques,
review and analysis of, [Hout81]
macromethodology for, [Putn78]
size estimation based on data structure metrics,
effort estimation based on metric evolution, [Wang84]
structural forecasting, [Wolv74]
comparison of techniques, [Roac80]
data requirements for, [Wolv74]
effort estimation, [Schn78], [Wals77a], [Zelk79]
back-to-front programming prediction, [Wang83]
relationship with other variables, [Basi85e]
using 1 programmer, 4 program characteristics, [Chry78]
examples of,
deep space networking, [Taus81]
from SEL resource forecasting, [Basi78a]
resource utilization curves,
Parr curve, [Basi81f]
Rayleigh curve, [Pica81]
adjustments for maintenance effort, [Wien84]
based on system structure, [Parr80]
separated as work and cost units, [Jone78]
software cost estimation study, [Herd79]
to support management, [Putn77], [Putn78], [Putn79]
Cost Models, [DACS79a]
COCOMO, evaluation and tailoring of, [Miya85]
Jensen macrolevel model, [Jens83a]
sensitivity analysis of, [Jens83b]
PRICE, [Frei79]
in a life cycle case study, [Kuhn82]
SOFCOST: Grumman's cost estimating model, [Dirc81]
WICOMOM: Wang Institute cost model, [Dems82]

avionics software support cost model, [SYSC83]
classification of,
evaluation of classes, [Ducl82]
evaluation of, [Cook80], [Thib81]
extending to include modularity factors, [Wood80a]
for fault tolerance strategies, [Scot87]
for test planning, [Brow89], [Goel81]
meta-model for resource expenditures, [Bail80]
program size estimation model, [Itak82]
reflecting complexity of personnel coordination, [Theb84]
review of, [Ducl82]
simulation models,
TSL: Total Software Life-Cycle Model, [Ducl82]
size, complexity, personnel skill, specification volatility, [Okad82]
staffing implications, [Taus82]
Coupling,
applications of,
measuring the design process, [Crui80]
for generation of hierarchical system descriptions,
based on data bindings, [Selb88a], [Selb88b]
Coverage Monitors,
Program Testing Translator, [Stuc72]
for Ada, see Ada, [Pout87]
principles and practices for, [Paig77a]
self-metric software *see also* PET
Cyclomatic Complexity, [McCa76]
adaptations of, [Bake79a], [Hans78]
applications of,
aid to testing, [McCa82a], [McCa82c], [Perr88]
complexity measurement, [Elsh78c], [Hans78]
measure of program structuredness, [McCa76]
productivity prediction, [Blai85a], [Curt79a], [Curt79b], [Curt81], [Wood81a]
program size estimation, [Gaff79]
project management, [Suno82]
support for regression analysis, [McCa82a]
comparison with other measures, [Bake80], [Gaff79], [Harr81b], [Kitc81], [Wood79a], [Wood81a]
evaluation of, [Bake79a], [Basi81g], [Evan84a], [Harr81a]
ability to provide objective measure of effort, [Kitc81]
anomalies and extension to overcome, [Myer77]
measures of comprehensibility, [Boys79]
validation of across FORTRAN programs, [Basi83b]
for Ada, see Ada, [Taus87a]
relationships,
with development effort, [Lind89]
with other measures, [Henr81a], [Lind89]

DACC: Design Assertion Consistency Checker,
cost-effectiveness of, [Boeh75a]
DACS,
glossary of software engineering terms, [DACS79b]
quantitative software models, [DACS79a]

software life cycle tools directory, [DACS85]
DAISTS: Data Abstraction, Implementation, Specification and Testing System, [Gann80], [Gann81], [Haml79], [McMu82]
evaluation of, [McMu80]
DARTS: Design Aids for Real-Time Systems, [CSDL80], [Furt81]
DAVE, [Oste75a], [Oste75b], [Oste76a]
experience with, [Fosd76a], [Oste76b]
DISSECT, [Howd77b]
advantages, limitations, and uses of, [Howd76e]
DREAM: Design Realization, Evaluation and Modeling System, [Ridd78], [Ridd79]
DDN: DREAM Design Notation, [Ridd78]
Data Based Program Testing, [Lask86]
Data Bases,
development data,
BCS software production data, [Blac77]
comparison of RADC and SEL, [Turn81a]
software engineering, [Romb87c]
Data Collection and Analysis, *see also* **SEL**
applications of,
experimental research, [Basi84b]
for experimental research, [Zelk82]
management, [Basi84b]
approaches,
SARE: Software Acquisition Resource Expenditure, [Duma83]
data requirements for,
cost estimating, [Wolv74]
reliability measurement, [Litt80b], [McCa87a], [Thay75]
examples of, [Bake77]
ASTROS measurement program, [John75]
for error data, [Fung85], [Rube75], [Thib78]
goal-directed data collection,
based on change data, [Basi81b]
four applications of, [Basi85f]
to evaluate development methodologies, [Basi82c]
methodologies for evaluating failure databases, [Duva80]
techniques, [RADC76a]
validation and analysis, [Basi80c]
Data Flow Analysis, [Carr82], [Fosd76b], [Herm76]
algorithms for, [Alle74], [Alle76], [Bart78], [Fosd76a], [Hech75], [Hech77a], [Jach84]
applications of, [Oste81a]
detection of some unexecutable paths, [Oste77]
required element testing, [Ntaf81a], [Ntaf82]
support for automatic program slicing, [Weis84]
automated support for *see also* DAVE, ASSET, FORTEST, STAD
block testing, [Lask82]
criteria,
Laski-Korel criteria, [Lask83]
Rapps-Weyuker criteria, [Rapp80]
complexity of, [Weyu84a]
comparison of, [Clar85a], [Clar86a], [Lask87]
error detection ability, [Girg86a]

selectivity of path selection criteria, [Zeil88a]
feasible criteria for nonexecutable paths, [Fran86], [Fran88]
d-tree testing, [Lask82]
data flow classification, [Fosd76a]
for concurrent systems, [Reif79c], [Tayl80b], [Tayl80c]
for recursive PL1 programs, [Rose75]
formalism for specifying diverse range of sequences, [Olen86]
using node listings, [Kenn75]
Data Space Analysis, [Paig81]
Debugging, [Brow73b], [Rust71]
and understanding, [Luke80]
architectural support for,
requirements for, [John82a]
automated support for,
applications of,
generation of program traces/profiles, [Satt75]
desirable features,
concurrent systems, [Baia85], [Webe83]
distributed systems, [Garc84]
real-time systems,
reconstructing execution host/target, [Tayl82b]
survey of, [Schw70a]
techniques for improving efficiency, [Laue79], [Satt75]
automated tools,
AIDS: Advanced Interactive Debugging System, [Hart79]
ALADDIN for assembly language, [Fair79]
ECSP Concurrent Debugger, [Baia85], [DeFr85], [Late84]
EXDAMS, [Balz69]
FORTRAN post mortem dump system, [Ng78]
Incense for displaying data structures, [Myer83]
PEBUG: Purdue Extendable Debugging System, [Blai71]
for Ada *see also* Ada,
for Algol W, [Satt72], [Satt75]
for dataflow machines, [Wahl88]
for distributed systems, [Schi81]
TAP, [Gord86], [Gord88]
based on EDL *see also* EDL,
debugging commands, [Stan80]
for real-time systems,
RED and implementation schemes for, [Hill83]
knowledge-based, [Hara83], [Shap81]
LAURA, [Adam80]
PROUST, [John84]
symbolic debugger, [Brue83], [John79]
RAIDE language independent system, [John78]
Symbolic Debug/1000, [HCP82]
source level debugger for HP-1000, [John83]
by independent persons, [Musa76]
empirical stopping rule, [Form77]
hierarchical approach, [Lask79]
models for assessing effects of process imperfections, [Down85a], [Down86]

psychological study of, [Goul72], [Goul74], [Goul75]
role of, [Schw70a]
strategies, [Laue79]
supported by,
computational and domain testing, [Clar83b]
error-sensitive testing, [Fost83]
knowledge-based model of fault localization, [Sedl83]
program slicing, [Weis84]
Decision Tables, [Pooc74]
checks for redundancy, consistency, completeness, [Pooc74]
Deductive Reasoning, [Dijk68]
Dependability Measurement, *see also* **Reliability Measurement, Availability Measurement**
Design Analysis, [Balz81], [Gerr85]
automated support for, DREAM
DECA, [Carp75]
SAMM modeling tool, [Lamb78]
TINKER: interleaving testing/design, [Lieb80]
based on formal specification, [Gutt80]
concurrent systems,
to detect synchronization errors, [Tai85a]
inspections, see Code Reading and Inspections, [Faga76]
testability analysis,
automated support for, [Yin80]
using assertions *see also* DACC,
using executable specifications, [Davi82b]
using finite state machines *see also* Automata Theoretic,
Design Evaluation, *see also* **Coupling, Cohesion**
application of Software Science, [Szul81]
application of software science,
for Ada, [Szul84]
evolution of design metrics research, [Romb88f]
measures of complexity, [Whit80]
measures of quality, [HenrXX], [Troy81]
Design Indicators, [Ross88]
System Entropy Function, [Moha79]
System Work Function, [Moha79]
automated support for, [Szul83], [Yin78], [Yin79], [Yin80]
metrics for embedded real-time designs, [Szul80]
Development Environments,
ARROWSMITH-P expert system for management, [Basi85g]
Cedar Programming Environment, [Teit84]
Hughes design analysis and testability system, [Yin80]
IPE: Incremental Programming Environment, [Medi81]
Interlisp programming environment, [Teit81]
LEONARDO project, [Conk86]
PDS 2: Process Design System 2, [Kopp76]
SOFTING Software Engineering Environment, [Snee85]
SPS: Software Productivity System, [Boeh84b]
SSAGS: Syntax and Semantics Analysis and Generation System, [Payt82]
SSES: Software Specification, Evaluation System, [Hodg76]
Toolpack, [Oste83], [Oste84]

design principles, [Tayl85], [Tayl86b], [Tayl87]
guidelines for incorporating metrics, [Selb87a]
improvement-oriented, [Basi88]
integrated tool sets, [Oste83]
sharing intermediate representations, [Lamb83]
persistent typed object management,
PGRAPHITE model, [Wile88]
tool fragment approach, [Zeil87]
user interfaces, [Youn88b]
evaluation of,
methodology for, [Weid86]
workstations, [Koer84]
for Ada, Arcadia
ARCTURUS, [Stan83], [Stan84a], [Tayl85]
Alsys tool set,
Ada program VIEWer, [Maug85]
event-driven, symbolic debugger, [Maug85]
GRAPHITE: a meta-tool for development, [Clar86c]
based on wide-spectrum languages, [Luck86a]
programming-in-the-large, [Wolf85a]
for concurrent/distributed systems,
MUST flight software production environment, [Tayl78b]
automated support under UNIX, DEMOS/MP, [Mill84]
reverse engineering,
ADDS: Automated Design Description System, [Arth88]
role in quality assurance, [Cher80a], [Cher80b]
Distributed Systems,
analysis of designs, [Avru85], Constrained Expressions
based on modified Petri nets, [Cagl82]
debugging, [Schi81]
measurement of,
guidelines and standards for,
quality measurement, [Bowe83]
model for distributed computations,
FA/C Functionally Accurate/Cooperative, [Less81]
computation-communication model,
automated support under UNIX, DEMOS/MP, [Mill84]
to support distributed termination, [Fran80]
trace analysis, [Jard87]
DoD Guidelines and Standards, [DeMi87a]
Defense System Software Quality Program, [DODS86]
process-product relationships with STD-2167, [Lave88]
Defense Systems Software Development, [DOD88]
Technical Review/Audits for Systems, Equipment, Computer Programs. [MIL85]
independent verification and validation, [AFSC88a]
management indicators, [AFSC86a]
quality,
quality assurance, [Army84], [McWe84]
specification/measurement, [AFSC86b], [McCa77a]
survey of military standards, [Bowe79]
risk abatement, [AFSC88b]

test and evaluation,
Software Test and Evaluation Manual, [DODD87]
Test and Evaluation Master Plan guidelines, [DODD86b]
Test and Evaluation, [DODD86a]
guidelines for, [Army87]
operational testing,
maintainability, [AFOT87]
management guidelines, [AFOT86]
supportability, [AFOT88a], [AFOT88b]
usability, [AFOT82]
Domain Testing, [Whit78b]
as a debugging aid, [Clar83b]
error analysis of, [Whit78a], [Zeil89]
loop analysis problems, [Whit88a], [Wisz87]
test data selection strategies and error bounds,
Clarke-Richardson, complexity of, [Hass80]
White-Cohen, [Cohe78], [Pere85], [Whit86]
complexity of testing iterated borders, [Whit88a]
evaluation and complexity of, [Hass80]

EDL: Event Definition Language, [Bate81]
BA: behavioral abstraction approach, [Bate83a]
a basis for distributed system debugging tools, [Bate82]
automated support for, [Bate83b]
EQUATE, [Zeil86]
complexity of, [Zeil88b]
ESTCA: Error Sensitive Test Case Analysis, [Fost80], [Fost85]
application to debugging, [Fost83]
sensitive test data for logical expressions, [Fost84]
Economics, [Boeh81]
of fault-tolerance, [Mign82]
of modularization, [Camp76]
of quality assurance, [Albe76]
programming cost factors, [Boeh73], [Boeh75b], [Farr65], [Putn82]
Encryption Protocols,
formal verification,
examples of,
RSA Public Key Encryption Algorithm, [Boye84b]
testing and analysis,
Inatest, [Kemm87]
Environment Characteristics,
characteristic set,
customizing to an environment, [Basi85a]
forecasting productivity, [Basi85a]
Equivalence Partitioning, [Selb86]
automated support for,
AutoParts, [Soli85]
Error-Based Testing, [Clar83a], [Ostr79], [Weyu81]
extension to real-time, concurrent systems, [Long88]
formalism for,
characterizing completeness of tests, [Howd82a]

contrasting with other approaches, [Howd82a]
theory of, [More84]
Error Seeding,
capture-recapture sampling, [Dura81b]
evaluation of seeding methods, [Knig85a]
failure characteristics of syntactic changes, [Knig85a]
issues involved, [Knig85a]
Errors,
classification of, [Beiz83], [Mend79], [Ostr84], [RADC76a]
error causes, [Boeh75a]
error complexity (measure of detectability), [Naka89]
errors occuring in real-time systems, [Ande83]
errors occuring in system programs methods, [Endr75]
for all development phases, [Bowe80]
method for, [Amor75]
problems in, [Jeli72]
review of classification schemes, [Bowe80]
taking the programmer into account, [JohnXX]
data collection and analysis needs, [Fung85], [Rube75]
examples of, [Garm81]
errors from DOS/VS operating system, [Endr75]
errors, error-proneness, diagnosis in COBOL, [Lite76]
from IV&V projects, [Fuji77]
from special-purpose editor system, [Ostr84]
experiments in,
error occurence and detection, [Hoff77]
in distributed systems,
ordering errors, [Gord85b]
influencing factors, [Feue79a], [Gerh76a]
comments, [Howd88]
complexity, [Basi82d]
language factors, [Nage84]
program structure and complexity, [Brad75], [Gree76]
programmer and problem, [Nage82], [Nage84]
reasoning errors made in software construction, [Howd89a]
statistics,
across environments, [Weis82]
algorithm implementation errors, [Bulu74]
design errors, [Boeh75a]
error types, frequencies and habitats, [Schw70a]
errors detected in development/IV&V, [Rube75]
from a testing service, [Mill79b]
persistent errors, [Glas81]
recurrent errors in real-time systems, [Goel78]
syntactical errors, [Boie72]
system program errors, [Endr75]
types, distribution, test/correction times, [Shoo75]
Evaluation Approaches,
for evaluation of,
computer models, [Prat80]
development practices, [Basi81c], [McGa82]

Reduced Form for sharing complexity data, [Harr85]
analysis of change data, [Basi81b], [Basi81e], [Basi82a], [Weis81], [Weis85c]
automatic generation of artificial systems, [Rowl88]
cluster analysis, [Chen81]
coupling evaluation with measurement, [Selb85]
effectiveness of elimination of faults, [Curr76]
error analysis, [Glas80], [Howd80b], [Weis78], [Weis82]
game theoretic testing, [Cher88]
goal-based paradigm for, [Basi82c], [Basi85c]
mutation analysis for test data adequacy, [Ntaf81a]
other, [Panz81b]
procedural approach, [Henr85]
statistical model for evaluating effectiveness, [Card87b]
environments, [Weid86]
error relationships,
analysis of change data, [Basi82d]
experimental work in software engineering, [Basi86a]
human understanding, cloze tests, [Hall86]
metrics (example from STARS program), [Gord85a]
reliability models,
replicated experiments, [Nage82], [Nage84]
stepwise statistical methodology, [Troy86]
software prototypes, [Chur86]
test data selection criteria,
RELAY model of error detection, [Rich86a]
Examples of,
compiler validation,
ACVC: Ada Compiler Validation Capability, [Will89]
measurement,
metrics applied to relational data base, [Redd84a], [Redd84b]
safe/reliable computing on Airbus/ATR Aircraft, [Roqu86]
testing a multiprogramming system, [Hans73]
testing and analysis approaches, [Mill75e], [Muno88]
PEI Testing Methodology, [Post87]
for nuclear reactor protection systems, [Geig79]
case studies, [Uren87]
testing and validation, [Ho78]
testing of the TRIDENT CC system, [Oxma78]
Exhaustive Testing, [Brow72a], [Shoo74]
Experimental Design, [Coch50]
beat the system, [Budd80a]
behavioral or psychological approaches, [Broo80a]
designing a measurement experiment, [Basi77b]
reproducible experiments, [Come79]
sampling theory and applications, [Coch53]
Extremal-Special Value (ESV) Testing,
for fault-tolerant systems, [Vouk86b]
Extremal-Special Values (ESV) Testing,
for fault-tolerant systems, [Vouk86a]

FAST: Fortran Analysis System, [Brow78]

FDM: Formal Development Methodology, [Kemm80]
Ina Jo, [Kemm80], [Sche85]
abstract machine model of a specification, [Berr87]
language reference manual, [Loca80]
testing of specifications,
Inatest, [Eckm84], [Eckm85], [Kemm85a]
example of analyzing encryption protocols, [Kemm87]
with temporal logic for concurrency properties, [Wing89]
comparison with other techniques, [Cheh81]
status and future directions, [Kemm81], [Kemm86]
theory of, [Berr87]
FORTEST, [Girg85]
experiments in error detection ability, [Girg86a]
FORTRAN,
automated support for, *see also* FAST, Mothra, FORTEST, DISSECT, DAVE, PET
Automatic Code Evaluation System, [Hall73], [Hall74], [Rama73], [Rama74a]
BRANANL for identifying basic blocks, [Fosd74]
FAVS, [GRC79]
RXVP verification system, [Mill74d], [Mill75f]
FETE: execution time estimator, [Igna71]
FORTRANL static source code analyzer, [Li87]
FORTVER documentation and error diagnosis, [Conr85]
NBS test programs, [NBS74]
SAP: Static Source Code Analyzer Program, [Deck82a]
SELFMET for self-metric instrumentation, [Urba73]
application of Software Science, [Otte76]
post mortem dump system, [Ng78]
static analyzer, [Slav75]
symbolic execution, [Clar76b], [Fava79], [Rama76]
SADAT, [Voge80]
test drivers,
test procedure language/processor, [GE77a], [GE77b], [Panz76], [Panz78a], [Panz78b], [Panz78c]
use of software probes, [Page74]
how it's used and needed compiler support, [Knut71]
impact on reliability, [Goel88]
Failure Mode and Effects Analysis (FMEA), [Bunc80], [Laws83], [Reif79a]
example for satellite, launch vehicle, reentry systems, [SAMS77]
Fault-Based Testing,
RELAY model of error detection, [Rich86b]
applications of, [Rich88]
for analysis of test data selection criteria, [Rich86a]
for testing, [Rich87a]
classification of techniques, [More88]
symbolic testing, [More87], [More88]
theory of, [More87], [More88]
Fault-Tolerance, [Aviz78]
and fault-intolerance, [Aviz75]
as a basis for system structuring, [Rand75]
automated support for, [Wild87]
evaluation of technology, [Ande85], [Sliv84]
examples of,

air traffic control system, [Aviz87]
experiments in, [Dunh85]
with the SIFT operating system, [Brun85]
for dataflow machines, [Srin85]
principles and practices, [Ande81]
queuing analysis of, [Nico87]
real-time systems, [Ande83]
relationship of fault tolerance/elimination techniques, [Shim88]
reliability evaluation,
based on directed acyclic graphs, [Sahn87]
strategies, *see also* Recovery Blocks, N-Version Software
comparison of, [Grna80a], [Scot84b], [Scot87]
correspondent computing,
implementation for Ada, [Lee89a]
cost model for, [Mign82], [Scot87]
detector redundant scheme, [Han76]
modeling of, [Grna80b]
redundant data structures, [Blac81], [Tayl80a]
repetitive run modeling for failure/fault estimation, [Dunh86]
Fault-Tree Analysis, [Harv82], [Leve83b], [McIn83], [Vese81]
automated support for, [Rola86], [Stol84]
for Ada, [Leve83c]
for both hardware and software, [Han76]
Finite State Machines,
applications of, Communication Protocols
interpretation correctness, [Ferr77]
model environment for validation, [Hend75]
requirements modeling for testability, [Chan85]
specifying/verifying data abstractions, [Shan82]
testing correctness of control structures, [Chow78]
estimates of software size from, [Brit82]
state machine specification technique, [Prin78]
Flavor Analysis, [Howd87], [Howd89a]
Flow Expressions,
for specification of concurrent systems, [Shaw78]
Formal Verification (Hardware),
examples of,
FM8501 verified microprocessor, [Hunt85], [Hunt87]
reusable library, [Bevi88]
test vector generation, [Vose88]
Formal Verification, *see also* **Invariant Assertion, Intermittent Assertion, Induction**
automated support for,
modification of first-order rules for algebraic expresssions, [Sark89]
practical problems, [Boye84a], [Luck77]
state of the art, [Crai86], [Crai87a]
survey of,
mechanical support for formal reasoning, [Lind88d]
theorem provers, [Elsp72a]
automated tools *see also* m-EVES, AFFIRM, Gypsy, HDM, Boyer-Moore Computational Logic, Boyer-Moore Computational Logic, FDM, Stanford Pascal Verifier
interactive program verification system, [Deut73]

logical basis and implementation, [Igar73]
program verifier, [King69], [King70]
concurrent/distributed systems *see also* Temporal Logic,
EBS: Event-Based Specification Language,
comparison of,
EBS with temporal logic and trace approaches, [Chen83]
Misra-Chandy's proof method, [Misr81]
nonassertional approach, [Lamp79a]
problems in, [Grie77]
proving total correctness, [Flon78b], [Flon81], [Misr82]
proving weak correctness, [Flon78a]
examples of,
Byzantine Generals problem, [Lamp82]
Dijkstra's garbage collector, [Grie77]
for a reactor protection system, [Ehre73]
proof of a calendar program, [Lamp79b]
proof of a program: FIND, [Hoar71b]
proof of computer interval arithmetic, [Good70]
for Ada, *see also* Ada
impact of language design, [Wulf76]
interpretation correctness, [Ferr77]
introduction to, [Berg82], [Grie76], [Lond75]
methods,
classification of, [Mili84]
constructive approach, [Good75e], [Hoar72], [Wegb77]
in support of transformational programming, [Krie86]
heuristic approach, [Katz73]
stacking approaches, [Hunt87], [Moor88]
survey of theory and techniques, [Elsp72a]
of compilers,
for micro Gypsy, [Moor88]
of structured programs, [Ling79]
principles of, [Good79b]
proofs, completeness, transcendentals and sampling, [Davi77]
state of the art, [Kemm86], [Oste80]
prospects for, [Dahl78], [DeMi79a], [Fetz88]
supported by,
abstract data types, [Flon77]
automated support for, [Gutt78a]
control structure abstraction, [Gerh76b]
program traces, [Howd78c]
state machines for interpretation, program, implementation correctness, [Ferr77]
symbolic execution, [Burs74], [Dill87]
for communication protocols, [Bran78]
limitations, (dis)advantages of methods, [Dill88a]
Function Point Analysis,
applications of, [Symo88]
productivity measurement, [Albr79], [Albr81]
with a productivity index, [Behr83]
comparison with other measures, [Albr83]
estimating handbook, [Zwan84]

partial alternative method, [Symo88]
review of metric derivation/calibration, [Vern89]
Functional Analysis, [Howd87]
data type transformation analysis, [Howd86]
functional trace analysis, [Howd86]
operator sequence analysis, [Howd86]
Functional Testing, [Elme71], [Howd81a], [Howd86], [Howd87]
applications of,
module and integration testing, [Howd85]
security testing, [Glig87]
based on,
Basic User Perceived functions, [HennXX]
algebraic data type specifications, [Boug86], [Choq86]
Prolog interpreter, [Boug86], [Choq86]
design abstractions, [Howd80a]
formal specification, [Lask88a]
backtracking issues, [Mill75c]
category-partition method,
automated support for, [Ostr86], [Ostr88]
comparison with other techniques, [Basi85b], [Howd80c], [Hwan81], [Selb86]
reliability of, [Howd80c]
relationship of test data to operational usage, [Basi84a]
test control process for, [Elme69]

General,
analysis of validation techniques for scientific programs, [Howd79], [Howd80b]
basic text on testing, [Myer79]
effectiveness of static, dynamic techniques, [Howd80b]
formal methods,
prospects for, [Levi78]
formal program testing, [Cart81]
issues of liability, [Joyc87a]
number of tests necessary to verify a program, [Shoo79]
problems in large-scale system development, [Broo75]
program test methods, [Hetz73]
software validation, [Carr80]
testing for an individual programmer with limited resources, [Bran80]
why does software die, [Brow80a]
Glossary for,
debugging, [John82b]
software engineering, [Babs83], [DACS79b], [IEEE83a]
software tools and techniques, [Reif79b]
Grammars,
attribute grammars,
for test data generation, [Dunc78], [Dunc81]
relating logic programs with, [Dera85]
context-free grammars,
for test data generation,
Mockingbird, [Gorl87]
for compiler testing, [Bazz82]
for testing parsers/debugging grammars, [Purd72]

for test plan generation, [Baue79a]
formal grammars,
for compiler testing,
example of verification, [Hous77]
Graph Theory, [Cant89], [Stig74]
Dilworth's theorem for acyclic digraphs, [Ntaf79], [Ntaf81b]
algorithms for,
available expressions at entrance, [Ullm73]
path building, complexity of, [Gabo76]
applications of,
control flow analysis,
graph-theoretic constructs for, [Alle71]
data flow analysis, [Hech75]
design simulation, [Schn77b]
partitioning to highlight element relationships, [Paig75]
performance and reliability analysis, [Sahn87]
predicting execution behavior, [Olde83]
program design and debugging, [Schn79b]
testing, [Beiz83], [Fosd76a], [Jach84], [Paig72], [Paig78a], [Stic78]
path cover problems, [Ntaf81b]
reducible flow graphs, [Hech72]
review of partitioning methods, [Paig77b]
Gypsy Verification Environment, [Good75c], [Good84b]
Gypsy language, [Ambl76a], [Ambl76b]
verified compiler for micro Gypsy, [Youn86c]
comparison with other techniques, [Cheh81], [Crai88a], [Kauf87a], [Kauf87b]
examples of,
message flow modulator, [Good82b]
proof of a distributed system, [Good82a]
verification of communication protocols, [DiVi82]
verification of security kernels, [EPI82]
status and future directions, [Good86a], [Kemm86]
symbolic execution of concurrent systems, [Eckm83a]

HDM: Hierarchical Development Methodology, [Elsp72b], [Elsp73], [Elsp74], [Robi79], [Silv79]
EHDM, [Crow85a]
specification language, [Crow85b]
Muse to enhance HDM for A1 certification, [Halp87]
SPECIAL: SPECIfication and Assertion Language, [Robi77], [Roub77], [Rush84]
comparison with other techniques, [Cheh81], [Gogu80], [Mill81b]
examples of,
verification of the Provably Secure Operating System, [Neum75]
verification of the SIFT operating system, [Gold80], [Mell82], [Stan84b]
status and future directions, [Kemm86]
Hoare's Logic,
generalized to concurrent programs,
relation to Pnueli's temporal logic formalism, [Lamp80]
the Decomposition Principle meta-rule, [Lamp84]
survey of results of application, [Apt81]
Human Factors,
behavioral analysis of programming,

frequency of syntactical errors, [Boie72]
cognitive psychology,
and Software Science, [Coul83]
cognitive science of programming, [Curt83]
display techniques to facilite comprehension, [Vemu80]
problem solving capabilities/performance, [Grif72], [Love77a]
theory of the learnable, [Vali84]
types of programming knowledge, [Solo84]
experiments in, [Basi79b]
effects of modern coding practices, [Shep79]
for developing quality software, [Shne77c]
impact of degree of discipline, [Basi78b]
impact of flowcharts, [Shne77a]
impact of specification symbology/spatial arrangement, [Shep81]
influences on understanding, [Shep77]
methods for, cloze tests, [Hall86]
program comprehension, [Boys79]
psychological study of debugging, [Goul72], [Goul74], [Goul75]
team design, [Basi78b], [Reit79]
factors in team programming, [Theb83]
fault tolerance,
influence of programmer profiles on coincident errors, [Vouk85b]
human errors in programming, [Youn74]
mental effort related to program clarity, a measure of, [Gord77]
program structure,
impact on program understanding, [Love77b], [Miar83]
psychological complexity,
of maintenance tasks, [Curt79a], [Curt79b], [Curt81]
relationship with software complexity, [Evan83b]
psychology of programming, [Shne80], [Wein71]
review of research, [Shei81]

IEEE Guidelines and Standards,
configuration management, [IEEE83c]
measures to produce reliable software, [IEEE87]
quality assurance plans, [IEEE84]
software engineering terms, [IEEE83a]
software quality metrics methodology, [IEEE88]
test documentation, [IEEE83b]
IOGen, [Jenk86], [Lind85], [Lind88a], [Lind88b], [Lind88c]
TESTgen, [Coff87]
typing mechanism for CAIS, [Lind87]
ISO-OSI,
conformance testing methodology/framework,
abstract test suite specification, [ISO87b]
general concepts, [ISO87a]
Incremental Analysis,
automated support for *see also* PIC, AdaPIC Toolset
for logic programming,
GCLP: Generic Constraint Logic Programming, [Wild88]
sources of incompleteness, [Wild88]

Independent Verification and Validation, [JLC84]
DoD guidelines and standards, [AFSC88a]
evaluation of methodology for flight dynamics, [Page85]
for certification of minimum testing criteria, [Sork79]
planning and conduct, [Fuji77]
practical experience with, [Page84]
role of independent validation agency, [Agil76]
Induction, [Ande79a]
computational induction,
subgoal induction, [Morr77]
generator induction for data structures, [Wegb76]
inductive assertion, [Gall81], [Hant76], [Kell76], [Lamp77], [Lond75]
automated support for *see also* HDM, Gypsy Verification Environment
inductive inference, [Angl80], [Angl83], [Blum75]
computational complexity of, [Angl76]
to support investigation of program testing, [Cher86]
proofs of equational theories with constructors, [Huet80]
Information Flow Analysis, [Carr82]
Instrumentation, [Prob80]
SELFMET for self-metric FORTRAN instrumentation, [Urba73]
applications of, Coverage Monitors, Run-Time Monitoring
collecting program attribute values, [Huan78]
detection of data flow anomalies, [Huan79]
profile keeping, [Knut71]
optimal measurements for frequency counts, [Knut73]
distributed environments,
METRIC: a kernel instrumentation system, [McDa77]
software probes,
for testing FORTRAN, [Page74]
optimal placement of monitors, [Rama75b]
statement contrasted with branch probes, [Prob82c]
Integrated Application of Techniques,
automated support for *see also* Toolpack, TEAM
benefits of,
achieved in the PIMS Trending Project, [Post87]
experiments in, [Selb86]
formal verification, testing, analysis, [Oste80], Partition Analysis
investigative approaches,
state-space exploration, [Youn89c]
measurement, [Kafu81]
and documentation, [Schr84]
proofs, analytic models, testing, [Triv80]
testing and analysis, [Clar82], [Howd82b], [Oste81b], [Oste84]
combining static and dynamic analysis, [Tayl83c]
with symbolic execution and formal verification, [Oste80]
fault-based techniques, [Youn88a]
functional and structural testing, [Clar78a]
functional testing, [Howd85], [Howd87]
mutation and perturbation testing *see also* EQUATE,
test generation with design, [Lask88a]
Integration Testing,

auditing of SOFTING, [Snee85]
genesis of discrepancies, [Jeli72]
military standards and metrics, [Bowe79]
white box approach, [Hale82]
Interface Analysis,
automated support for *see also* IOGen, SADMT, PIC, AdaPIC Toolset
comparison with other techniques, [Howd77c]
for program structuring, [Trio86]
interface control,
formal model for, [Wolf85b], [Wolf86a]
modeling stabilization of a large system, [Hane72]
Interface Specifications,
input-output specifications, [Haml77a]
organization for specifying abstract interfaces, [Clem84]
Intermittent Assertion,
correctness of continually operating programs, [Mann78]
total correctness, [Grie79]
translating other proofs into, [Grie79]
validity of program transformations, [Mann78]

LISP,
Metric for analysis of program performance, [Wegb75]
formal verification,
Turing completeness of pure Lisp, [Boye83]
test data for proving LISP programs, [Budd78c]
testing and analysis,
mutation analysis, [Budd80a]
Language Design,
approaches for improved testing/analysis, [Kosy73]
experiments in,
design principles to promote reliability, [Gann75]
effects of high- and low-level languages, [Bish86]
graphical vs textual design languages, [HenrXX]
impact of static typing and typeless, [Gann76], [Gann77]
language features, stylistic/design techniques, [Shne75]
nonprocedural languages and productivity, [Hare82]
requirements for,
exception handling, [Good75d]
formal verification, [DeMi79a], [Wulf76]
module interconnection languages, [DeRe76]
powerful checking by compilers, [Will79]
Language Specification,
denotational semantics: Scott-Strachey approach, [Stoy77]
dynamic grammar, [Hanf70]
semantics of programming languages,
expressed in SEMANOL(73), [Ande76b]
using Petri nets (for Ada tasking), [Mand85]
Lines of Code (LOC),
applications of,
predicting productivity, [Curt79a], [Curt79b], [Curt81], [Hals77d], [Wood81a]
comparison with other measures, [Albr83], [Wood81a]

Loop Analysis,
automated support for, [Wate79]
by solving first order recurrence relations, [Chea78]
determining provability/unprovability, [Meye67]
heuristic/extraction for predicate synthesis, [Wegb74]

MAP, [Warr82]
using static analysis to support debugging, [Tisc83]
Machine Architectures,
dataflow machines,
distributed debugging methodology/simulator, [Wahl88]
simulated program execution, [Land79]
software development tools for, [Jarr84]
support for fault tolerance, dataflow graphs, [Srin85]
vector processors,
for mutation analysis, [Gali87a], [Gali87b], [Gree87], [Krau86], [Ligo87], [Math86]
mutant unification, [Math88a], [Math88b], [RegoXX]
transformation techniques, [Math87a], [Math87b]
unified scheduling of mutants, [Krau88]
Maintainability,
definitions of, [Gelp79], [Gilb79]
experiments in,
effectiveness of subjective/objective measures, [Gibs89]
relationship with system structure, [Gibs89]
influencing factors, [Grad87a], [Grem84], [Lohs84], [Romb87a], [Romb87b], [Shep78], [Vess83], [vanH68]
measurement of, [Feue79a], [Romb89a]
a case study, [Blai85b]
figure-of-merit for modification complexity, [Yau78]
stability and modifiability, [Romb87a], [Romb87b]
characteristic metric set, [Romb84]
stability measure, [Yau79]
using complexity metrics, [Bern84], [Harr82], [Wake88]
using quality metrics, [Henr88a], [Kafu85b]
via questionnaires, [AFOT87]
ripple analysis, [Hane72], [Hsie82], [Yau78]
testing of, [Gelp79]
Mathematical Foundations,
boolean algebra, [Beiz83]
fallibility in mathematics and programming, [Gerh76a]
for structured programming, [Mill72b]
formal notations for design, [Hoar87]
mathematical theory of computation, [Mann74]
predicate calculus,
notions of extension and equivalence, [Gall81]
regression analysis,
analysis of variance and regression, [Dunn74]
multiple linear regression, [Drap66]
sampling theory, [Haml87]
problems in, [Haml86]
statistical theory,
in information content and complexity, [Shoo77b]

stochastic processes, [Cinl75]
Measurement and Evaluation Systems,
AMS: Automated Measurement System, [Sief88]
Mentor for measurement/documentation, [Schr84]
SMDC: Software Metrics Data Collection System, [Yu88a]
for Ada, *see also* TAME, ADAMAT
Metrics, [Gilb76] *see also* **Quality Measures**
applications of,
identifying error-prone software,
decision tree framework, [Selb87c]
review of process, product measures, [Shen85]
performance evaluation, [Lync81]
reliability measurement of military systems, [Koss88]
size and effort estimation, [Wang84]
software development management, [Gaff81a]
support for allocation of resources, [Shen85]
team design, [Theb83]
candidate top 10 list of metric relationships, [Boeh87]
characteristics set of cost/quality metrics, [Selb85]
classification of, [Basi86c]
critical issues, [Ejio87]
frameworks for,
decision trees, [Selb87c], [Selb89]
introduction and overview, [Cook82], [Duns83]
metrics and models, [Cont86]
selection of,
supported by measures of yield and coverage, [Kafu85a]
units of measure, [Jone78]
alternative to lines of code,
based on Deviation-values (D-values), [Miya87]
validation of, [Kafu88], [Perk86]
difficulties in evolving and validating, [Gaff81a]
framework for evaluation of,
example from the DoD STARS program, [Gord85a]
Modal Logic,
relation of Manna's and Floyd's techniques, [Burs74]
Mothra, [DeMi86b], [DeMi87d], [DeMi88a]
design principles, [DeMi87b]
functional capabilities, [DeMi86a]
interpreter requirements, [Offu87]
testing of Mothra, [BowsXX]
thematic tools for testing, [DeMi87b]
user manual, [Guin87], [SERC87]
Multiple Domain Test Coverage, [Redw83]
Mutation Analysis, [Acre79], [Budd78a], [DeMi79b], [DeMi87b]
a measure of test data adequacy,
used in comparison of testing techniques, [Ntaf81a]
applied to,
Ada *see also* Ada,
LISP, [Budd80a]
decision table programs, [Budd78b], [Budd80a]

numerical software, [Henn81]
automated support for, Mothra
Ada and FORTRAN, [Appe88]
CMS1 for COBOL, [Hank80]
EXPER for FORTRAN, [Budd80a], [Budd80c]
portable mutation testing suite, [Budd83a]
users guide to the pilot mutation system, [Budd77]
using vector processors, [Gali87b], [Gree87], [Krau86], [Ligo87], [Math86]
mutant unification, [Math88a], [Math88b], [RegoXX]
transformation techniques, [Math87a], [Math87b]
unified scheduling of mutants, [Krau88]
vectorization over multiple data sets, [Gali87a]
competent programmer hypothesis,
formal analysis of, [Gour81], [Gour83]
theoretical and empirical studies, [Budd80a]
constraint based test data generation, [DeMi87c]
coupling effect hypothesis, [DeMi78]
theoretical and empirical studies, [Budd80a], [Budd80b]
different forms of,
firm mutation analysis, [Wood88]
see Weak Mutation Analysis, [Girg85]
see also Specification Mutation *see also* Specification Mutation,
syntax directed/semantics aided, [Wu87a], [Wu87b], [Wu88]
effectiveness of, [Acre80]
integrated with perturbation testing,
automated support for *see also* EQUATE
problems and solutions, [Budd81], [Ridd80]
determining dead or alive, [Wood88]
determining equivalence, [Budd80b]
heuristics for, [Bald79]
generation of mutation-adequate test data, [DeMi88a]
stability of test data, [Burn78]
state of the art, [Lipt78]

N-Version Software, [Chen78b]
advantages and limitations, [Aviz84], [Bish86]
applications of,
for tolerance of design faults, [Aviz85]
software testing, [Bril87], [Shim88]
automated support for,
DEDIX distributed supervisor/testbed, [Aviz85]
coincident errors, [Eckh85]
computing reference/observed distribution, [Vouk85c]
evaluation of assumption of independence, [Knig86a]
influence of programmer profiles, [Vouk85b]
testing for version independence, [StJe85]
experiments in, [Aviz77], [Gmei79], [Knig84]
analysis of faults in, [Bril84]
failure probabilities, [Knig86b]
specification of, [Kell82], [Kell83]
reliability evaluation, [Dunh86]

Markov model for, [Sone80]
data domain model for, [Scot83a], [Scot83b], [Scot87]
validation of, [Scot84a], [Scot84b]
testing and analysis,
back-to-back testing, [Bish86]
based on use of VDM and Prolog, [Bloo86]
extremal-special value testing, [Vouk86b]
extremal-special values testing, [Vouk86a]
random testing, [Vouk86a], [Vouk86b]
effectiveness of, [Vouk85a], [Vouk85c]
structural testing, [Vouk86a], [Vouk86b]
theoretical basis for study of redundant software,
choice of "n", [Eckh85]
effectiveness of, [Eckh85]

Operating Systems,
FTOS: Fault Tolerant Operating Systems, [Sone81]
performance evaluation of, [Rums77]
security verification, *see also* FDM, Gypsy, AFFIRM
comparison of techniques, [Cheh81]
specification for, [Kore84]
requirements for, [Step74]
structure of,
evaluation of based on information flow, [Henr79]
modularity considerations, [Schn77c]
monitors as a structuring method, [Hoar74]
test control process for functional testing of, [Elme69]
Operational Usage Profiles,
measures of testing representativeness,
estimator for operational usage reliability, [Brow75]
representativeness of functional test data, [Basi84a]
sampling theory,
problems in, [Haml86], [Haml87]
specification of, [Brow75]
test cases to cover entire input domain, [Nels78]
Oracles,
based on specifications, [Ande76b]
T-3 Testing Tool, [Lawr87]
using algebraic axioms *see also* DAISTS,
pseudo oracles, [Davi81]
the oracle assumption, [Weyu80a]
reasonableness of and consequences, [Weyu82]

PACE: Product Assurance Confidence Evaluator,
FLOW, part of the PACE system, [Brow72a]
PET, [Stuc73], [Stuc74], [Stuc75a], [Stuc77]
PIC: Precise Interface Control, [Wolf85b], [Wolf85c], [Wolf86a]
for Ada *see also* AdaPIC Toolset
PL1,
automated support for,
EFFIGY for symbolic execution, [King75a], [King76]

data flow analysis, [Rose75]
path testing, [Bagg80]
numerical profile,
using Software Science, [Elsh76a], [Elsh76b], [Zweb79]
PSL PSA, [Teic74], [Teic77]
Partial Evaluation,
applications of, [Beck76]
interpretive and compiling methods, [Beck76]
Partition Analysis, [Clar84], [Rich81a], [Rich81c], [Rich85b]
effectiveness of, [Rich82]
examples of application, [Rich81b]
specifications for, [Rich81d]
Pascal,
automated support for, [Kern81]
GRAPHTRACE interactive trace of heap, [Getz83]
Pascal validation suite, [Wich79]
data flow analysis *see also* ASSET,
formal verification *see also* Stanford Pascal Verifier,
knowledge-based debugging,
PROUST, [John84]
symbolic execution,
UNISEX: a Unix-based executor, [Eckm83b], [Kemm85b], [Soli83]
with path expressions, [Camp79]
Path Analysis,
constrained path problems, [Ntaf79]
effectiveness of,
for testing predicates, [Zeil81b]
finding minimum path cover, [Ntaf79], [Ntaf81b]
solving nonlinear inequalities, [ElspXX]
unexecutable paths,
allegations to avoid unfeasibility problems, [Wood80b]
detection supported by data flow analysis,
heuristics for detecting some classes of, [Oste77]
Path Expressions,
applications of,
specification of process synchronization, [Camp74], [Camp79]
for timing analysis, [Hsie89]
path rules variant for debugging, [Brue83]
Path Testing, [Beiz83]
automated support for,
PL1 programs, [Bagg80]
test drivers, [Shoo79]
comparison with other techniques, [Dura81a]
generation of test data, [Howd75a]
path prefix testing strategy, [Prat87]
reliability of, [Howd76c], [Pimo75]
Performance Evaluation,
a metrics success story, [Lync81]
applications of,
as a design tool, [GilkXX]
modeling for program optimization, [Shol75]

automated support for,
performance modeling, [Ches77]
basic quantities,
computing based on queuing network models, [Denn78]
definitions of, [Denn78]
operational relationships between, [Denn78]
determining upper bound on running time, [Meye67]
for operating systems, [Rums77]
issues faced and alternative techniques, [Warn72]
of Ada programs, [Lee89b], [Stan83]
of compilers,
discrimination rate, [Shaw89]
relation between compile time and modularity, [Shaw89]
test generator, [Bazz82]
of concurrent systems,
based on directed acyclic graphs, [Sahn87]
of real-time system programs, [Ginz65]
response times of level structured systems, [Hart84]
state transition balance, one-step behavior and homogeneity concepts, [Denn78]
supported by,
abstract data types with performance data, [Boot80]
closed-form expressions, [Wegb75]
Metric for LISP, [Wegb75]
formal methods, [Levi78]
model simulation, [Lee89b]
operational analysis,
quantifying errors in assumptions, [Beng87]
petri net and queuing network models, [Chan89]
state model of computation, probabilistic grammar-based input, [GilkXX]
timed Petri nets, [Razo85]
Perturbation Testing, [Zeil81a], [Zeil81b], [Zeil83a]
comparison with other techniques, [Zeil84]
for computation errors, [Zeil84]
for domain errors, [Zeil83b], [Zeil89]
integrated with mutation analysis,
automated support for *see also* EQUATE,
Petri Nets, [Pete77], [Pete81]
applications of,
analysis of concurrent/distributed systems, [Cagl82], Static Concurrency Analysis
performance analysis, [Razo85]
analysis of real-time systems,
performance analysis,
automated support for, [Chan89]
safety, recoverability, fault-tolerance, [Leve87]
language description (for Ada tasking), [Mand85]
static concurrency analysis,
for Ada, see Ada, [Mura89], [Shat88]
Process Programming, [Oste87]
applications of, [Romb88c]
generating information bases, [Romb88a], [Romb89b]
automated support for *see also* Arcadia

based on software development graphs, [Bjor87]
specification language for, [Romb88a], [Romb88c], [Romb88d]
Productivity,
estimation, [Wals77a]
using complexity metrics, [Curt79a], [Curt79b], [Curt81]
influencing factors, [Chry78], [Lawr81], [Vosb84]
classification of, [Vosb84]
complexity, [Chen78a]
human, [Grif72], [Love77a]
saturation in team-oriented development, [Theb83]
language design, [Bish86], [Hare82]
programming/organizational, [Card87b], [Duns80], [Jeff85], [Sack68]
issues of the 80's, [Jone81]
limits to, [Jone79]
Program Slicing,
methods for, properties and applications of, [Weis84]
Program Structure,
classification of structure types, [Turn80]
consideration for,
ease of error detection, using simulation, [Schn77b]
error detection and recovery, [Horn74]
reliability prediction, [Shoo76]
impact on,
complexity measures, [Evan83a], [Evan84c]
error detectability, [Gree76]
understanding, [Love77b], [Wood81b]
measures for, [Gide74]
stability measure, [Soon77], [Yau79]
properties of "good" structure, [Chan73]
Program Traces,
applications of,
debugging Ada programs,
trace database model, [LeDo85]
proving properties of programs, [Howd78c]
specification, [MacL82]
automated support for,
selective trace using frequency counts, [Satt72], [Satt75]
symbolic traces, [Howd78c]
value traces, [Howd78c]

Quality,
correlation with testing effort, [Tuck65]
data sheets, [Besh85]
factors in, [Press83], [Walt79]
software quality framework, [Boeh78], [Bowe85], [Cava78]
impact of,
structured programming, [Bake72a]
team design, [Reit79]
chief programmer teams, [Bake72a]
user's view, [Davi85]
Quality Assurance, [Dunn82]

automated support for, [Brow73a]
experience with, [Ande88]
management and development tools, [Cher80b]
role of programming environments, [Cher80a], [Cher80b]
based on inspections, [Bark89]
economics of,
impact of approaches on quality and cost, [Albe76]
examples of, [Krac78]
planning for Ada development with 2167, [Bark89]
guidelines and standards, [Press83]
Computer Society standard for SQA plans, [Buck79]
Defense System Software Quality Program, [DODS86]
DoD guidelines and standards, [Army84]
DoD quality indicators, [AFSC86b]
IEEE software quality metrics methodology, [IEEE88]
IEEE standard for quality assurance plans, [IEEE84]
RADC measurement manual, [McCa80a], [McCa80b]
distributed systems, [Bowe83]
engineering handbook, [McWe84]
example from AT&T Bell Laboratories, [Ingl86]
example of telecommunication requirements, [Eric85]
for embedded real-time designs, [Szul80]
for flight dynamics software, [Perr87]
handbook for specification/measurement, [McCa77a]
industry and government requirements, [Land86]
measures to produce reliable software, [IEEE87]
metrics standard (concept of), [Sing86]
operational testing,
maintainability, [AFOT87]
supportability, [AFOT88a], [AFOT88b]
usability, [AFOT82]
quality specification and evaluation, [Bowe85]
survey of military standards and metrics, [Bowe79]
human incentives, [Mizu83]
in a quality management program, [McCa79], [Walt79]
practices, [Brya80], [Ligh76]
statistical quality control, [Gran72]
Quality Measures, [Gaff81b] *see also* **Design Evaluation**
Procedural Approach to the Evaluation of Software,
Design and Management Indicators, [Ross88]
applications of,
cost estimation, [Ducl82]
based on pattern recognition methods, [McGi77]
for distributed systems, [Bowe83]
metrics for, [Evan87]
anomaly detection, prediction, acceptance, [McCa80a]
based on interconnectivity, [Kafu81]
evaluation and prediction, [McCa77b]
products and process, STARS metrics, [Szul84]
testability and testedness, [Moha76a], [Moha76b]
user satisfaction,

automated support for, [Bail83]
review of, [Ives83]
models and metrics for management/engineering, [Basi80a]
prediction formulae,
automated support for, [Amst76]
utility of, [McCa78]
validation of, [Kafu85a]
Queueing Analysis,
applications of,
performance analysis, [Denn78]
automated support for, [Chan89]
for fault-tolerant systems, [Nico87]

Random Testing,
comparison with other techniques, [Dura81a], [Haml88], [Ntaf81a]
for fault-tolerant systems, [Vouk86a], [Vouk86b]
effectiveness of, [Vouk85a], [Vouk85c], [Vouk86a]
Recovery Blocks, [Hech76]
automated support for, [Ande76a]
consensus recovery block method,
reliability evaluation,
data domain model for, [Scot83a], [Scot83b], [Scot87]
validation of, [Scot84a], [Scot84b]
for concurrent systems,
sufficient conditions for limiting rollback, [Kant80]
performance of, [Wels83]
data domain model for,
validation of, [Scot84b]
reliability evaluation,
data domain model for, [Scot83a], [Scot83b], [Scot87]
validation of, [Scot84a]
reliability model for, [Hech76], [Hech79]
techniques for constructing acceptance tests, [Hech79]
Regression Analysis, [Lee88], [Leun88]
0-1 integer programming, [Fisc77]
alternative retest philosophies, [Fisc77]
automated support requirements, [Panz78a]
based on Cyclomatic Complexity, [McCa82a]
data structure for storing information, [Leun88]
measure of tests affected by instruction change, [Leun88]
test selection, [Cox81]
Reliability, [Bend86], [Jeli72]
concepts and concerns, [Rose85b]
definitions of, [Jeli72], [Musa79b], [Shoo77a], [Weis85b], [Weis88b]
designing/implementing a reliability program, [Rose85b]
influencing factors, [Jeli72]
Ada and FORTRAN, [Goel88]
development practices, [Card87b]
effects of field service on multisite software, [Bake88]
language design, [Bish86], [Gann75], [Gann76], [Gann77]
investigative approaches,

stepwise statistical methodology, [Troy86]
issues in software engineering, [Down85b]
principles and practices, [Mora78a], [Myer76], [Myer78b]
relationship to hardware reliability, [Piku76]
study of radar system software reliability, [Bowe78]
theory of,
MTSR: Mathematical Theory of Software Reliability, [RADC76a]
critique of, [Haml78b]
user's view, [Davi85]
Reliability Measurement, [Musa80a], [Musa87], [Thay78]
Software Reliability Measurement Framework, [McCa87a], [McCa87b]
reliability and estimation studies, [Goel82]
applications of, [Musa80b], [Musa87], [Shoo77c]
acceptance testing, [Thom80]
determine continuation/termination of testing, [Thom80]
supporting system engineering, [Musa79b]
warranty provision, [Thom80]
comparison of,
methods for obtaining confidence intervals, [Myhr68]
during development, [Shoo73]
error rate forecasting methods, [Nage82], [Nage84]
examples of,
from a space shuttle software project, [Misr83]
plan for the Air Force ASTROS project, [John75]
execution-time theory of, [Musa79a]
guidelines and standards,
RADC guidebook, [Goel83], [McCa87b]
example of telecommunication requirements, [Eric85]
management guidebook, [Glas79]
measurement, estimation and prediction, [Hech77b]
metrics for military systems, [Koss88]
operational reliability, [Brow75], [Litt78], [Mora75], [Nels73], [Nels78], [Weis85b], [Weis86], [Weis88b]
certifying from statistical testing, [Curr86], [Mill87a]
methods for determining confidence bounds, [Dura80]
supported by statistical sampling, [Dura81b], [East72]
supported by statistical testing, [Dyer85a]
others, [Hech77c], [Krus78], [Mora72]
principles and practices, [Hoer74]
quantitative measurement of, [Brow76]
review of prediction methods, [Misr83]
state of the art, [Litt80a], [Litt80b], [MiyaXX]
research directions, [Litt78]
system reliability, [Ande79b], [Han76], [Land77]
Bayesian software-hardware estimation, [Thom80]
comparison of hardware/software reliability, [Musa80b]
tutorials, [Hech80]
Reliability Models, [Dale86], [Musa80b]
Poisson model for Markov and semi-Markov structured software, [Litt76]
accounting for service organization characteristics, [Bake88]
analysis and validation of, [Scha79], [Wigg84]
Poisson and binomial models, [Angu80]

continuous probability distribution models, [Goel80c], [Schi78]
discrete models, [Broo80b]
discrete probability distribution models, [Goel80c], [Schi78]
time and data domain models, [Goel80c], [Schi78]
applications of, [Goel83], [Goel85]
Markov models,
validation, [Triv80]
probability-based model applied to code reading experiment, [Jeli73]
reliability growth models to support management, [Krug88]
automated support for,
SMERFS: Statistical Modeling and EStimation of Reliability Functions for Software, [Farr88]
classification of,
based on residual error size and testing process, [Rama82]
comparison of, [Farr83], [Musa87], [Suke77a], [Suke77b], [Suke79]
criteria for, [Iann84]
data requirements, [Duva80], [Farr83], [Litt80b], [Thay75]
de-eutrophication process model, [Jeli72]
deterministic and statistical models, [Moha79]
elimination of perfect debugging assumption, [Ohba89]
experiments in,
6 models applied to a C3I project, [Angu83]
failure rate assumption,
relaxation of, [Giam86]
for prediction,
analysis of quality of,
comparison of models, inference procedures, [Keil87]
micro model based on program structure, [Shoo76], [Shoo77a]
number of errors at start of testing,
based on development characteristics, [Taka89]
probabilistic model, [Shoo72], [Shoo77a]
for probabilistic program correctness, [Dura78]
supported by error reducing performance of development processes, [Dura78]
growth models,
for project management, [Krug88]
historical development of, [Schi78]
metrics,
incorporating into, [Henr88b]
parameter estimation,
examples of,
application of methods on a C3I project, [Angu83]
validation of methods, [Angu80]
resolving constraints from availability of data,
S-shaped and hyperexponential models, [Ohba84]
review of, [Farr83], [Goel83], [Goel85], [RADC76a]
selection of, [Abde86], [Goel83]
specific models, [Litt75]
Bayesian differential debugging model, [Litt80c]
Error Complexity Model, [Naka89]
Goel-Okumoto model,
for estimation of optimal test time, [Goel81]
Jelinski-Moranda model, [Litt81b]

comparison with other models, [Suke79]
experiments in, [Rowl88]
variations for early error estimation, [Mora80]
Poisson model for Markov and semi-Markov structured software, [Litt79]
S-shaped reliability growth model, [Yama83]
Schick-Wolverton model,
comparison with other models, [Suke79]
modified Schick-Wolverton model, [Suke79]
data domain models,
for fault-tolerant systems, [Scot83a], [Scot83b], [Scot84a], [Scot84b], [Scot87]
Markov model, [Sone80]
for fault-tolerant systems, [Hech79]
non-homogeneous Poisson process, [Schn75]
probabilistic model and stopping rule for debugging, [Form77]
stochastic Markov process for hardware/software, [Bene85]
automated support for, [Bene85]
stochastic growth model, [Litt81a]
stochastic model based on a non-homogeneous Poisson process, [Goel79]
with applications, [Goel80a], [Goel80b]
time-based models,
Musa's execution time-based model, [Musa84]
application in a computation center software, [Haml78c]
evaluation of, [Mill80c]
state/time-dependent failure rate, imperfect debugging,
binomial model for error occurrences, [Shan81]
maximum likelihood estimates for parameters, [Shan81]
relationship with other models, [Shan81]
survey of, [Shoo77a]
time-based models,
Bayesian growth model, [Litt73]
Musa's execution time-based model, [Musa75], [Musa79a]
Poisson-process models,
impact of test process, [Ehrl87]
completely monotonic regression estimates, [Mill85], [Mill86]
Reproducible Testing, [Weis88a]
approaches, [Tai85c], [Tai86]
for host-target environment, [Tayl82b]
for testing monitors, [Brin78]
automated support for dataflow machines, [Wahl88]
for Ada, [Tai85b]
Required Element Testing,
comparison with other techniques, [Ntaf81a]
strategies for, [Ntaf82]
evaluation of, [Ntaf84]
Requirements Analysis,
SAMM modeling tool, [Lamb78]
testability modeling using finite state machines, [Chan85]
Resource Estimation, see Cost Estimation, [Bail80]
Reusable LIbraries,
software,
Ada Software Repository,

metric analysis of, [Leac89]
Reusable Libraries,
hardware,
for hardware verification, [Bevi88]
software,
Ada Software Repository, [Conn87]
RLF: Reusability Library Framework project, [Sold89]
repository management, [Sold89]
Reuse,
analysis of, [Selb87d], [Selb88c]
domain modeling, [Sold89]
for Ada, see Ada, [Romb88h]
investigation of reuse and complexity, [Basi82d]
measurement of reusability, [Hess88]
research framework for, [Basi87d]
software,
evaluation of life cycle models for, [Guin89]
Reuse Libraries,
software,
Moorehouse object-oriented reuse library, [Jone89]
Revealing Subdomains, [Weyu80c]
Review of, [NBS82a]
automated support tools, [Perr88], [Rama75a], [Reif75]
cost estimating,
conversion cost estimating techniques, [Hout81]
models, [Ducl82]
formal functional specifications for modules, [Lisk79]
formal verification, [Dunn84]
graph partitioning methods, [Paig77b]
human factors,
psychological research on programming, [Shei81]
measurement,
studies at General Motors, [Elsh78a]
metrics,
Software Science, supporting evidence, [Fitz78a]
complexity metrics, [Ducl82]
for user information satisfaction, [Ives83]
process/product measures for error-prone software, [Shen85]
quality metrics, [Ducl82]
testing metrics, [Perr88]
reliability,
models, [Farr83], [Goel83], [Goel85], [RADC76a]
prediction methods, [Misr83]
testing and analysis techniques, [Clar78a], [Dunn84]
for real-time software, [Quir85]
testing environments, [Rama75a]
testing strategies, [Dunn84]
Risk Analysis, *see also* **FMEA**
DoD guidelines and standards,
risk abatement, [AFSC88b]
cost/benefit analysis,

using Bayesian decision model, [Wein80]
Risk Reduction Approaches,
dual programming, [Long77]
multi-specification, [Long77]
Run-Time Monitoring, *see also* **Instrumentation**
automated support for,
Algol68 numerical algorithms testbed, [Henn76a]
for Ada *see also* Ada,
for concurrent/distributed systems, EDL [Yau80]
Observer, [Ayac79]
problems, practices, roles, tools, [Joyc87b]
inquiry language and processor, [Cohe77]

SADMT, [Linn88]
automated support for,
SADMT/SF: SADMT Simulation Facility, [Linn88]
SAGEN user's guide, [Kapp88]
example of an architecture specification, [Ardo88]
interface to SADMT/SF, [Linn88]
SEL, [Basi77a], [Basi78a], [Card82]
Composite Specification Model (CSM), [Agre87]
compendium of tools, [Deck82b]
cost estimation, [McGa84]
data collection and analysis,
analyzing error data, [Basi81e]
automated support for, [Gree81]
database,
organization and user's guide, [Lo83], [NASA81]
procedures for the rehosted SEL database, [Hell87]
guide to data collection, [Chur82]
data compendium, [Turn81b]
glossary of software engineering terms, [Babs83]
operation of, [Basi78c]
recent studies, [McGa85a]
relationship equations, [Freb79]
specification measures for, [Agre84a], [Agre84c]
SEL Comparisons,
RADC and SEL software development data, [Turn81a]
resource utilization curves, [Basi81f]
SELECT a symbolic execution system, [Boye75]
SEL Evaluations and Experiments, [Card85b]
Ada, [Agre86], [Godf87]
IV&V methodology for flight dynamics, [Page85]
complexity measures, [Basi83b]
fault prediction and reliability assessment, [Basi86e]
resource forecasting, [Basi78a]
resource quality impact on product and process, [McGa85b]
software development practices, [Agre84b], [Chen81]
designing a measurement experiment, [Basi77b]
impact of design practices, [Card86a], [Card86b]
impact on productivity and reliability, [Card87b]

lessons learned, [Basi85d]
statistical model for evaluating effectiveness, [Card87b]
statistics on errors, [Basi82a], [Weis85c]
software metrics, [Basi81g]
structural coverage in SEL environment, [Basi84a]
study of Musa's reliability model, [Mill80c]
summary of software measurement experiences, [Vale89]
SEL Software Development Characteristics,
development measures, [Card84], [Hell87]
dynamic variables, [Basi83d], [Doer85]
evaluation of management measure, [Page82]
evaluation of, [Basi79c], [Basi81d], [Card81]
relationship among development variables, [Basi81a], [Basi85e]
environment characteristics, [Basi79a]
calculation and use of, [Basi85a]
use and interpretation, [Romb85b]
STAD: System for Testing and Debugging, [Kore85], [Kore86a], [Kore88]
YODA: Your Own Ada Debugger, [Lask88b]
trace database model, [LeDo85]
dependence-based modeling, [Kore86b], [Kore87]
Safety Analysis, *see also* **Fault-Tree Analysis** [Leve83d]
based on constrained expressions, see Constrained Expressions, [Leve87]
evaluation standards for safety critical software, [Parn88]
issues and research directions, [Leve86b]
of timing properties in real-time systems,
based on RTL: Real-Time Logic, [Jaha86]
quantitative measurement of safety, [Brow76]
Security Analysis,
basic security concepts, [Hall80]
Security Verification, *see also* **FDM, Gypsy, AFFIRM, A1 Certification, HDM**
comparison of specification paradigms, [Kauf87b]
comparison of techniques, [Mill81b]
requirements for secure operating systems, [Step74]
specification/verification of o/s security, [Feie80], [Kore84]
Self-Checking,
experiments in, [Cha87]
Simulation,
applications of,
design and validation aid, [Jack71]
evaluate designs/ease of error detection, [Schn77b]
performance evaluation, [Lee89b]
testbed for cooperative distributed problem solving, [Less80]
automated support for, SADMT, DARTS
TASKIT: Tasking Ada Simulation Kit), [Ange89]
Sneak Analysis, [Godo77]
Software Development Management, [Daly77]
control over software engineering process, [Dyer80]
designing/implementing a reliability program, [Rose85b]
guidelines and standards,
configuration management, [IEEE83c]
management indicators, [AFSC86a]

maintenance, [Adam84]
relationships of strategies to repair maintenance, [Grad87a]
state of the art, [Thay80]
supported by, SEL Software Development Characteristics
cost estimation, [Putn77], [Putn78], [Putn79]
macro variable models, [Gaff80]
management indicators, [Ross88]
models and metrics for, [Basi80a], [Gaff81a]
process-related productivity factors, [Vosb84]
productivity, performance, progress measurement, [Howe84]
reliability growth models, [Krug88]
Software Development Practices, *see also* **Chief Programmer Teams**, **Structured Programming**
a rigorous approach, [Jone80]
design,
evaluation of technology and practices, [Brun86]
overview of formal methods, [Hoar87]
parallel program design, [Chan88]
structured design, [Stev74], [Your76]
with constant evaluation by, [Ches77]
evaluation of, *see also* SEL Evaluations and Experiments
impact on understandability and modifiability, [Shep78]
lessons learned, [Basi85d]
through application to real projects, [Snee84]
problems and proposed solutions, [Zelk78]
programming,
by action clusters, [Naur69]
programming style, [Kern74a], [Kern74b]
Software Development Process,
guidelines and standards,
Defense Systems Software Development, [DOD88]
life cycle approaches,
evaluation wrt reuse, [Guin89]
evaluation wrt validation and verification, [Guin89]
iterative enhancement, [Basi75]
cost model for, [Ducl82]
paradigmatic approach, [Walk81]
risk-driven approach, Spiral Model, [Boeh86]
transformational approach, [Baue79b], [Baue89]
PROSPECTRA project, [Krie86]
model of construction/reasoning errors, [Howd89a]
prototyping,
evaluation of software prototypes, [Chur86]
operational specification as a basis for, [Balz82]
prototyping versus specifying, [Boeh84a]
uses of and techniques, [Tayl82a]
tailoring and improving process *see also* TAME,
Software Physics, [Hals75a], [Knij78]
analysis of Akiyama's debugging data, [Funa75]
evaluation of, [Love76]
experiments in, [Gord76]
Software Science, [Chri81], [Fitz78a] *see also* **Software Physics**, [Hals77a], [Hals78], [Harr88b], [Yeh79]

APL and Halstead's theory of metrics, [Deke81]
Halstead's criteria and statistical algorithms, [Bohr75]
adaptations of, [Bake79a]
applications of, [Smit79]
compiler performance evaluation, [Shaw89]
error prediction prior to testing, [Corn76], [Otte78], [Otte79], [Otte81]
evaluating modularity concepts, [Bake79b]
productivity prediction, [Come79], [Curt79a], [Curt79b], [Curt81], [Grem84], [Hals77d], [Moha79]
project management, [Hals77b], [Suno82]
automated support for, [Otte76]
comparison with other measures, [Albr83], [Bake80], [Blai85a], [Gaff79], [Kitc81], [Wood81a]
correlation with other measures, [Lind89]
counting strategies, [Fits79]
description and example of, [Salt82]
evaluation of, [Bake79a], [Basi81g], [Fitz78b], [Hame82], [Lass81], [List82], [Mora78c], [Shen83]
relationship between estimated/actual size, [Card87a]
relationship with development effort, [Kitc81], [Lind89]
review of supporting evidence, [Fitz78a], [Shen83]
validation across FORTRAN programs, [Basi83b]
with respect to cognitive psychology, [Coul83]
example analysis,
from technical writing, [Hals77c]
of COBOL programs, [Shen80]
of IBM programming products, [Smit80a]
of PL1 programs, [Elsh76a], [Zweb79]
of programming size, [Smit80b]
real-time switching system, [Bail81]
experiments in, [Come79]
for designs, [Szul81], [Szul84]
foundations, [Hals72a], [Hals73a], [Hals73b], [Hals76]
influencing factors,
basic constructs, [Lass79]
effect of the counting method, [Elsh78b]
vocabulary effects, [Fits80]
language level metric, [Cont81], [Olde77]
length equation, [John81]
theory, [Hals75b]
Special Values Testing,
comparison with other techniques, [Howd77c]
Specification-Based Testing, *see also* **Constraint Logic Programming**
comparison with other techniques, [Hetz76]
for test data generation, [Gour81], [Gour83], [Lask88a]
T-3 Testing Tool, [Lawr87]
state of the art, [Gour81]
using Prolog, [Boug85a]
Specification,
applications of, [Parn79]
transformational programming, [Baue89]
evaluation of techniques,
criteria for, [Lisk75]
formal methods of, [Berg82]

review of functional specifications techniques, [Lisk79]
incremental construction by combining of parallel elaborations, [Feat89]
research directions, [Lisk75]
role of, [Lisk75], [Lisk79]
testing, verification and analysis,
the LEONARDO project, [Gerh88a]
Specification Languages, *see also* **Finite State Machines, Abstract Data Types, PSL/PSA**
ASLAN, [Auer85]
Lotos, [Brin87], [ISO87c], [Najm87]
executing Lotos specifications, [Bria86]
PSL/PSA: Problem Statement Language/Problem Statement Analyzer,
modeled using axiomatic methods, [Gerh84]
SEMANOL(73), [Ande76b]
SEQuIFY sequence model system, [Gerh88a]
X a computer-based specification language, [Bish86]
Abstract specifications,
roles and examples of, [Parn77]
algebraic axioms,
combined with predicate transformers, [Gutt80]
example from a text editor, [McMu83]
used as a test driver *see also* DAISTS,
algebraic specifications, [Ehri85]
OBJ: a language for writing and testing, [Gogu79a]
testing of, [Gogu79b]
theory and application to testing, [GaudXX]
applications of,
defining abstract models of a system, [Ches77]
describing program behavior,
using time sequences, [Dahl79a]
documenting hierarchical design process, [Ches77]
for monitoring and debugging Ada,
relational algebra, [DiMa85]
for oracles, SEMANOL(73), [Ande76b]
runnable specifications as a design tool, [Davi82b]
testing communication protocols, [Dsso86]
to facilitate proof of correctness, [Noon75]
automated support for, [Pate89]
behavioral abstraction approach *see also* EDL,
desirable features of, [Gogu80]
distributed systems,
EBS: Event-Based Specification Language, [Chen83]
for Ada *see also* Ada,
for data types,
final data type specifications, [Kami80]
for hardware, VHSIC
HDL: Hardware Design Language, [Luck86a]
for real-time systems,
RT-ASLAN, [Auer86]
based on Lucid, [Skil89]
temporal assertions, [Lamp83]
larch family, [Gutt85]

predicate calculus,
for testing programs by specification mutation, [Budd85]
semi-formal approaches,
design conversations, [Conk88]
role-activity models, [Conk86]
scenarios, [Wexe87]
using traces to write abstract specifications, [Bart77]
a formal foundation, [MacL82]
Specification Mutation,
applications of,
for program testing, [Budd85]
Specification Testing and Analysis, [Gerr85], [Prob82b]
automated support for,
Inatest, [Eckm84], [Eckm85], [Kemm85a]
concurrent systems,
to detect synchronization errors, [Tai85a]
dual specification comparison based on symbolic execution, [Rama81]
Standards Checking,
automated support for, [Henn84]
Program Testing Translator, [Stuc72]
for Ada, based on DIANA intermediate form, [Byrn89]
Stanford Pascal Verifier, [Luck79a], [Luck79b]
survey of applications, [Luck77]
State Transition Models,
applications of,
axiomatic approaches, [Gutt77]
specification/verification of communication protocols, [Suns77], [Suns82]
automated support for *see also* AFFIRM,
Statement Testing,
automated support for *see also* DAISTS,
comparison with other techniques, [Selb86]
procedure coverage as an alternative, [Basi84a]
State of the Art,
DoD practices, [STE86]
automated support, [DeMi87a], [Reif75]
Ada compilation systems, [Bend89]
development environments, [Tayl87]
concepts/research issues in technology, [Wegn79]
contributions of experiments to software engineering, [Basi86a]
data collection and analysis, [Thib78]
formal verification, [Kemm86], [Land86], [Youn89a]
automated support for, [Crai86], [Crai87a]
for Ada, [Mayf85], [Mayf86], [Roby85]
measurement, [Youn89a]
design metrics evolution, [Romb88f]
metrics in quality assurance, [Gaff81b]
productivity issues of the 80's, [Jone81]
reliability measurement, [Bend86], [Keil87], [MiyaXX], [Musa80b]
software development management, [Thay80]
testing and analysis, [Budd83b], [DeMi87a], [Gerh79], [Good79a], [Hans84], [INFO79], [Land86], [Mill79a], [Youn89a]

challenges to the testing community, [Mill79c]
code reading and inspections, [Faga86]
data flow analysis, [Clar86a]
examination based on testing process model, [Gelp88]
issues, [Adri80]
mutation analysis, [Lipt78]
research directions, [Howd87]
specification-based program testing, [Gour81]
strengths, weaknesses, ope. ational characteristics, [Oste80], [Oste80]
techniques for real-time software, [Quir85]
technology needs in the 80's, [Mill79a]
verification in the 80's, [Gerh78]
State of the Practice,
automated support, [DeMi87a]
programming problem areas, [Elsh76b]
testing and analysis, [DeMi87a]
Static Analysis,
types of errors found and resource costs, [Gann79]
Static Concurrency Analysis, [Saxe77]
RGA: Reachability Graph Analyzer, [Morg84], [Morg86], [Morg87]
algorithm for, [Tayl81]
applications of,
reconstructing execution host/target, [Tayl82b]
structural testing, [Tayl86a]
combined with,
dynamic analysis, [Tayl83c]
principles for automated support, [Tayl83c]
symbolic execution, [Youn86a]
complexity of, [Tayl83b]
for Ada, see Ada, [Tayl83a]
syntax-based synchronization analysis with feasibility constraints, [Carv88]
Statistical Testing, [Dyer82a], [Dyer85a], [Mill72d]
certification of reliability, [Curr86], [Mill87a]
estimation of reliability, [Dyer85a]
relationship to formal verification, [Mill87a]
Structural Testing,
automated support for, DAISTS, FORTEST [Mill74a], [Mill74b]
FLOW, part of the PACE system, [Brow72a]
requisite support for concurrent systems, [Tayl86a]
combined with functional testing,
automated support for, [Clar78a]
comparison of coverage of metrics, [Ntaf85], [Weis85a]
comparison with other techniques, [Howd80c], [Hwan81]
fault detection effectiveness/cost faults, [Basi85b]
coverage measures,
as indicators of system performance, [Wu87c]
based on LCSAJs, [Henn76b]
definitions, [Mill80a]
for Ada, [Wu87c]
hierarchy of, [Wood80b]
statement and expression *see also* DAISTS,

evaluation of,
error detection ability, [Girg86a]
relationship of coverage/representativeness, [Brow75]
selectivity of path selection criteria, [Zeil88a]
exercising program segments, [Popk78]
for fault-tolerant systems, [Vouk86a], [Vouk86b]
Structured Programming, [Dahl72]
Dijkstra's calculus for formal program development, [Grie76]
and complexity,
formalization and application of, [McCl76]
measuring and controlling complexity, [McCl78a]
sources of complexity, [McCl78a]
error-free programming, [Mill75d]
experiments in, [Basi81c]
evaluation of, [Broo81], [John75]
formal verification of, [Ling79]
impact on quality, [Bake72a]
predicting effect on resource consumption, [Parr80]
process as well as program structure, [McCl78b]
theory of, [Ling79], [Mill75a]
Structured Testing, [Wals77c]
applications of, [McCa82a]
using Cyclomatic Complexity, [McCa76], [McCa82c], [Perr88]
Survey of,
automated support tools, [FSTC83], [Mill77b], [NBS82b], [Perr83]
debugging tools, [Schw70a]
error analysis work, [Amor75]
error types, frequencies and habitats, [Schw70a]
formal verification,
automated support tools,
Stanford Pascal Verifier, applications of, [Luck77]
mechanical support for formal reasoning, [Lind88d]
theorem provers, [Elsp72a]
results of Hoare's logic approach, [Apt81]
techniques, [Adri82], [Elsp72a]
for parallel programs, [Barr85]
for procedure and data abstractions, [Shan82]
theory, [Elsp72a]
measurement,
military standards/metrics for quality, [Bowe79]
reliability,
models, [Rama82], [Shoo77a]
technological management techniques, [Glas79]
testing and analysis techniques, [Adri82], [Bils83], [Mill72c], [NBS82b]
dynamic analysis, [Howd81c]
methods for estimating test data adequacy, [Rama82]
static analysis, [Howd81b]
communication protocols, recent developments, [Sari88a]
Symbolic Execution, *see also* **Symbolic Testing**
applications of, [Clar81a], [Clar85b]
debugging,

Symbolic Debug/1000, [HCP82]
using path rules, [Brue83]
fault-based testing, [More88]
symbolic fault tracking *see also* Perturbation Testing,
formal verification, [Clar84], [Hant76], [King76]
adaptation of Manna's technique, [Burs74]
for Ada *see also* Ada,
of communication protocols, [Bran78]
prototyping, [Cohe82]
testing and analysis, [Chea79], [Clar76a], [Clar76b], [Clar81c], [Darr78], [King75a], [King75b], [King76], [Rich85a]
combined with static concurrency analysis, [Youn86a]
compiler testing, [Same76]
fault-based testing, [More87]
partition analysis, [Rich81c]
path generation, [Clar84]
goal-oriented approach, [Wood80c]
test data generation methods, [Chen76], [Clar76a], [Clar84], [Howd77c], [Rama76]
automated support for, [Clar81a], [Clar85b]
design of, [Howd77a]
automated tools, *see also* SELECT, DISSECT
EFFIGY for PL/1 programs, [King75a], [King76]
Inatest, [Eckm84], [Eckm85], [Kemm85a]
UNISEX: a Unix-based executor for Pascal, [Eckm83b], [Kemm85b], [Soli83]
for Ada, see Ada, [Harr88a]
for ELI, [Chea79]
for FORTRAN, [Clar76b], [Rama76]
SADAT, [Voge80]
an executor based on MACSYMA, [Fava79]
conceptual representation for programs with side-effects, [Hewi76]
path selection, [Wood78]
strengths, weaknesses, operational characteristics, [Oste80]
Symbolic Testing,
Lindenmayer grammars, [Howd78e]
automated support for *see also* DISSECT,
estimation of cost using available systems, [Howd77a]
comparison with other techniques, [Howd77a], [Howd77c]
for Ada, see Ada, [Clar86b]
reliability of, [Howd77a], [Howd77b], [Howd77c]
System Structure,
cluster partitioning, [Hutc83]
automated support for, [Bela81]
metric to quantify partition complexity, [Bela81]
quantifying ratios of coupling/cohesion, [Selb88a], [Selb88b]
to support error localization, [Selb88a], [Selb88b]
cost of modularization, [Camp76]
criteria for modularization, [Card85d], [Parn72a], [Parn72b], [Schn77c]
based on issues of fault tolerance, [Rand75]
for extensible/contractable software, [Parn78]
information hiding, [Parn72c]
monitors as a structuring method, [Hoar74]

hierarchical ordering of functions/variability, [Dijk76b]
evaluation of,
based on information flow, [Henr79], [Henr81b], [Henr84]
experiments in,
global vs parameterized module connections, [Lohs84]
relationship with maintainability, [Gibs89]
meanings of the term "hierarchical structure", [Parn74]
modeling stabilization of a large system, [Hane72]
relating rate of progress to, [Parr80]
response times of level structured systems, [Hart84]
System Testing, [Perr88]
aided by structured analysis, [McCa82b]
estimating duration of, [Krug88]
impact on reliability growth models, [Ehrl87]
methods, [Cele81]
priority rules for test case selection, [Pets85]

TAME: Tailoring A Measurement Environment, [Basi87a], [Basi87b], [Romb88e]
exploiting feedback from evaluation, [Basi88]
improvement-oriented process model, [Romb88b]
integrating measurement into environments, [Basi87c]
lessons learned in the development process and measurement, [Romb85a]
tailoring process to goals, environments, [Basi87c], [Basi88]
TEAM: Testing, Evaluation, and Analysis Medley,
ARIES: a multi-lingual interpreter, [Epp86], [Zeil87]
design principles of, [Clar88a]
evaluation of testing and analysis techniques, [Clar88b]
integration of testing and analysis techniques, [Clar88a]
model for, [Clar88b]
TSL: Task Sequencing Language, [Helm85], [Luck86a]
TSL-2 for distributed systems, [Luck87]
testing and debugging of Ada programs,
runtime monitor, [Luck87]
Technology Transfer, [Whit88b]
Temporal Logic, [Krog87], [Lamp83], [Pnue77]
applications of,
design/synthesis of synchronization skeletons, [Clar81b]
verification of finite-state concurrent systems, [Clar86d]
combined with Ina Jo *see also* FDM
comparison of,
EBS with temporal logic and trace approaches, [Chen83]
complexity of, [Sist88]
proof systems based on temporal logic, [Barr84], [Nguy86], [Owic82]
time, clocks and the ordering of events, [Lamp78]
Test Data,
aid to proving correctness, [Gell78], [Howd78b]
Test Data Adequacy, [Weyu80b]
completeness criteria, [Wals85]
based on ability to distinguish functions, [Howd80d]
based on testing complexity, [Tai80]
for concurrent/distributed systems, [Weis87]

theory of,
abstract definition of, [Weyu83]
axiomatic theory of adequacy, [Weyu84b], [Weyu89], [Zweb89]
determining correctness, [Broo80d]
reliability, [Good75a], [Good75b], [Haml78a], [Howd76c], [Ostr78], [Ostr80]
revealing test criteria/subdomains, [Weyu80c]
testing for probable correctness, [Haml86], [Haml87]
theoretical analysis, [Davi83b]
two notions of correctness, [Budd80d]
Test Data Selection,
for loop free programs, [Cher79]
methodology for, [Howd74a], [Howd76b]
supported by,
analysis of memory dump, [Ehre76]
integer programming, [Lee88]
test case specifications,
TESTER/1, [Pete76]
Test Data Selection Criteria, *see also* **Mutation Analysis** [Clar78b], **Structural Testing, Data Flow Analysis**
for abstract code, DAISTS, EQUATE, Symbolic Fault Tracking
syntactic, semantic, methodological problems, [Zeil88c]
using Prolog, [Boug86]
Test Drivers,
TST: Ada Test Support Tool, [Maye89]
for Ada, see Ada, [Bess87]
for FORTRAN,
test procedure language/processor, [GE77a], [GE77b], [Panz76], [Panz78a], [Panz78b], [Panz78c]
for path testing, [Shoo79]
for pseudo-exhaustive testing, [Bagg78]
Test Effectiveness, [Perr83]
based on,
error reducing performance of development processes, [Dura78]
evaluation of test representativeness, [Brow75]
specifications in predicate calculus, [Budd85]
estimation of residual faults and effectiveness, [Bowe84]
formalism for completeness of error-based techniques, [Howd82a]
measurement of,
Algol68 numerical algorithms testbed, [Henn78], [Henn84]
Test Management, [Evan84b], [Perr83]
allocation and utilization of resources, [Shen85]
establishing comany-wide metrics program, [Grad87b]
guidelines and standards, [Hetz84]
Software Test and Evaluation Manual, [DODD87]
Test and Evaluation Master Plan guidelines, [DODD86b]
Test and Evaluation guidelines, [Army87], [DODD86a]
operational testing,
management guidelines, [AFOT86]
software acquisition guidance (maintenance), [Stan77]
methodology for test specification and auditing, [Ceri81]
test control process for functional testing, [Elme69]
traceability from requirements to system test, [Care77]
Test Path Adequacy, *see also* **Perturbation Testing**

measure for advantage of testing another path, [Zeil81b]
Test Path Generation,
algorithms for, [Han76]
automated support for *see also* Symbolic Execution
complexity of algorithms for building a path, [Gabo76]
notions of required pairs/paths, [Ntaf79], [Ntaf81b]
Test Planning, [Bran80], [Perr83]
based on structural characteristics, [Moha79]
based on testing theory, [Moha79]
effort estimation,
based on Goel-Okumoto reliability model, [Goel81]
based on measure of testability, [Moha76b]
optimum allocation of effort, [Down85a], [Down86]
predicting error content prior to testing,
using Software Science, [Corn76], [Otte78], [Otte81]
predicting errors content prior to testing,
using Software Science, [Otte79]
probabilistic cost model for optimal number of test cases, [Brow89]
for systems testing, [Perr88]
guidelines and standards, [Hetz84]
optimal testing, [Mitt82]
supported by,
network analysis, [Krau73]
test plan generation using formal grammars, [Baue79a]
Testing Environments, *see also* **Mothra**
Algol68 numerical algorithms testbed, [Henn78], [Henn84]
FORTRAN Automatic Code Evaluation System, [Rama73], [Rama74a]
ISMS experimental program testing facility, [Fair75]
IUTF: Interactive Unit Test Facility, [Tsal86]
Prufstand, [Snee78]
architectural overview of a distributed testbed, [Garc83]
for Ada *see also* TEAM,
ATVS: Ada Test and Verification System, [RADC86]
knowledge-based,
for kernel system calls of UNIX systems, [Pesc85]
program testing assistant, [Chap82]
review of, [Rama75a]
Testing Strategies, [Dunn84]
for expert systems, [Hite88]
for large, complex real-time systems, [Ginz65]
grey box testing, [Prob80], [Prob82a]
partition testing, [Haml88]
comparison with other strategies, [Haml88]
Theory of Programming, [Dahl72], [Davi83a], [Grie81]
Dijkstra's calculus, [Grie76]
a discipline, [Dijk76a]
axiomatic basis for, [Hoar69], [Hoar71a]
computability and unsolvability, [Davi82a]
computing as a physical science, [Good88]
convergence, correctness and equivalence,
of functional programs, [Mann70]

equivalence problem for loop-free programs, [Ibar82]
function semantics for sequential programs, [Mill80b]
mathematical theory of computation, [Mann74]
model of large program development, [Bela76]
nondeterminism, [Kenn80]
notions of correctness,
existential/universal partial/total correctness, [Gall81]
relationship between,
mathematical proof, algebraic languages, transcendental numbers, proof by sampling, [Davi77]
Theory of Testing, [Howd78d], [Prat83]
NP-completeness, [Gare78]
applications of,
linking theory with practice, [Mill77a]
test planning, [Moha79]
concurrent systems, [Weis87], [Weis88a], [Weis88c]
extension of sequential methods/theory, [Weis88a]
error propagation and elimination, [More81]
error-based testing, [More84]
fault-based testing, [More87], [More88]
investigative approaches,
(dis)advantages to theoretical/empirical, [Howd78a]
Popperian, [Cher87a]
abstract, [Boug85b], [Cher88]
as equivalence problem, [Howd78b]
general model for static analysis, [Howd83]
inductive inference, [Cher86], [Cher87b]
mathematical framework, [Gour81], [Gour83]
modeling the testing process, [Down86]
to study testing/debugging effectiveness, [Down85a]
uniform/nonuniform execution models, [Down86]
models of correct programs and testing, [Howd74b]
Tools,
JAVS: Jovial Automated Verification System, [RADC76b]
NODAL, [Mait80]
PACE: Product Assurance Confidence Evaluator,
programmer's guide, [Hoff73]
analysis,
supported by data management system, [John77]
classification of, [Reif79b]
practical applications of, [Brow72b]
test data generation, [Bast78], [Chen75], [Holt76]
ATDG: Automated Test Data Generator System, [Hoff75], [Hoff76]
for recursive programs having simple errors, [Broo80c]
supported by Prolog, [Gerh85]
testing of, [Henn79]
Trace Analysis,
for distributed systems, [Jard87]
communication protocols, [Boch88]
for conformance and arbitration testing, [Boch87a]
Transition Testing, [Beiz83]
Tutorials,

models and metrics for management/engineering, [Basi80a]
reliability, [Hech80]
testing and validation, [Mill81a]
validation and verification, [Yeh77]

User Interface Models,
Chiron for software environments, [Youn88b]

VDM: Vienna Software Development Method, [Bjor78], [Bjor82], [Bjor87]
example in analysis phase, [Bloo86]
with Prolog,
for animation of programs, [Bloo86]
for back-to-back testing of diverse software, [Bloo86]
VHSIC,
analysis of, [Luck86b]
semantics of timing constructs, [Luck86c]

Walkthroughs, see Code Reading and Inspections, [Myer78a]
Weak Mutation Analysis, [Howd82a]
automated support for, *see also* FORTEST
error detection ability, [Girg86a]
Wide-Spectrum Languages,
applications of,
transformational programming, [Baue89]
basis for a software development environment, [Luck86a]
for program specification and development, [Baue79b]

Zipf's Law,
applications of,
estimating size and effort, [Moha79]
m-EVES, [Crai88b], [Pase87a]
comparison with other techniques, [Crai88a]
example of low water mark problem, [Crai87b]
m-NEVER theorem prover, [Crai88a], [Pase87b]
m-Verdi, [Crai87c], [Crai87d], [Crai88a]

2. REFERENCES

[AFOT82] HQ Air Force Operational Test and Evaluation Center (AFOTEC). November 1982. *Software Operational Test and Evaluation Guidelines: Software Usability - Evaluator's Guide*. AFOTEC Pamphlet 800-2, Vol. 4. **

[AFOT86] HQ Air Force Operational Test and Evaluation Center (AFOTEC). August 1986. *Software Operational Test and Evaluation Guidelines: Management of Software Operational Test and Evaluation*. AFOTEC Pamphlet 800-2, Vol. 1.

[AFOT87] HQ Air Force Operational Test and Evaluation Center (AFOTEC). March 1987. *Software Operational Test and Evaluation Guidelines: Software Maintainability - Evaluator's Guide*. AFOTEC Pamphlet 800-2, Vol. 3.

[AFOT88a] HQ Air Force Operational Test and Evaluation Center (AFOTEC). May 1988. *Software Operational Test and Evaluation Guidelines: Software Support Resources - Evaluation Guide*. AFOTEC Pamphlet 800-2, Vol. 5.

[AFOT88b] HQ Air Force Operational Test and Evaluation Center (AFOTEC). November 1988. *Software Operational Test and Evaluation Guidelines: Software Support Life Cycle Process Evaluation Guide*. AFOTEC Pamphlet 800-2, Vol. 2.

[AFSC86a] Air Force Systems Command. January 1986. *Software Management Indicators*. AFSC Pamphlet 800-43.

[AFSC86b] Air Force Systems Command. January 1986. *Software Quality Indicators*. AFSC Pamphlet 800-14.

[AFSC88a] Air Force Systems Command and Air Force Logistics Command. 1988. *Software Independent Verification and Validation (IV&V)*. AFSC/AFLCP Pamphlet 800-5.

[AFSC88b] Air Force Systems Command and Air Force Logistics Command. 1988. *Software Risk Abatement*. AFSC/AFLCP Pamphlet 800-45.

[Abde86] Abdel-Ghaly, A.A., P.Y. Chan, and B. Littlewood. "Evaluation of Competing Software Reliability Predictions." 12/9 (Sep 1986):950-967.

[Acre79] Acree, A.T., R.A. DeMillo, T.J. Budd, R.J. Lipton, and F.G. Sayward. September 1979. *Mutation Analysis*. Georgia Institute of Technology. Technical Report GIT-ICS-70/08. Also Yale University Research Report 155. **

[Acre80] Acree, A.T. 1980. *On Mutation*. Ph.D. thesis, Georgia Institute of Technology.

[Adam80] Adam, A., and J. Laurent. "LAURA, A System to Debug Student Programs." *Artificial Intelligence*, 15/1-2 (Jan 1980):75-122.

[Adam84] Adams, E.N. "Optimizing Preventive Service of Software Products." *IBM Journal of Research and Development*, 28/1 (Jan 1984):2-14. **

[Adri80] Adrion, W.R. "Issues in Software Validation, Verification, and Testing." In *1980 TIMS-ORSA Conference. ORSA/TIMS Bulletin*, 10 (Sep 1980):80. **

[Adri82] Adrion, W.R., M.A. Branstad, and J.C. Cherniavsky. "Validation, Verification, and Testing of Computer Software." *ACM: Computing Surveys*, 14/2 (Jun 1982):159-192.

[Agil76] Agile, C.R. May 1976. *The Role of an Independent Software Validation Agency*. Fort Belvoir, VA: Defense Systems Management School. **

[Agre84a] Agresti, W.W. June 1984. *Definitions of Specification Measures for the Software Engineering Laboratory*. Computer Sciences Corp. Technical Report CSC/TM-84/6085. **

[Agre84b] Agresti, W.W., F.E. McGarry, D.N. Card, et al. 1984. *Measuring Software Technology*. New York: Springer-Verlag. **

[Agre84c] Agresti, W.W., V.E. Church, and F.E. McGarry. December 1984. *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-84-003. **

[Agre86] Agresti, W. 1986. "SEL Ada Experiment: Status and Design Experience." In *Proceedings 11th Annual Software Engineering Workshop*, December, Greenbelt, MD. NASA/GSFC. **

[**Agre87**] Agresti, W.W. June 1987. *Guidelines for Applying the Composite Specification Model (CSM)*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-87-003. **

[**Aho86**] Aho, A.V., R. Sethi, and J.D. Ullman. 1986. *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison Wesley. **

[**Al-J82**] Al-Jarrah, M.M.F. 1982. *The Study and Application of Program Analysis in a Cobol Environment*. Ph.D. thesis, Brunel University. **

[**Albe76**] Alberts D.S. 1976. "The Economics of Software Quality Assurance." In *Proceedings AFIPS National Computer Conference*, vol. 45, June 7-10, New York, NY, 433-442. Montvale, NJ: AFIPS Press.

[**Albr79**] Albrecht, A.J. 1979. "Measuring Application Development Productivity." In *Proceedings IBM Application Development Symposium*, October 14-17, Monterey, CA, 83-92. GUIDE Int. and SHARE Int., IBM Corp. **

[**Albr81**] Albrecht, A.J. 1981. "Function Points as a Measure of Productivity." In *Proceedings GUIDE 53 Meeting*, November 12, Dallas, TX. **

[**Albr83**] Albrecht, A.J., and J.E. Gaffney. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." *IEEE: Transactions on Software Engineering*, 9/6 (Nov 1983):639-648.

[**Alle71**] Allen, F.E. 1972. "Graph-Theoretic Constructs for Program Control Flow Analysis." In *Proceedings Information Processing (IFIP) Congress '71*, 385-390. **

[**Alle74**] Allen, F.E. 1974. "Interprocedural Data Flow Analysis." In *Proceedings Information Processing (IFIP) Congress '74*, August 5-10, Stockholm, Sweden, 398-402. Amsterdam: North-Holland.

[**Alle76**] Allen, F.E., and J. Cocke. "A Program Data Flow Analysis Procedure." *ACM: Communications of the ACM*, 19/3 (Mar 1976):137-147.

[**Ambl76a**] Ambler, A.L., et al. 1976. "Gypsy: A Language for Specification and Implementation of Verifiable Programs." *ACM: SIGPLAN Notices*, 12/3 (Mar 1976).

[**Ambl76b**] Ambler, A.L., D.I. Good and W.F. Burger. August 1976. *Report on the Language GYPSY*. Certifiable Minicomputer Project. University of Texas. Technical Report ICSCA-CMP-1. **

[**Amor75**] Amory, W., and J.A. Clapp. January 1975. *Engineering of Quality Software Systems (A Software Error Classification Methodology)*. Griffiss Air Force Base, NY: Rome Air Development Center. RADC-TR-74-325, Vol. III.

[**Amor89**] Amoroso, E.G., and T.D. Nguyen. 1989. "An Approach to Ada Compiler Acceptance Testing." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 266-268. Washington, DC: ACM Ada Technical Committee. **

[**Amst76**] Amster, S.J., E.J. Davis, B.N. Dickman, and J.P. Kuoni. 1976. "An Experiment in Automatic Quality Evaluation of Software." In *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York, 171-179. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press.

[**Ande76a**] Anderson, T., and R. Kerr. 1976. "Recovery Blocks in Action: A System Supporting High Reliability." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA:,. Washington, DC: IEEE Computer Society Press.

[**Ande76b**] Anderson, E.R., F.C. Belz, and E.K. Blum. "SEMANOL(73), A Metalanguage for Programming the Semantics of Programming Languages." *Acta Informatica*, 6/1 (1976):109-131.

[**Ande79a**] Anderson, R.B. 1979. *Proving Programs Correct*. New York: John Wiley & Sons.

[**Ande79b**] Anderson, T., and B. Randell (eds.). 1979. *Computing Systems Reliability*. Cambridge: Cambridge University Press. **

[**Ande81**] Anderson, T., and P.A. Lee. 1981. *Fault Tolerance Principles and Practices*. Englewood Cliffs, NJ: Prentice Hall. **

[**Ande83**] Anderson, T., and J.C. Knight. "A Framework for Software Fault Tolerance in Real-Time Systems." *IEEE: Transactions on Software Engineering*, 9/3 (May 1983):355-364.

[**Ande85**] Anderson, T., P.A. Barrett, D.N. Halliwell, and M.R. Moulding. "Software Fault Tolerance: An Evaluation." *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1502-1510.

[Ande88] Anderson, J.D., and J.A. Perkins. 1988. "Experience Using an Automated Metrics Framework in the Review of Ada Source for WIS." In *Proceedings 6th National Conference on Ada Technology*, March 14-17, Arlington, VA, 32-41. Washington, DC: ACM Ada Technical Committee.

[Andr81] Andrews, D.M., and J.P. Benson. 1981. "An Automated Program Testing Methodology and Its Implementation." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 254-261. Washington, DC: IEEE Computer Society Press.

[Ange89] Angel, M., and P. Juozitis. 1989. "Taskit: An Ada Simluation Tool Kit Featuring Machine Independent Parallel Processing." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 122-127. Washington, DC: ACM Ada Technical Committee. **

[Angl76] Angluin, D.C. 1976. *An Application of the Theory of Computational Complexity to the Study of Inductive Inference*. Ph.D. diss., University of California at Berkeley. **

[Angl80] Angluin, D. "Inductive Inference of Formal Languages from Positive Data." *Information and Control*, 45/2 (May 1980):117-135.

[Angl83] Angluin, D., and C.H. Smith. "Inductive Inference: Theory and Methods." *ACM: Computing Surveys*, 15 (1983):237-269.

[Angu80] Angus, J.E., R.E. Schaffer, and A. Sukert. 1980. "Software Reliability Model Validation." In *Proceedings Annual Reliability and Maintainability Symposium*, 191-199.

[Angu83] Angus, J., et al. August 1983. *Reliability Model Demonstration Study*. Griffiss Air Force Base, NY: Rome Air Development Center. RADC-TR-83-207.

[Appe88] Appelbe, W.F., R.A. DeMillo, D.S. Guindi, K.N. King, and W.M. McCracken. 1988. *Using Mutation Analysis for Testing Ada Programs*. Purdue University. Technical Report SERC-TR-9-P. Also published in *Proceedings Ada Europe Conference*, June, Munich, Germany . New York: Cambridge University Press.

[Apt80] Apt, K.R., N. Francez, and W.P. de Roever. "A Proof System for Communicating Sequential Processes." *ACM: Transactions on Programming Languages and Systems*, 2/3 (Jul 1980):359-385.

[Apt81] Apt, K.R. "Ten Years of Hoare's Logic: A Survey–Part I." *Transactions on Programming Languages and Systems*, 3/4 (Oct 1981):431-483.

[Apt83a] Apt, K.R. "Formal Justification of a Proof System for Communicating Sequential Processes." *Journal of the ACM*, 30/1 (Jan 1983):197-216.

[Apt83b] Apt, K.R. 1983. "A Static Analysis of CSP Programs." In *Proceedings of the Workshop on Program Logic*, June, Pittsburgh, PA. **

[Ardo88] Ardoin, C.D., S.H. Edwards, M.R. Kappel, C.J. Linn, J.L. Linn, and J. Salasin. April 1988. *A Simple Example of an SADMT Architecture Specification: Version 1.5*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2036.

[Army84] US Army. 1984. *Software Quality Engineering Handbook*. US Army Computer Systems Command, Quality Assurance Directorate.

[Army87] U.S. Army Missile Command. February 1987. *Software Test and Evaluation Manual, Vol. II, Guidelines for Software Test and Evaluation in the Department of Defense*.

[Arth88] Arthur, J.D., R.E. Nance, and K.T. Stevens. 1988. *Prospects for Automated Documentation Analysis in Support of Software Quality Assurance*. Virginia Polytechnic Institute. TR-88-33.

[Auer85] Auernheimer, B., and R.A. Kemmerer. March 1985. *ASLAN User's Manual*. University of California at Santa Barbara. Technical Report TRCS84-10. **

[Auer86] Auernheimer, B., and R.A. Kemmerer. "RT-ASLAN: A Specification Language for Real-Time Systems." *IEEE: Transactions on Software Engineering*, 12/9 (Sep 1986):879-889.

[Aver84] Avery, S. June 1984. *Development of a Behavior Generator for Constrained Expressions*. University of Massachusetts. Technical Report SDLM/84-2. **

[Aviz75] Avizienis, A. 1975. "Fault Tolerance and Fault Intolerance: Complementary Approaches to Reliable Computing." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 458-464. IEEE Cat. No. 75CH0940-7CSR.

[Aviz77] Avizienis, A., and L. Chen. 1977. "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution." In *Proceedings 1st International Computer Software and*

Applications Conference, November 8-11, Chicago, IL, 149-155. Long Beach, CA: IEEE Computer Society Press.

[Aviz78] Avizienis, A. 1978. "Fault-Tolerance: The Survival Attribute of Digital Systems." In *Proceedings of the IEEE*, 66 (1978), 1109-1125. **

[Aviz84] Avizienis, A., and J.P. Kelly. "Fault Tolerance by Design Diversity: Concepts and Experiments." *IEEE: Computer* 17/8 (Aug 1984):67-80.

[Aviz85] Avizienis, A. "The N-Version Approach to Fault-Tolerant Software." *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1491-1501.

[Aviz87] Avizienis, A. "On the Achievement of a Highly Dependable and Fault-Tolerant Air Traffic Control System." *IEEE: Computer*, 20/2 (Feb 1987):84-90.

[Avru83] Avrunin, G., and J. Wileden. 1983. "Algebraic Techniques for the Analysis of Concurrent Systems." In *Proceedings IEEE 16th Hawaii International Conference on System Sciences*, January, Honolulu, HA, 51-57. **

[Avru85] Avrunin, G.S., and J.C. Wileden. "Describing and Analyzing Distributed Software System Designs." *ACM: Transactions on Programming Languages and Systems*, 7/3 (Jul 1985):380-403.

[Avru86] Avrunin, G.S., L.K. Dillon, J.C. Wileden, and W.E. Riddle. "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems." *IEEE: Transactions on Software Engineering*, 12/2 (Feb 1986):278-292.

[Ayac79] Ayache, J.M., P. Azema, and M. Diaz. 1979. "Observer: A Concept for Detecting at Run Time Control Errors in Concurrent Systems." In *Proceedings IEEE Fault-Tolerant Computing Symposium*, June, Madison. **

[Babs83] Babst, T.A., F.E. McGarry, and M.G. Rohleder. October 1983. *Glossary of Software Engineering Laboratory Terms*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-82-105. **

[Bagg78] Baggi, D.L., and M.L. Shooman. 1978. "An Automatic Driver for Pseudo-Exhaustive Software Testing." In *Proceedings COMPCON '78*, February 28 - March 3, San Francisco, CA, 278. IEEE. **

[Bagg80] Baggi, D.L., and M.L. Shooman. March 1980. *Software Test Models and Implementation of Associated Test Drivers*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-80-45.

[Baia84] Baiardi, F., A. Fantechi, and M. Vaneschi. 1984. "Static Checking of Interprocess Communication in ECSP." In *Proceedings ACM-SIGPLAN '84 Symposium on Compiler Construction*, June, Montreal. Published in *ACM: SIGPLAN Notices*, 19/6 (Jun 1984):290-299.

[Baia85] Baiardi, F., N. De Francesco, E. Matteoli, S. Stefanini, and G. Vaglini. 1985. "Development of a Debugger for a Concurrent Language." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 98-106. Washington, DC: IEEE Computer Society Press.

[Bail80] Bailey, J.W., and V.R. Basili. August 1980. *A Meta-Model for Software Development Resource Expenditures*. University of Maryland. Technical Report TR-935. Also published in *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 107-116. Washington, DC: IEEE Computer Society Press.

[Bail81] Bailey, C.T., and W.L. Dingee. 1981. "A Software Study Using Halstead Metrics." In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March:189-197.

[Bail83] Bailey, J.E., and S.W. Pearson. "Development of a Tool for Measuring and Analyzing Computer User Satisfaction." *Management Science*, 29/5 (May 1983):530-545. **

[Bake72a] Baker F.T. 1972. "System Quality Through Structured Programming." In *Proceedings AFIPS Fall Joint Computer Conference*, vol. 41, December 5-7, Anaheim, CA, 339-343. Montvale, NJ: AFIPS Press.

[Bake72b] Baker, F.T. "Chief Programmer Team Management of Production Programming." *IBM Systems Journal*, 11/1 (1972):131-149.

[Bake77] Baker, W.F. June 1977. *Software Data Collection and Analysis: A Real-Time System Project History*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-77-192. **

[Bake79a] Baker, A.L. 1979. *Software Science and Program Complexity*. Ph.D. diss., Ohio State University. **

[Bake79b] Baker, A.L., and S.H. Zweben. "The Use of Software Science in Evaluating Modularity Concepts." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):110-120.

[Bake80] Baker, A.L., and S.H. Zweben. "A Comparison of Measures of Control Flow Complexity." *IEEE: Transactions on Software Engineering*, 6/6 (Jun 1980):506-512.

[Bake81] Baker, F.T. 1981. "Chief Programmer Teams." In *Tutorial on Structured Programming: Integrated Practices*, V.R. Basili and F.T. Baker (eds.). IEEE. **

[Bake88] Baker, C.T. "Effects of Field Service on Software Reliability." *IEEE: Transactions on Software Engineering*, 14/2 (Feb 1988):254-258.

[Bald79] Baldwin, D., and F. Sayward. 1979. *Heuristics for Determining Equivalence of Program Mutations*. Yale University. Computer Science Research Report 276. **

[Balz69] Balzer, R.M. "EXDAMS - Extendable Debugging and Monitoring System." In *Proceedings AFIPS Spring Joint Computer Conference*, vol. 34, 567-580. Montvale, NJ: AFIPS Press.

[Balz81] Balzer, R.M. 1981. *Design Specification Validation*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-81-102. **

[Balz82] Balzer, R.M., N. Goldman, D. Wile. 1982. "Operational Specification as the Basis for Rapid Prototyping." In *Proceedings Rapid Prototyping Conference*. **

[Barb88] Barbeau, M., and B. Sarikaya. 1988. "A Computer Aided Design Tool for Protocol Testing." In *Proceedings IFOCOM '88*, March, New Orleans. **

[Bark89] Barkataki, S., and J. Kelly. 1989. "Software Quality Assurance in an Ada Environment." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 362-367. Washington, DC: ACM Ada Technical Committee. **

[Barr82] Barringer, H., and I. Mearns. "Axioms and Proof Rules for Ada Tasks." In *Proceedings 6th International Computer Software and Applications Conference*, March 9-12, San Diego, CA. Los Angeles, CA: IEEE Computer Society.

[Barr84] Barringer, H., and R. Kuiper. 1984. "Now You may Compose Temporal Logic Specifications." In *Proceedings 16th ACM Symposium on the Theory of Computing*, April 30 - May 2, Washington, DC, 51-63. Baltimore, MD: ACM Order Department.

[Barr85] Barringer, H. 1985. "A Survey of Verification Techniques for Parallel Programs." *Lecture Notes in Computer Science*, Vol. 191. New York: Springer-Verlag.

[Bart77] Bartussek, W., and D.L. Parnas. 1977. *Using Traces to Write Abstract Specifications for Software Modules*. University of North Carolina. Technical Report TR 77-02. **

[Bart78] Barth, J.M. "A Practical Interprocedural Data-Flow Analysis Algorithm." *ACM: Communications of the ACM*, 21/9 (Sep 1978):724-736.

[Bart80] Bartlett, K.A., and D. Rayner. 1980. "The Certification of Data Communication Protocols." In *Proceedings IEEE Symposium on Computer Network Protocols*, May, Washington, DC, 12-17. **

[Barz75] Barzdin, J.M., J.J. Bicevskis, and A.A. Kalninsh. 1975. "A Construction of Complete Sample System for Correctness Testing." In *Lecture Notes in Computer Science*, Vol. 32, 1-12. Berlin: Springer-Verlag. **

[Basi75] Basili, V.R., and A.J. Turner. "Iterative Enhancement: A Practical Technique for Software Development." *IEEE: Transactions on Software Engineering*, 1/4 (Dec 1975):390-396.

[Basi77a] Basili, V.R., M.V. Zelkowitz, F.E. McGarry, R.W. Reiter Jr., W.F. Truszkowski, and D.L. Weiss. May 1977. *The Software Engineering Laboratory*. Greenbelt, MD: NASA/GSFC. Report SEL-77-001. **

[Basi77b] Basili, V.R., and M.V. Zelkowitz. 1977. "Designing a Software Measurement Experiment." In *Proceedings 2nd Life Cycle Management Workshop*, August. **

[Basi78a] Basili, V.R., and M.V. Zelkowitz. "Analyzing Medium-Scale Software Developments." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 116-123. Washington, DC: IEEE Computer Society Press.

[Basi78b] Basili, V.R., and R.W. Reiter Jr. August 1978. *Investigating Software Development Approaches*. University of Maryland. Technical Report TR-688. **

[Basi78c] Basili, V.R., and M.V. Zelkowitz. 1977. "Operation of the Software Engineering Laboratory." In *Proceedings U.S. Army Computer Systems Command Software Life Cycle Management Workshop*, August 21-22. **

[Basi79a] Basili, V.R., and M.V. Zelkowitz. "Measuring Software Development Characteristics in the Local Environment." *Computer and Structures*, 10/8 (Aug 1979):39-43.

[Basi79b] Basili, V.R., and R.W. Reiter. "An Investigation of Human Factors in Software Development." *IEEE: Computer*, 12/12 (Dec 1979):21-38.

[Basi79c] Basili, V.R., and R.W. Reiter, Jr. 1979. "Evaluating Automatable Measures of Software Development." In *Proceedings Workshop on Quantitative Software Models*, October, Kiamesha Lake, NY, 107-116. IEEE Computer Society.

[Basi80a] Basili, V.R. 1980. *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society. Also published in *ASME Advances in Computer Technology*, Vol. 1. **

[Basi80b] Basili, V.R., and D.H. Hutchens. 1980. "A Study of a Family of Structural Complexity Metrics." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 13-16.

[Basi80c] Basili, V.R. 1980. "Data Collection Validation and Analysis." In *Draft Software Metrics Panel Final Report*, A.J. Perlis, F.G. Sayward, and M. Shaw (eds.). Washington, DC. **

[Basi81a] Basili, V.R., and K. Freburger. "Programming Measurement and Estimation in the Software Engineering Laboratory." *Journal of Systems and Software*, 2/1 (Feb 1981):47-57.

[Basi81b] Basili, V.R., and D.M. Weiss. 1981. "Evaluation of a Software Requirements Document by Analysis of Change Data." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 314-322. Washington, DC: IEEE Computer Society Press.

[Basi81c] Basili, V.R., and R.W. Reiter. "A Controlled Experiment Quantitatively Comparing Software Development Approaches." *IEEE: Transactions on Software Engineering*, 7/5 (May 1981):299-320.

[Basi81d] Basili, V.R. 1981. "Evaluating Software Development Characteristics: Assessment of Software Measures in the Software Engineering Laboratory." In *Proceedings 6th Annual Software Engineering Workshop*, December, Grenbelt, MD. NASA/GSFC.

[Basi81e] Basili, V.R., and D.M. Weiss. 1981. "Analyzing Error Data in the Software Engineering Laboratory." In *Proceedings 4th Minnowbrook Workshop on Software Performance Evaluation*, August, Blue Mountain Lake, NY. **

[Basi81f] Basili, V.R., and J. Beane. "Can the Parr Curve Help with the Manpower Distribution and Resource Estimation Problems." *Journal of Systems and Software*, 2/1 (Feb 1981):59-69.

[Basi81g] Basili, V.R., and T. Phillips. 1981. "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory." In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March.

[Basi82a] Basili, V.R., and D.M. Weiss. December 1982. *Evaluating Software Development by Analysis of Changes: The Data from the Software Engineering Laboratory*. University of Maryland. Technical Report TR-1236.

[Basi82b] Basili V.R., J. Bailey, J.D. Gannon, E. Kruesi, E. Katz, S. Sheppard, and M.V. Zelkowitz. "Monitoring an Ada Software Development Project." *General Electric Company: Newsletter*, 1/2 (Dec 1982). Also published in *ACM: Ada Letters*, II/1 (Jun 1982):1.58-1.61. Updated and published in *ACM: Ada Letters*, IV/I (Jul-Aug 1984):32-39.

[Basi82c] Basili, V.R., and D.M. Weiss. 1982. *A Methodology for Collecting Valid Software Engineering Data*. University of Maryland. Technical Report TR-1235. Also published in *IEEE: Transactions on Software Engineering*, 10/6 (Nov 1984):728-738.

[Basi82d] Basili, V.R., and B.T. Perricone. 1982. *Software Errors and Complexity: An Empirical Investigation*. University of Maryland. Technical Report TR-1195. Also published in *Communications of the ACM*, 27/1 (Jan 1984):42-52.

[Basi83a] Basili, V.R., and E.E. Katz. 1983. "Metrics of Interest in an Ada Development." In *Proceedings IEEE Computer Society Workshop on Software Engineering Technology Transfer*, April 25-27, Miami

Beach, FL, 22-29. Los Angeles, CA: IEEE Computer Society.

[Basi83b] Basili, V.R., R.W. Selby, Jr., and T.Y. Phillips. "Metric Analysis and Data Validation Across Fortran Projects." *IEEE: Transactions on Software Engineering*, 9/6 (Nov 1983):652-663.

[Basi83c] Basili, V.R. "An Empirical Study of a Syntactic Complexity Family." *IEEE: Transactions on Software Engineering*, 9/6 (Nov 1983):664-672.

[Basi83d] Basili, V.R., and C. Doerflinger. 1983. "Monitoring Software Development Through Dynamic Variables." In *Proceedings 7th International Computer Software and Applications Conference*, Novemer 7-11, Chicago, IL, 394-395. Los Angeles, CA: IEEE Computer Society. **

[Basi84a] Basili, V.R., and J. Ramsey. September 1984. *Structural Coverage of Functional Testing*. University of Maryland. Technical Report TR-1442. Also published in *Proceedings 7th Minnowbrook Workshop on Software Performance Evaluation*, July 24-27, Blue Mountain Lake, NY.

[Basi84b] Basili, V.R., and R.W. Selby Jr. "Data Collection and Analysis in Software Research and Management." In *Proceedings American Statistical Association and Biometric Society Joint Statistical Meetings*, August 13-16, Philadelphia, PA, 21-30. **

[Basi84c] Basili, V.R., and E.E. Katz. January 1984. *A Taxonomy of Metrics in Terms of an Ada Development*. University of Maryland. **

[Basi84d] Basili, V.R., N.M. Panlilio-Yap, C.L. Ramsey, C. Shih, and E.E. Katz. May 1984. *A Quantitative Analysis of a Software Development in Ada*. University of Maryland. Technical Report TR-1403. Also published in *IEEE: Computer*, 18/9 (Sep 1985):53-65.

[Basi85a] Basili, V.R., and R.W. Selby Jr. 1985. "Calculation and Use of an Environment's Characteristic Software Metric Set." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 386-391. Washington, DC: IEEE Computer Society Press.

[Basi85b] Basili, V.R., and R.W. Selby Jr. May 1985. *Comparing the Effectiveness of Software Testing Strategies*. University of Maryland. Technical Report TR-1501. Also published in *IEEE: Transactions on Software Engineering*, 13/12 (Dec 1987):1278-1296.

[Basi85c] Basili, V.R. July 1985. *Quantitative Evaluation of Software Methodology*. University of Maryland. Technical Report TR-1519. Also published in *Proceedings 1st Pan Pacific Computer Conference*, September. Australian Computer Society.

[Basi85d] Basili, V.R. 1985. "Can We Measure Software Technology: Lessons Learned from 8 Years of Trying." In *Proceedings 10th Annual Software Engineering Workshop*, December, Greenbelt, MD. NASA/GSFC. **

[Basi85e] Basili, V.R., and N.M. Panlilio-Yap. 1985. "Finding Relationships Between Effort and Other Variables in the SEL." In *Proceedings 9th International Computer Software and Applications Conference*, October 9-11, 221-228. Los Angeles, CA: IEEE Computer Society.

[Basi85f] Basili, V.R., and R.W. Selby Jr. 1985. "Four Applications of a Software Data Collection and Analysis Methodology." In *Proceedings 10th Annual Software Engineering Workshop*, December, Greenbelt, MD. NASA/GSFC. **

[Basi85g] Basili, V.R., and C. Loggia-Ramsey. 1985. "ARROWSMITH-P: A Prototype Expert System for Software Engineering Management." In *Proceedings IEEE Symposium on Expert Systems in Government*, October 23-25, McLean, VA, 252-264. Los Angeles, CA: IEEE Computer Society. **

[Basi85h] Basili, V.R., E.E. Katz, N.M. Panlilo-Yap, C.L. Ramsey, and S. Chang. "Characterization of an Ada Software Development." *IEEE: Computer*, 18/9 (Sep 1985):53-65.

[Basi86a] Basili, V.R., R.W. Selby Jr., and D.H. Hutchens. 1986. "Experimentation in Software Engineering." *IEEE: Transactions on Software Engineering*, 12/7 (Jul 1986):733-743.

[Basi86b] Basili, V.R., H.D. Rombach, and R.W. Selby Jr. August 1986. "The Role of Code Reading in the Software Life Cycle." In *Proceedings 9th Minnowbrook Workshop on Software Performance Evaluation*, August 5-8, Blue Mountain Lake, NY. **

[Basi86c] Basili, V.R., and E.E. Katz. 1986. *A Formalization and Categorization of Software Metrics*. University of Maryland. Working Paper. **

[Basi86d] Basili, V.R., and L. Wu. 1986. "Structure Coverage Tools for Ada Software Systems." In *Proceedings 4th Annual National Conference on Ada Technology*, March, Atlanta, GA. **

[Basi86e] Basili, V.R., and D. Patnaik. August 1986. *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*. University of Maryland. Technical Report TR-1699. **

[Basi87a] Basili, V.R., and H.D. Rombach. 1987. "TAME: Tailoring an Ada Measurement Environment." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 318-325. Washington, DC: ACM Ada Technical Committee.

[Basi87b] Basili, V.R., and H.D. Rombach. 1987. "Tailoring the Software Process to Project Goals and Environments." University of Maryland. Technical Report TR-1728. Also published in *Proceedings 9th International Conference on Software Engineering*, March 30 - Arpil 2, Monterey, CA, 345-357. Washington, DC: IEEE Computer Society Press.

[Basi87c] Basili, V.R. June 1987. *TAME: Integrating Measurement into Software Environments*. University of Maryland. Technical Report TR-1764. (TAME-TR-1-1987). **

[Basi87d] Basili, V.R. 1987. "Software Reuse: A Research Framework." In *Proceedings 10th Minnowbrook Workshop on Software Reuse*, August, Blue Mountain Lake, NY. Submitted to *IEEE Computer Magazine*. **

[Basi88] Basili, V.R., and H.D. Rombach. "The TAME Project: Towards Improvement-Oriented Software Environments." *IEEE: Transactions on Software Engineering*, 14/6 (Jun 1988):758-773.

[Bast78] Bastani, F.B. 1978. *The Specification, Design and Implementation of an Automated Test Data Generator*. M.S. Report, University of California at Berkeley. **

[Bate81] Bates, P.C. and J.C. Wileden. 1981. *Event Definition Language: An Aid to Monitoring and Debugging Complex Software Systems*. University of Massachusetts. COINS Technical Report 81-17. **

[Bate82] Bates, P.C., and J.C. Wileden. 1982. "EDL: A Basis for Distributed System Debugging Tools." In *Proceedings 15th Hawaii International Conference on System Sciences*, January, Honolulu, Hawaii, 86-93. **

[Bate83a] Bates, P.C., and J.C. Wileden. "An Approach to High-Level Debugging of Distributed Systems." *Journal of Systems and Software*, 3 (Dec 1983):255-264.

[Bate83b] Bates, P.C., J.C. Wileden, and V.R. Lesser. 1983. "A Debugging Tool for Distributed Systems." In *Proceedings 2nd Annual Phoenix Conference on Computers and Communications*, 311-315. **

[Batt87] Battaglia, M. May 1987. *Integrated Diagnostics Program Plan and Roadmap*. Joint Policy Coordinating Group: Logistics Research, Development Test and Evaluation Integrated Diagnostics Working Panel. **

[Baue79a] Bauer, J.A., and A.B. Finger. 1979. "Test Plan Generation Using Formal Grammars." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 425-432. Washington, DC: IEEE Computer Society Press. **

[Baue79b] Bauer, F.L., M. Broy, R. Gratz, W. Hesse, B. Krieg-Brueckner, H. Partsch, P. Pepper, and H. Wossner. 1979. "Towards a Wide-Spectrum Language to Support Program Specification and Program Development." In *Program Construction: Lecture Notes in Computer Science*. Springer-Verlag.

[Baue89] Bauer, F.L., B. Moller, H. Partsch, and P. Pepper. "Formal Program Construction by Transformations–Computer-Aided, Intuition-Guided Programming." *IEEE: Transactions on Software Engineering*, 15/2 (Feb 1989):165-180.

[Bazz82] Bazzichi, F., and I. Spadafora. "An Automatic Generator for Compiler Testing." *IEEE: Transactions on Software Engineering*, 8/4 (Jul 1982):343-353.

[Beck76] Beckman, L., A. Haraldson, O. Oskarsson, and E. Sandewall. "A Partial Evaluator and Its Use as a Programming Tool." *Artificial Intelligence*, 7/4 (1976):319-357.

[Beel85] Beeler, J. "Programmer Productivity: Never Have So Many Done So Little for So Much." *Computerworld*, December 30, 1985, 40-46. **

[Behr83] Behrens, C.A. "Measuring the Productivity of Computer Systems Development Activities with Function Points." *IEEE: Transactions on Software Engineering*, 9/6 (Nov 1983):648-652.

[Beiz83] Beizer, B. 1983. *Software Testing Techniques*. New York: Van Nostrand Reinhold.

[Bela76] Belady, L.A., and M.M. Lehman. "A Model of Large Program development." *IBM Systems Journal* 15/3 (1976):225-251.

[Bela77] Belady, L.A. August 1977. "Software Complexity." In *Software Phenomenology*. Washington, DC: U.S. Army Institute for Research in Management Information and Computer Science. **

[Bela81] Belady, L.A., and C.J. Evangelist. "System Partitioning and Its Measure." *Journal of Systems and Software*, 2/1 (Feb 1981):23-29.

[Belf79] Belford, P.C., R.C. Berg, and T.L. Hannan. 1979. "Central Flow Control Software Development: A Case Study of the Effectiveness of Software Engineering Techniques." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 85-93. Washington, DC: IEEE Computer Society Press. **

[Belk86] Belkhouche, B., J.E. Urban. "Direct Implementation of Abstract Data Types from Abstract Specifications." *IEEE: Transactions on Software Engineering*, 12/5 (May 1989):649-661.

[Bell74] Bell, D.E., and J.E. Sullivan. June 1974. *Further Investigation into the Complexity of Software*. MITRE. Technical Report MTR-2874, Vol. II. **

[Bend86] Bendell, A., and P. Mellor (eds.). 1986. *Reliability: State of the Art*. Oxford: Pergamon Infotech. **

[Bend89] Bender, M.E., and T.E. Griest. 1989. "Real-Time Ada Demonstration Project." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 154-161. Washington, DC: ACM Ada Technical Committee. **

[Bene85] Benejean, R., J.C. Michon, and J.P. Signoret. 1985. "Software Tools as an Aid for Hardware and Software Reliability Analysis."

[Beng87] Bengtson, N.M. "Measuring Errors in Operational Analysis Assumptions." *IEEE: Transactions on Software Engineering*, 13/7 (Jul 1987):767-776.

[Bens81] Benson, J.P. 1981. "Adaptive Search Techniques Applied to Software Testing." In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March:109-116.

[Bent87] Bentley, J.L., and B.W. Kernighan. January 1987. *A System for Algorithm Animation: Tutorial and User Manual*. AT&T Bell Laboratories. **

[Beny79] Benyon-Tinker, G. 1979. "Complexity Measures in an Evolving Large System." In *Proceedings Workshop on Quantitative Software Models*, October, Kiamesha Lake, NY, 117-127. IEEE Computer Society. **

[Bera83] Berard, E.V. "Halstead's Metrics and Ada." *ACM: Ada Letters*, 3/3 (Nov-Dec 1983):33-44.

[Berg82] Berg, H.K., W.E. Boebert, W.R. Franta, and T. Moher. 1982. *Formal Methods of Program Verification and Specification*. Englewood Cliffs, NJ: Prentice-Hall.

[Berl80] Berlinger, E. 1980. "An Information Theory Based Complexity Measure." In *Proceedings AFIPS National Computer Conference*, vol. 49, May 19-22, Anaheim, CA, 773-779. Arlington, VA: AFIPS Press.

[Bern84] Berns, G.M. "Assessing Software Maintainability." *ACM: Communications of the ACM*, 27/1 (Jan 1984):15-23.

[Berr87] Berry, D.M. "Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language." *IEEE: Transactions on Software Engineering*, 13/2 (Feb 1987):184-201.

[Besh85] Besharatian, R.H., M. Bloom, and J. Salasin. 1985. "Software Quality Data Sheets: Can Software Consumers be Informed?" In *Proceedings IEEE Global Telecommunications Conference*, December 2-5, New Orleans, LA, 56-60. Piscataway, NJ: IEEE Service Center.

[Bess87] Besson, M. and B. Queyras. 1987. "GET: A Test Environment Generator for Ada." In *Proceedings Ada Europe Conference*, May 26-28, Stockholm, Sweden, :237-250. New York: Cambridge University Press

[Bevi87] Bevier, W.R. October 1987. *A Verified Operating System Kernel*. Ph.D. diss., University of Texas. Also published as Computational Logic, Inc., Technical Report CLI-11. **

[Bevi88] Bevier, W.R. June 1988. *A Library for Hardware Verification*. Computational Logic, Inc., Technical Report Internal Note 57. **

[Bieb85] Biebow, B., and J. Hagelstein. 1985. "Algebraic Specification of Synchronization and Errors." In *Proceedings Colloquim on Software Engineering*, Berlin. **

[Bils83] Bilsel, M.S. April 1983. *A Survey of Software Test and Evaluation Techniques*. Georgia Institute of Technology. Technical Report GIT-ICS-83/08. **

[Bird83] Bird, D.L., and C.U. Munoz. "Automatic Generation of Random Self-Checking Test Cases." *IBM Systems Journal*, 22/3 (1983):229-245. **

[Bish86] Bishop, P.G., D.G. Esp, M. Barnes, P. Humphreys, G. Dahl, and J. Lahti. "PODS – A Project on Diverse Software." *IEEE: Transactions on Software Engineering*, 12/9 (Sep 1986):929-940.

[Bjor78] Bjorner, D., and C.B. Jones. 1978. "The Vienna Development Method." In *Lecture Notes in Computer Science*, Vol. 61. Springer-Verlag. **

[Bjor82] Bjorner, D., and C.B. Jones. 1982. *Formal Specification and Software Development*. Englewood Cliffs, NJ: Prentice-Hall. **

[Bjor87] Bjorner, D. 1987. "On the Use of Formal Methods in Software Development." In *Proceedings 9th International Conference on Software Engineering*, March 30 - Arpil 2, Monterey, CA, 17-29. Washington, DC: IEEE Computer Society Press.

[Blac77] Black, R.K., R.P. Curnow, R. Katz, and M.D. Gray. March 1977. *BCS Software Production Data*. Boeing Computer Services Inc. Final Technical Report RADC-TR-77-116. **

[Blac81] Black, J.P., D.J. Taylor, and D.E. Morgan. "A Case Study in Fault Tolerant Software." *Software-Practice and Experience*, no. 11 (1981):143-157.

[Blai71] Blair, J. 1971. "Extendable Non-Interactive Debugging." In *Debugging Techniques in Large Systems*, R. Rustin (ed.). Englewood Cliffs, NJ: Prentice-Hall.

[Blai85a] Blaine, J.D., and R.A Kemmerer. "Complexity Measures for Assembly Language Programs." *Journal of Systems and Software*, 5 (1985):229-245.

[Blai85b] Blaine, J. 1985. *Software Metrics and Program Maintenance: A Case Study of a Real Time Software Project*. M.S. thesis, University of California at Santa Barbara. **

[Bloo86] Bloomfield, R.E., and P.K. Froome. "The Application of Formal Methods to the Assessment of High Integrity Software." *IEEE: Transactions of Software Engineering*, 12/9 (Sep 1986):988-993.

[Blum75] Blum, L., and M. Blum. "Toward a Mathematical Theory of Inductive Inference." *Information and Control*, 28/1 (May 1975):125-155.

[Boch78] Bochmann, G.v. "Finite Descriptions of Communication Protocols." *Computer Networks*, 2 (Oct 1978):361-372. **

[Boch80] Bochmann, G.v., and C.A. Sunshine. "Formal Methods in Communication Protocol Design." *IEEE: Transactions on Computers*, C-28/4 (Apr 1980):624-631. **

[Boch87a] Bochmann,G.v., R. Dssouli, J.-R. Zhao. March 1987. *Trace Analysis for Conformance and Arbitration Testing*. Montreal University. Research Report. **

[Boch87b] Bochmann, G.v., G.W. Geber, and J.-M. Serre. "Semi-Automatic Implementation of Communication Protocols." *IEEE: Transactions on Software Engineering*, 13/9 (Sep 1987):989-1000.

[Boch88] Bochmann, G.v., C.S. He, D. Ouimet, and J.-R. Zhao. January 1988. *Protocol Testing using Automated Trace Analysis*. Montreal University. Research Report. **

[Boeh73] Boehm, B.W. "Software and Its Impact: A Quantitative Assessment." *Datamation*, 19/5 (May 1973):48-59. **

[Boeh75a] Boehm, B.W. "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software." *IEEE: Transactions on Software Engineering*, 1/1 (Mar 1975):125-133.

[Boeh75b] Boehm, B.W. 1975. "The High Cost of Software." *Practical Strategies for Developing Large Software Systems*, 3-14.

[Boeh78] Boehm, B.W., J.R. Brown, H. Kaspar, M. Lipow, G.J. MacLeod, and M.J. Merrit. 1978. *Characteristics of Software Quality*. New York: North Holland. Previously published as TRW Technical Report 25201-6001-RU-00 in 1973. Also published in *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA, 592-605. Washington, DC: IEEE Computer Society Press.

[Boeh81] Boehm, B.W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall. Also published in *IEEE: Transactions on Software Engineering*, 10/1 (Jan 1984):4-21.

[Boeh84a] Boehm, B.W., T.E. Gray, and T. Seewaldt. "Prototyping Versus Specifying: A Multiproject Experiment." *IEEE: Transactions on Software Engineering*, 10/3 (May 1984):290-303.

[Boeh84b] Boehm, B.W., M.H. Penedo, E.D. Stuckle, et al. "A Software Development Environment for Improving Productivity." *IEEE: Computer*, 17/5 (Jun 1984):30-42.

[Boeh86] Boehm, B.W. "A Spiral Model of Software Development and Enhancement." *ACM: Software Engineering Notes*, 11/4 (Aug 1986):22-42.

[Boeh87] Boehm, B. "Industrial Software Metrics Top 10 List." *IEEE: Software*, (Sep 1987):84-85.

[Bohr75] Bohrer, R. 1975. "Halstead's Criteria and Statistical Algorithms." In *Proceedings 8th Annual Computer Science Statistics Symposium*, February, Los Angeles, CA, 262-266. **

[Boie72] Boies, S.J., and J.D. Gould. June 1972. *A Behavioral Analysis of Programming--On the Frequency of Syntactical Errors*. Yorktown Heights, NY: IBM Research Center. Report RC-3907. **

[Bonn84] Bonnett, B. 1984. "Software in Safety and Security Critical Systems." Presented at COMPCON 84, September, Washington, DC. Transcript of the panel session available from A.W. Friend, ELEX 70343, NAVELEX, Washington, DC. **

[Boot80] Booth, T.L., and C.A. Wiecek. "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design." *IEEE: Transactions on Software Engineering*, 6/3 (Mar 1980):138-151.

[Boro72] Borodin, A. "Computational Complexity and the Existence of Complexity Gaps." *ACM: Journal of the ACM*, 19/1 (Jan 1972):158-183.

[Boug85a] Bouge, L., N. Choquet, L. Fribourg, and M.C. Gaudel. 1985. "Application of Prolog to Test Sets Generation from Algebraic Specifications." In *Proceedings TAPSOFT Joint Conference on Theory and Practice of Software Development*, March, Berlin. **

[Boug85b] Bouge, L. "A Proposition for a Theory of Testing: An Abstract Approach to the Testing Process." *Theoretical Computer Science*, 37/2 (1985):151-181. **

[Boug86] Bouge, L., N. Choquet, L. Fribourg, and M.C. Gaudel. "Test Sets Generation from Algebraic Specifications Using Logic Programming." *Journal of Systems and Software*, 6/4 (Nov 1986):343-360.

[Bowe78] Bowen, J.B. January 1978. *AN/SPS-52B (DDG) Radar System Software Reliability Study*. Hughes-Fullerton. Technical Report FR77-14-1106. **

[Bowe79] Bowen, J.B. "A Survey of Standards and Proposed Metrics for Software Quality Testing." *IEEE: Computer*, 12/8 (Aug 1979):37-42.

[Bowe80] Bowen, J.B. 1980. "Standard Error Classification to Support Software Reliability Assessment." In *Proceedings AFIPS National Computer Conference*, vol. 49, May 19-22, Anaheim, CA, 697-705. Arlington, VA: AFIPS Press.

[Bowe83] Bowen, T.P, J.V. Post, J. Tsai, P.E. Presson, and R.L. Schmidt. July 1983. *Software Quality Measurement for Distributed Systems, Vols. I, II and III*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-83-175.

[Bowe84] Bowen, J. 1984. "Estimation of Residual Faults and Testing Effectiveness." In *Proceedings 7th Minnowbrook Workshop on Software Performance Evaluation*, July 24-27, Blue Mountain Lake, NY. **

[Bowe85] Bowen, T.P., G.B. Wigle, and J.T. Tsai. February 1985. *Specification of Software Quality Attributes*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-85-37 (3 Vols.).

[Bows87] Bowser, J. 1987. *Reference Manual for Ada Mutant Operators*. Unpublished manuscript. **

[BowsXX] Bowser, J.H., and C.A. Budinger. *Procedures Used in the Testing of Mothra*. Georgia Institute of Technology. **

[Boye75] Boyer, R.S., B. Elspas, and K.N. Levitt. 1975. "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution." *ACM: SIGPLAN Notices*, 10/6 (Jun 1975):234-245.

[Boye79] Boyer, R.S., and J.S. Moore. 1979. *A Computational Logic Handbook*. Perspectives in Computing, vol. 23. San Diego, CA: Academic Press.

[Boye80] Boyer, R.S., and J.S. Moore. "A Theorem Prover for Recursive Functions." *ACM: SIGSOFT Software Engineering Notes*, 5/3 (Jul 1980):4.

[Boye81] Boyer, R.S., and J.S. Moore. 1981. *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures*. In *The Correctness Problem in Computer Science*, R.S. Boyer and J.S. Moore (eds.). London: Academic Press. **

[Boye83] Boyer, R.S., and J.S. Moore. 1983. *A Mechanical Proof of the Turing Completeness of Pure Lisp*. University of Texas. Technical Report ICSCA-CMP-37. **

[Boye84a] Boyer, R.S., and J.S. Moore. "Proof Checking, Theorem Proving, and Program Verification." *American Mathematical Society Contemporary Mathematics Series*, no. 29 (1984):119-132.

[Boye84b] Boyer, R.S., and J.S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm." *American Mathematical Monthly*, 91/3 (1984):181-189. **

[Boye88] Boyer, R.S., and J.S. Moore. 1988. *A User's Manual for a Computational Logic*. Computational Logic Inc. Technical Report CLI-18. **

[Boys79] Boysen J.P. 1979. *Factors Affecting Computer Program Comprehension*. Ph.D. diss., Iowa State University.

[Brad75] Bradley, G.H., T.F. Green, G.T. Howard, and N.F. Schneidewind. 1975. "Structure and Error Detection in Computer Software." In *Proceedings AIEE Conference*, 54-59. **

[Bran78] Brand, D., and W.H. Joyner. "Verification of Protocols Using Symbolic Execution." *Computer Networks*, 2 (1978). **

[Bran80] Branstad, M.A., J.C. Cherniavsky, and W.R. Adrion. February 1980. *Validation, Verification, and Testing for the Individual Programmer*. Gaithersburg, MD: National Bureau of Standards. Special Publication 500-56.

[Bria86] Briand, J.P., M.C. Fehri, L. Logrippo, and A. Obaid. 1986. "Executing Lotos Specifications." In *Proceedings 6th IFIP Workshop on Protocols*, June, 73-84. North-Holland. **

[Bril84] Brilliant, S.S. May 1985. *Analysis of Faults in a Multi-Version Software Experiment*. M.S. thesis, University of Virginia. **

[Bril87] Brilliant, S.S. September 1987. *Software Testing Using Multiple Versions*. Ph.D. diss., University of Virginia. **

[Brin73] Brinch Hansen. P. "Testing Multiprogramming Systems." *Software Practice and Experience*, 3/2 (Apr-Jun 1973):145-150.

[Brin78] Brinch Hansen, P.B. "Reproducible Testing of Monitors." *Software Practice and Experience*, 8/6 (1978):721-729.

[Brin85] Brindle, A.F., R.N. Taylor, and D.F. Martin. September 1985. *A Debugger for Ada Tasking*. El Segundo, CA: The Aerospace Corp. ATR-85(8033)-1. Also published in *IEEE: Transactions on Software Engineering*, 15/3 (Mar 1989):293-304.

[Brin87] Brinksma, E., G. Scollo, and C. Steenberger. 1986. "Lotos Specifications, Their Implementations and Their Tests." In *Proceedings 6th IFIP Workshop on Protocols*, June, 349-360. North-Holland. **

[Bris79] Bristow, G., C. Drey, B. Edwards, and W.E. Riddle. 1979. "Anomaly Detection in Concurrent Programs." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 265-273. Washington, DC: IEEE Computer Society Press. **

[Brit82] Britcher, R.N., and J.E. Gaffney. 1982. "Estimates of Software Size from State Machine Designs." In *Proceedings 2nd Annual Software Engineering Workshop*. Greenbelt, MD. NASA/GSFC. **

[Brit88] Britcher, R.N. "Using Inspections to Investigate Program Correctness." *IEEE: Computer*, 21/11 (Nov 1988):38-44.

[Broo75] Brooks, F.P. 1975. *The Mythical Man-Month*. Reading, MA: Addison Wesley.

[Broo80a] Brooks, R.E. "Studying Programmer Behavior: The Problem of Proper Methodology." *ACM: Communications of the ACM*, 23/4 (Apr 1980):207-213.

[Broo80b] Brooks, W.D., and R.W. Motley. April 1980. *Analysis of Discrete Software Reliability Models*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-80-84. **

[Broo80c] Brooks, M. 1980. *Automatic Generation of Test Data for Recursive Programs Having Simple Errors*. Ph.D. thesis, Stanford University. **

[Broo80d] Brooks, M.F. 1980. *Determining Correctness by Testing*. Ph.D. diss., Stanford University.

[Broo81] Brooks, W.D. "Software Technology Payoff: Some Statistical Evidence." *Journal of Systems and Software*, 2/1 (Feb 1981):3-9.

[Brop87] Brophy C., W. Agresti, and V.R. Basili. 1987. "Lessons Learned in the Use of Ada Oriented Design Methods." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and*

Washington Ada Symposium, March 16-19, Arlington, VA, 231-236. Washington, DC: ACM Ada Technical Committee.

[Brow72a] Brown, J.R., and R.H. Hoffman. 1972. "Evaluating the Effectiveness of Software Verification – Practical Experience with an Automated Tool." In *Proceedings AFIPS Fall Joint Computer Conference*, vol. 41, December 5-7, Anaheim, CA, 181-190. Montvale, NJ: AFIPS Press.

[Brow72b] Brown, J.R. September 1972. "Practical Applications of Automated Software Tools." Redondo Beach, CA: TRW Systems. Technical Report TRW-SS-72-05. **

[Brow73a] Brown, S.R., et al. 1973. "Automated Software Quality Assurance." In *Program Test Methods*, 181-204. Englewood Cliffs, NJ: Prentice Hall. **

[Brow73b] Brown, A.R., and W.A. Sampson. 1973. *Program Debugging*. New York: American Elsevier and MacDonald. **

[Brow75] Brown, J.R., and M. Lipow. "Testing for Software Reliability." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 518-527. IEEE Cat. No. 75CH0940-7CSR.

[Brow76] Brown, J.R., and M. Lipow. August 1976. "The Quantitative Measurement of Software Safety and Reliability." In *TRW Software Series*. Revised from TRW Report SDP-1776, August 1973. **

[Brow78] Browne, J.C., and D.B. Johnson. 1978. "FAST: A Second Generation Program Analysis System." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 142-148. Washington, DC: IEEE Computer Society Press.

[Brow80a] Brown, P.J. "Why Does Software Die?" In *Life-Cycle Management*, Infotech State of the Art Report, 8/7 (1980). **

[Brow80b] Browne, J., and M. Shaw. June 1980. *Toward a Scientific Base for Software Evaluation*. ONR (AD A087 412), Software Metrics Panel Final Report. **

[Brow89] Brown, D.B., S. Maghsoodloo, and W.H. Deason. "A Cost Model for Determining the Optimal Number of Software Test Cases." *IEEE: Transactions on Software Engineering*, 15/2 (Feb 1989):218-229.

[Brue83] Bruegge, B., and P. Hibbard. 1983. "Generalized Path Expressions: A High Level Debugging Mechanism." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):61-78. Baltimore, MD: ACM Order Department.

[Brun85] Brunelle, J.E., and D.E. Eckhardt. 1985. "Fault Tolerant Software: Experiments with the SIFT Operating System." In *Proceedings 5th AIAA Conference on Computers in Aerospace*, October, Long Beach, CA, 355-360. **

[Brun86] Bruns, G., S. Gerhart, C. Johnson, and A. Yaung. June 1986. *Design Technology Assessment*. MCC. Technical Report STP-179-87. **

[Brya80] Bryan, W.L. 1980. "The Practical Application of Software Product Assurance." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 131-136.

[Bryk89] Brykczynski, B., and C. Youngblut. 1989. *Towards SDS Testing and Evaluation: A Collection of Relevant Topics*. IDA Draft Memorandum Report M-513. VA: Institute for Defense Analyses.

[Buck79] Buckley, F. "A Standard for Software Quality Assurance Plans." *IEEE: Computer*, 12/8 (Aug 1979):43-51.

[Buck81] Buck, F.O. September 1981. *Indicators of Quality Inspections*. Kingston, NY: IBM Systems Products Division. Technical Report 21.802. **

[Budd77] Budd, T.A., and F. Sayward. 1977. *Users Guide to the Pilot Mutation System*. Yale University. Technical Report 114. **

[Budd78a] Budd, T.A., R.J. Lipton, F.G. Sayward, and R. DeMillo. 1978. "The Design of a Prototype Mutation System for Program Testing." In *Proceedings AFIPS National Computer Conference*, vol. 47, June 5-8, Anaheim, CA, 623-627. Arlington, VA: AFIPS Press.

[Budd78b] Budd, T.A., and R.J. Lipton. 1978. "Mutation Analysis of Decision Table Programs." In *Proceedings 1978 Conference on Information Sciences and Systems*, Baltimore, MD, 346-349. John Hopkins University. **

[Budd78c] Budd, T.A., and R.J. Lipton. 1978. "Proving LISP Programs Using Test Data." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983), 374-403. Baltimore, MD: ACM Order Department. **

[Budd80a] Budd, T.A., R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1980. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs." In *Proceedings 7th ACM Annual Symposium on Principles of Programming Languages*, January 28-30, Las Vegas, NV, 220-233. Baltimore, MD: ACM Order Department.

[Budd80b] Budd, T.A. 1980. *Mutation Analysis of Program Test Data*. Ph.D. thesis, Yale University. **

[Budd80c] Budd, T.A., R. Hess, and F.G. Sayward. 1980. *EXPER Implementor's Guide*. Yale University. **

[Budd80d] Budd, T.A., and D. Angluin. 1980. *Two Notions of Correctness and Their Relation to Testing*. University of Arizona. Technical Report 80-19b. Also published in *ACTA Informatica*, no. 18 (1982):31-45.

[Budd81] Budd, T.A. 1981. "Mutation Analysis: Ideas, Examples, Problems and Prospects." In *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 129-148. Amsterdam: North-Holland. **

[Budd83a] Budd, T.A. March 1983. *The Portable Mutation Testing Suite*. University of Arizona. Technical Report TR 83-8. **

[Budd83b] Budd, T.A. 1983. "Techniques for Advanced Software Validation." In *State of the Art Report, 11:3, Software Engineering: Development*. Berkshire, England: Pergamon Infotech. **

[Budd85] Budd, T.A, and A.S. Gopal. "Program Testing by Specification Mutation." *IEEE: Computer Language*, 10/1 (Jan 1985).

[Bulu74] Bulut, N., and M.H. Halstead. "Impurities Found in Algorithm Implementations." *ACM: SIGPLAN Notices*, 9/3 (Mar 1974):9-12. **

[Bunc80] Bunce, W.E. 1980. "Hardware and Software: An Analytical Approach." In *Proceedings Annual Reliability and Maintainability Symposium*, 209-213.

[Burn78] Burns, J. 1978. "The Stability of Test Data from Program Mutation." In *Digest IEEE Workshop on Software Testing and Test Documentation*, December 18-20, Ft. Lauderdale, FL, 324-334. IEEE Computer Society Technical Committee on Software Engineering. **

[Burs74] Burstall, R.M. 1974. "Program Proving as Hand Simulation with a Little Induction." In *Proceedings Information Processing 6th World Computer Congress*, August 5-10, Stockholm, Sweden, 308-312. Amsterdam: North-Holland.

[Byrn89] Byrnes, C. 1989. "A DIANA Query Language for the Analysis of Ada Software." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 511-518. Washington, DC: ACM Ada Technical Committee. **

[CSC78] *Acceptance Test Methods*. Computer Sciences Corp. Report TM-78/6296, October 1978. **

[CSDL80] *Design Aids for Real-Time Systems (DARTS: A Designer's Manual) (preliminary)*. Cambridge, MA: Charles Stark Draper Laboratory, Inc., January 1980. **

[Cagl82] Caglayan, M.U. 1982. *A Method for the Design, Representation and Analysis of Distributed Software Systems using Modified Petri Nets*. Ph.D. diss., Northwestern University.

[Cail79] Cailliau, R., and F. Rubin. "On a Controlled Experiment in Program Testing." ACM Forum, *ACM: Communications of the ACM*, 22/12 (Dec 1979):687-688.

[Camp74] Campbell R.H., and A.N. Habermann. 1974. "The Specification of Process Synchronization by Path-Expressions." In *Lecture Notes in Computer Science*, Operating Systems 16. G. Goos and J. Hartmanis (eds.), 89-102. New York: Springer-Verlag. **

[Camp76] Camp, J.W, and E.P. Jensen. 1976. "Cost of Modularity." *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press.

[Camp79] Campbell, R.H., and R.B. Kolstad. 1979. "Path Expressions in Pascal." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 212-219. Washington, DC: IEEE Computer Society Press.

[Cann85] Canning, J.T. 1985. *The Application of Structure and Code Metrics to Large-Scale Systems*. Ph.D. thesis, Blacksburg University. **

[Cant89] Cantone, G., A. Cimitile, and U. De Carlini. "Graphs, Programs and Metrics." Submitted to *IEEE: Transactions on Software Engineering*. **

[Card81] Card, D.N. 1981. "Identification and Evaluation of Software Measures." In *Proceedings 6th Annual Software Engineering Workshop*, December, Grenbelt, MD. NASA/GSFC. **

[Card82] Card, D.N., F.E. McGarry, J. Page, S. Eslinger, and V.R. Basili. February 1982. *The Software Engineering Laboratory*. Greenbelt, MD: NASA/GSFC. Report SEL-81-104. **

[Card84] Card, D.N., F.E. McGarry, J. Page, et al. March 1984. *Measures and Metrics for Software Development*. Greenbelt, MD: NASA/GSFC. Report SEL-82-002. **

[Card85a] Card, D.N., R.W. Selby Jr., F.E. McGarry, et al. April 1985. *A Comparison of Software Validation Techniques*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-85-001. **

[Card85b] Card, D.N. 1985. "A Software Technology Evaluation Program." In *Annais do XVIII Congresso Nacional de Informatica*, October. **

[Card85c] Card, D.N., C. Antle, and E. Edwards. December 1985. *Software Verification and Testing*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-85-005. **

[Card85d] Card, D.N., G.T. Page, and F.E. McGarry. 1985. "Criteria for Software Modularization." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England. Washington, DC: IEEE Computer Society Press. **

[Card86a] Card, D.N., V.E. Church, and W.M. Agresti. "An Empirical Study of Software Design Practices." *IEEE: Transactions on Software Engineering*, 12/2 (Feb 1986):264-271.

[Card86b] Card, D.N. October 1986. *Measuring Software Design*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-86-005. **

[Card87a] Card, D.N., and W.W. Agresti. "Resolving the Software Science Anomaly." *Journal of Systems and Software*, 7/1 (Mar 87):29-36.

[Card87b] Card, D.N., F.E. McGarry, and G.T. Page. "Evaluating Software Engineering Technologies." *IEEE: Transactions on Software Engineering*, 13/7 (Jul 1987):845-851.

[Care77] Carey, R., and M. Bendick. 1977. "The Control of a Software Test Process." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 327-333. Long Beach, CA: IEEE Computer Society Press.

[Carp75] Carpenter, L.C., and L.L. Tripp. 1975. "Software Design Validation Tool." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 395-400. IEEE Cat. No. 75CH0940-7CSR.

[Carr80] Carre, B.A. "Software Validation." *Microprocessors and Microsystems*, 4/10 (1980):395-406. **

[Carr82] Carre, B.A. 1982. "Control-Flow, Data-Flow and Information-Flow in Programs." In *IEE Colloquium Digest 82/85: A Review of Verification Methods for Software and Digital Systems*, 1:1-1:4. **

[Cars84] Carson, S.D. May 1981. *Geometric Models of Concurrency*. Ph.D. diss., University of Virginia. **

[Cart81] Cartwright, R. "Formal Program Testing." In *Proceedings 8th ACM Annual Symposium on Principles of Programming Languages*. **

[Carv88] Carver, R.H., and K.-C. Tai. 1988. "A Semantics-Based Approach to Analyzing Concurrent Programs." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 132-133. Washington, DC: IEEE Computer Society Press.

[Cava78] Cavano, J., and J.A. McCall. 1978. "A Framework for the Measurement of Software Quality." In *Proceedings ACM Software Quality Assurance Workshop*, November 15-17, San Diego, CA, 133-139. New York: Association for Computing Machinery.

[Cele80] Celentano, A., et al. "Compiler Testing Using a Sentence Generator." *Software Practice and Experience*, 10 (1980):897-918. **

[Cele81] Celentano, A., C. Ghezzi, and F. Liguori. "A Systematic Approach to System and Acceptance Testing." In *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 279-287. North-Holland. **

[Ceri81] Ceriani, M., A. Cicu, and M. Maiocchi. 1981. "A Methodology for Accurate Software Test Specification and Auditing." In *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 301-325. North-Holland. **

[Cha87] Cha, S.D., N.G. Leveson, T.J. Shimeall, and J.C. Knight. 1987. "An Empirical Study of Software Error Detection Using Self-Checks." In *Proceedings 17th International Symposium on Fault-Tolerant Computing*, July, Pittsburgh, PA, 156-161. **

[Cha88] Cha, S.S., N.G. Leveson, and T.J. Shimeall. 1988. *Safety Verification in Murphy Using Fault Tree Analysis*. University of California.

[Chan73] Chanon, R.N. December 1973. *On a Measure of Program Structure*. Ph.D. diss., Carnegie-Mellon University. Also published in *Proceedings Programming Symposium*, G. Goos and J. Hartmanis (eds.), April 9-11, Paris, France, 9-16. New York: Springer-Verlag.

[Chan79] Chandy, K.M., and J. Misra. 1979. *An Axiomatic Proof Technique for Networks of Communicating Processes*.
University of Texas at Austin. Technical Report TR-98. **

[Chan84] Chan, M., and S. Yam. "A Program Testing Assistant for BASIC-PLUS." *ACM: Software Engineering Notes*, 9/2 (Apr 1984):89-103.

[Chan85] Chandrasekharan, M., B. Dasarathy, and Z. Kishimoto. "Requirements-Based Testing of Real-Time Systems: Modeling for Testability." *IEEE: Computer*, 18/4 (Apr 1985):71-80.

[Chan88] Chandy, K.M. and J. Misra. 1988. *Parallel Program Design: A Foundation*. Reading, MA: Addison Wesley. **

[Chan89] Chang C.K., Y.-F. Chang, L. Yang, C.-R. Chou, and J.-J. Chen "Modeling a Real-Time Multitasking System in a Timed PQ Net." *IEEE: Software*, 6/2 (Mar 1989):46-52.

[Chap79] Chapin, N. 1979. "A Measure of Software Complexity." In *Proceedings AFIPS National Computer Conference*, vol. 48, June 4-7, New York, NY, 995-1001. Arlington, VA: AFIPS Press.

[Chap82] Chapman, D. "A Program Testing Assistant." *ACM: Communications of the ACM*, 25/9 (Sep 1982):625-634.

[Chea78] Cheatham, T.E. Jr., and D.A. Washington. 1978. "Program Loop Analysis by Solving First Order Recurrence Relations." In *Proceedings SIAM-SIGSAM Computer Algebra Symposium*, May. **

[Chea79] Cheatham, T.E., G.H. Holloway, and J.A. Townley. "Symbolic Evaluation and the Analysis of Programs." *IEEE: Transactions on Software Engineering*, 5/4 (Jul 1979):402-417.

[Cheh81] Cheheyl, M., et al. "Verifying Security." *ACM: Computing Surveys*, 13/3 (Sep 1981):279-340.

[Chen75] Chen, W.T., and C.V. Ramamoorthy. 1975. "Toward Automation of Test Data Generation." In *Proceedings International Computer Symposium*, Taipei, Taiwan. **

[Chen76] Chen, W.T. February 1976. *Toward Automated Validation of Computer Programs*. Ph.D. diss., University of California at Berkeley. **

[Chen78a] Chen, E.T. "Program Complexity and Programmer Productivity." *IEEE: Transactions on Software Engineering*, 4/3 (May 1978):187-194.

[Chen78b] Chen, L., and A. Avizienis. 1978. "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation." In *Proceedings 8th International Conference on Fault-Tolerant Computing*, June, Toulouse, France, 3-9. **

[Chen81] Chen, E., and M.V. Zelkowitz. 1981. "Use of Cluster Analysis to Evaluate Software Engineering Methodologies." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 117-123. Washington, DC: IEEE Computer Society Press.

[Chen83] Chen, B., and R.T. Yeh. "Formal Specification and Verification of Distributed Systems." *IEEE: Transactions on Software Engineering*, 9/11 (Nov 1983):710-722.

[Cher79] Cherniavsky, J.C. "On Finding Test Data Sets for Loop Free Programs." *Information Processing Letters*, 8/2 (Feb 1979):106-107. **

[Cher80a] Cherniavsky, J.C., W.R. Adrion, and M.A. Branstad. 1980. "The Role of Programming Environments in Software Quality Assurance." In *Proceedings National Electronics Conference, 13th Annual Alisomar Conference on Circuits, Systems and Computers*, Vol. 34, October 27-29, Chicago, IL, 468-472. Long Beach, CA: IEEE Computer Society.

[Cher80b] Cherniavsky, J.C., W.R. Adrion, and M.A. Branstad. 1980. "The Role of Testing Tools and Techniques in the Procurement of Quality Software and Systems." *Proceedings National Electronics Conference, 13th Annual Alisomar Conference on Circuits, Systems and Computers*, Vol. 34, October

27-29, Chicago, IL, 309-313. Long Beach, CA: IEEE Computer Society.

[Cher86] Cherniavsky, J.C., and C.H. Smith. 1986. "A Theory of Program Testing with Applications." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 110-119. Washington, DC: IEEE Computer Society Press.

[Cher87a] Cherniavsky, J.C. "Computer Systems as Scientific Theories: A Popperian Approach to Testing." In *Proceedings 5th Annual Pacific Northwest Software Quality Conference: Effective Software Practices*, October, Portland, OR, 397-308. **

[Cher87b] Cherniavsky, J.C., and C.H. Smith. "A Recursion Theoretic Approach to Program Testing." *IEEE: Transactions on Software Engineering*, 13/7 (Jul 1987):777-784.

[Cher88] Cherniavsky, J.C., and R. Statman. 1988. "Testing: An Abstract Approach." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 38-44. Washington, DC: IEEE Computer Society Press.

[Ches77] Chester, D.L., and R.T. Yeh. 1977. "Software Development by Evaluation of System Designs." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 435-441. Long Beach, CA: IEEE Computer Society Press.

[Choq85] Choquet, N., L. Fribourg, and A. Mauboussin. 1985. "Runnable Protocol Specifications using the Logic Interpreter SLOG." In *Proceedings 5th International Workshop on Protocol Specification, Verification, and Testing*, June, Toulouse, France. **

[Choq86] Choquet, N. 1986. "Test Data Generation using a Prolog with Constraints." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 132-141. Washington, DC: IEEE Computer Society Press.

[Chow78] Chow, T.S. "Testing Software Design Modeled by Finite-State Machines." *IEEE: Transactions on Software Engineering*, 4/3 (May 1978):178-187.

[Chri81] Christensen, K., G.P. Fitsos, and C.P. Smith. "A Perspective on Software Science." *IBM Systems Journal*, 20/4 (1981):372-387.

[Chry78] Chrysler, E. "Some Basic Determinants of Computer Programming Productivity." *ACM: Communications of the ACM*, 21/6 (Jun 1978):472-483.

[Chur82] Church, V.E., D.N. Card, F.E. McGarry, et al. August 1982. *Guide to Data Collection*. Greenbelt, MD: NASA Software Engineering Laboratory. Report SEL-81-101. **

[Chur86] Church, V.E., D.N. Card, W.W. Agresti, and Q.L. Jordan. "An Approach for Assessing Software Prototypes." *ACM: Software Engineering Notes*, 11/3 (Jul 1986):65-76.

[Chus87] Chusho, T. "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing." *IEEE: Transactions on Software Engineering*, 13/5 (May 1987):509-517.

[Cinl75] Cinlar, E. 1975. *Introduction to Stochastic Processes*. Englewood Cliffs, NJ: Prentice-Hall.

[Clar76a] Clarke, L.A. 1976. *Test Data Generation and Symbolic Execution of Programs as an Aid to Program Validation*. Ph.D. diss., University of Colorado at Boulder.

[Clar76b] Clarke, L.A. "A System to Generate Test Data and Symbolically Execute Programs." *IEEE: Transactions on Software Engineering*, 2/3 (Sep 1976):215-222.

[Clar78a] Clarke, L.A. 1978. "Testing: Achievements and Frustrations." In *Proceedings 2nd International Computer Software and Applications Conference*, November 13-16, Chicago, IL, 310-314. Long Beach, CA: IEEE Computer Society Press.

[Clar78b] Clarke, L.A. September 1978. "Automatic Test Data Selection Techniques." In *Infotech State of the Art Report on Software Testing*, Vol. 2, 43-64. **

[Clar81a] Clarke, L.A., and D.J. Richardson. 1981. "Symbolic Evaluation Methods - Implementations and Applications." In *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 65-102. Amsterdam: North Holland. **

[Clar81b] Clarke, E.M., and E.A. Emerson. 1981. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic*. Harvard University. Technical Report TR-12-81. **

[Clar81c] Clarke, L.A., and D.J. Richardson. 1981. "Symbolic Evaluation Methods for Program Analysis." In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones (eds.), 264-300. Englewood Cliffs, NJ: Prentice Hall. **

[Clar82] Clarke, L.A., and D.J. Richardson. 1982. "Reliable Test Data Selection Strategies – An Integrated Approach." In *Proceedings 4th Israel Conference on Quality Assurance*, October, Israel. **

[Clar83a] Clarke, L.A., and D.J. Richardson. 1983. "A Rigorous Approach to Error-Sensitive Testing." In *Proceedings IEEE 16th Hawaii International Conference on System Sciences*, January, Honolulu, HA. **

[Clar83b] Clarke, L.A., and D.J. Richardson. 1983. "The Application of Error-Sensitive Testing Strategies to Debugging." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):45-52. Baltimore, MD: ACM Order Department.

[Clar84] Clarke, L.A., and D.J. Richardson. 1984. "Symbolic Evaluation – An Aid to Testing and Verification." In *Software Validation*, H.L. Hausen (ed.), 141-166. North Holland.

[Clar85a] Clarke, L.A., A. Podgurski, D.J. Richardson, and S.J. Zeil. 1985. "A Comparison of Data Flow Path Selection Criteria." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 244-251. Washington, DC: IEEE Computer Society Press.

[Clar85b] Clarke, L.A., and D.J. Richardson. "Applications of Symbolic Evaluation." *Journal of Systems and Software*, 5/1 (1985):15-35.

[Clar86a] Clarke, L.A., A. Podgurski, D.J. Richardson, and S.J. Zeil. 1986. "An Investigation of Data Flow Path Selection Criteria." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 23-32. Washington, DC: IEEE Computer Society Press.

[Clar86b] Clarke, L.A., D.J. Richardson, and S.J. Zeil. September 1986. *Ada Symbolic Testing Techniques*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-86-141. **

[Clar86c] Clarke, L.A., Wileden, J.C., and A.L. Wolf. 1986. "GRAPHITE: A Meta-Tool for Ada Environment Development." In *Proceedings IEEE Conference on Ada Applications and Environments*, April 8-10, Miami Beach, FL, 81-90.

[Clar86d] Clarke, E.M., E.A. Emerson, and A.P. Sistla. 1980. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic." *ACM: Communications of the ACM*, 8/2 (Apr 1986):244-263. **

[Clar88a] Clarke, L.A., D.J. Richardson and S.J. Zeil. "Team: A Support Environment for Testing, Evaluation, and Analysis." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 28-30, Boston, MA, 121-129.

[Clar88b] Clarke, L.A., and S.J. Zeil. January 1988. *An Advanced Testing System for Ada, System Description and Design*. University of Massachusetts.

[Clem84] Clements, P.C., R.A. Parker, D.L. Parnas, J. Shore. and K.H. Britton. June 1984. *A Standard Organization for Specifying Abstract Interfaces*. Washington, DC: Naval Research Laboratory. NRL Report 8815. **

[Coch50] Cochran, W.G., and G.M. Cox. 1950. *Experimental Design*. New York: John Wiley & Sons.

[Coch53] Cochran, W.G. 1953. *Sampling Techniques*. New York: John Wiley & Sons.

[Coff87] Coffman, M.L. 1987. *Validation of Ada Package Interfaces*. M.S. thesis, Arizona State University. **

[Cohe77] Cohen, J., and N. Carpenter. "A Language for Inquiring about the Run-Time Behavior of Programs." *Software Practice and Experience*, 7/4 (Jul-Aug 1977):445-460.

[Cohe78] Cohen, E.A. 1978. *A Finite Domain-Testing Strategy for Computer Program Testing*. Ph.D. diss., Ohio State University. **

[Cohe82] Cohen, D., W. Swartout, and R.M. Balzer. "Using Symbolic Execution to Characterize Behavior." *ACM: SIGSOFT Software Engineering Notes*, 7/5 (Dec 1982):25-32.

[Cole85] Coles, R., et al. November 1985. *Software Reporting Metrices*. Mitre ESD. Report MTR 9650, Revision 2. **

[Come79] Comer, D., and M. H. Halstead. "A Simple Experiment in Top-Down Design." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):105-129.

[Conk86] Conklin, E.J. "LEONARDO: Design Environment of the 1990s." *RCA Engineer*, (Jan-Feb 1986). **

[Conk88] Conklin, E.J., and M. Begeman. January 1988. *gIBIS: A Hypertext Tool for Team Design Deliberation*. MCC. Technical Report STP-016-88. **

[Conn87] Conn, R. 1987. "The Ada Repository." In *Proceedings 32nd IEEE Computer Society International Conference*, February 23-27, San Francisco, CA, 372-375. Washington, DC: IEEE Computer Society Press.

[Conr85] Conradi, R., and D. Svanaes. January 1985. *FORTVER - A Tool for Documentation and Error Diagnosis of FORTRAN-77 Programs*. University of Trondheim. Technical Report 1/85. **

[Cont81] Conte, S.D. 1981. *The Software Science Language Level Metric*. Purdue University. Technical Report CSD-TR-373. **

[Cont85] Conti, R.A. 1985. "Debugging Ada Tasking." In *Proceedings 3rd Annual National Conference on Ada Technology*, 72-81. **

[Cont86] Conte, S.D., H.E. Dunsmore, and V.Y. Shen. 1986. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings Publishing Co.

[Cook80] Cook, J.F., and F.E. McGarry. December 1980. *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-80-007. **

[Cook81] Cook, J.F., and F.E. McGarry. February 1981. *Cost Reliability Estimation Models (CAREM) User's Guide*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-81-008. **

[Cook82] Cook, M.L. "Software Metrics: An Introduction and Annotated Bibliography." *ACM: SIGSOFT Software Engineering Notes*, 7/2 (Apr 1982):41-60.

[Corn76] Cornell, L., and M.H. Halstead. 1976. *Predicting the Number of Bugs Expected in a Program Module*. Purdue University. Technical Report CSD-TR-205. **

[Coul83] Coulter, N.S. "Software Science and Cognitive Psychology." *IEEE: Transactions on Software Engineering*, 9/2 (Mar 1983):166-171.

[Cox81] Cox, P.R. 1981. "Specification of a Regression Test for a Mini Computer Operating System." In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March:29-32.

[Crai86] Craigen, D. 1986. "Some Comments on Program Verification Systems." In *Proceedings Symposium on Safety and Security*, October 20-24, Glasgow, Scotland. Also published as I.P Sharp Associates, Technical Report TR-86-5420-02, November 1986. **

[Crai87a] Craigen, D. 1987. "Strengths and Weaknesses of Program Verification Systems." In *Proceedings 1st European Software Engineering Conference*, September 9-11, Strasbourg, France. Springer-Verlag. **

[Crai87b] Craigen, D. October 1987. *The Low Water Mark: An m-EVES Solution*. I.P. Sharp Associates. Working Paper 153. **

[Crai87c] Craigen, D. November 1987. *A Description of m-Verdi*. I.P. Sharp Associates. Technical Report TR-87-5420-02. **

[Crai87d] Craigen, D., and M. Saaltink. November 1987. *An m-Verdi User's Guide*. I.P. Sharp Associates. Technical Report TR-87-5420-12. **

[Crai88a] Craigen, D. 1988. "An Application of the m-EVES Verification System." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 21-36. Washington, DC: IEEE Computer Society Press.

[Crai88b] Craigen, D., S. Kromodimoeljo, I. Meisels, A. Neilson, B. Pase, and M. Saaltink. 1988. "m-EVES: A Tool for Verifying Software." In *Proceedings 10th International Conference on Software Engineering*, April 11-15, Singapore. Washington, DC: IEEE Computer Society Press.

[Crow85a] Crow, J., D. Denning, P. Ladkin, M. Melliar-Smith, J. Rushby, R. Schwartz, R. Shostak, and F.W. von Henke. November 1985. *SRI Verification System Version 2.0 User's Guide*. Menlo Park, CA: SRI International Computer Science Laboratory. **

[Crow85b] Crow, J., D. Denning, P. Ladkin, M. Melliar-Smith, J. Rushby, R. Schwartz, R. Shostak, and F.W. von Henke. 1985. *SRI Verification System Version 1.8 Specification Language Description*. Menlo Park, CA: SRI International Computer Science Laboratory. **

[Crui80] Cruickshank, R.D., and J.E. Gaffney. 1980. "Measuring the Development Process: Software Design Coupling and Strength Matrices." In *Proceedings 5th Annual Software Engineering Workshop*, November, Greenbelt, MD. NASA/GSFC. **

[Curr76] Curry R.W. 1976. "A Measure to Support Calibration and Balancing of the Effectiveness of Software Engineering Tools and Techniques." *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press.

[Curr83] Currit, P.A. 1983. "Cleanroom Certification Model." In *Proceedings 8th Annual Software Engineering Workshop*, November, Greenbelt, MD. NASA/GSFC. **

[Curr86] Currit, P.A., M. Dyer, and H.D. Mills. "Certifying the Reliability of Software." *IEEE: Transactions on Software Engineering*, 12/1 (Jan 1986):3-11.

[Curt79a] Curtis, B.A., S.B. Sheppard, P. Milliman, M. Borst, and L.T. Love. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):95-104.

[Curt79b] Curtis, B., S. Sheppard, and P. Milliman. 1979. "Third Time Charm: Stronger Predictions of Programmer Performance by Software Complexity Metrics." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 356-360. Washington, DC: IEEE Computer Society Press. **

[Curt80] Curtis, B. "Measurement and Experimentation in Software Engineering." In *Proceedings of the IEEE*, 68/9 (Sep 1980), 1144-1157. **

[Curt81] Curtis, B. 1981. "The Measurement of Software Quality and Complexity." In *Software Metrics: An Analysis and Evaluation*, A.J. Perlis, F.G. Sayward, and M. Shaw (eds.). Cambridge, MA: MIT Press.

[Curt83] Curtis, B. 1983. "Cognitive Science of Programming." In *Proceedings 6th Minnowbrook Workshop on Software Performance Evaluation*, July 19-22, Blue Mountain Lake, NY. **

[DACS79a] *Quantitative Software Models*. Griffiss Air Force Base, NY: Data and Analysis Center for Software, 1979. **

[DACS79b] Data and Analysis Center for Software (DACS). October 1979. *The DACS Glossary: A Bibliography of Software Engineering Terms*. Rome Air Development Center: Griffiss Air Force Base.

[DACS85] *Software Life Cycle Tools Directory*. Data and Analysis Center for Software, ITT Research Institute, March 1985. **

[DOD88a] DoD Standard-2167A. Defense Systems Software Development. 29 February 1988.

[DODD86a] DoD Directive 5000.3. March 12, 1986. *Test and Evaluation*.

[DODD86b] DoD Directive 5000.3-M-1. October 1986. *Test and Evaluation Master Plan Guidelines*.

[DODD87] DoD Directive 5000.3-M-3. November 1987. *Software Test and Evaluation Manual*.

[DODS86] DoD Standard STD-2168. 1 August 1986. *Defense System Software Quality Program* (draft).

[Dahl72] Dahl, O.J., E.W. Dijkstra, and C.A.R. Hoare. 1972. *Structured Programming*. Academic Press.

[Dahl78] Dahl, O.J. May 1978. *Can Program Proving be Made Practical?* University of Oslo, Institute of Informatics. **

[Dahl79a] Dahl, O.J. August 1979. *Time Sequences as a Tool for Describing Program Behavior*. University of Oslo. Research Report in Informatics 48. **

[Dahl79b] Dahll, G., and J. Lahti. 1980. "An Investigation of Methods for Production and Verification of Highly Reliable Software." In *Proceedings Safety of Computer Control Systems (SAFECOM) '79*, L. Lauber (ed.), , 75-79. New York: Pergamon. **

[Dale86] Dale, C. 1986. "Software Reliability Models." In *Reliability: State of the Art*, A. Bendell and P. Mellor (eds.), 31-44. Oxford: Pergamon Infotech. **

[Daly77] Daly, E.B. "Management of Software Development." *IEEE: Transactions on Software Engineering*, 3/3 (May 1977):229-242.

[Darr78] Darringer, J.A., and J.C. King. "Applications of Symbolic Execution to Program Testing." *IEEE: Computer*, 11/4 (Apr 1978):51-60.

[Davi77] Davis, P.J. "Proof, Completeness, Transcendentals, and Sampling." *Journal of the ACM*, 24/2 (Apr 1977):298-310.

[Davi81] Davis, M.D., and E.J. Weyuker. 1981. *Pseudo-Oracles for Nontestable Programs*. Courant Institute of Mathematical Sciences. **

[Davi82a] Davis, M.D. 1982. *Computability and Unsolvability*. New York: Dover Publications, Inc. **

[Davi82b] Davis, R.E. 1982. "Runnable Specifications as a Design Tool." in *Logic Programming*, K.L. Clark and S.-A. Tarnlund (eds.), 141-149. New York: Academic Press.

[Davi83a] Davis, M.D., and E.J. Weyuker. 1983. *Computability, Complexity, and Languages*. New York: Academic Press.

[Davi83b] Davis, M.D., and E.J. Weyuker. "A Formal Notion of Program-Based Test Data Adequacy." *Information and Control*, 56/1-2 (Jan-Feb 1983):52-71.

[Davi85] Davis, C.L., and E.A. Ireland. "Software Reliability and Quality, A User's View." In *Proceedings IEEE Global Telecommunications Conference*, December 2-5, New Orleans, LA, 69-72. Piscataway, NJ: IEEE Service Center. **

[Davi88a] Davis, J.S., and R.J. LeBlanc. "A Study of the Applicability of Complexity Measures." *IEEE: Transactions on Software Engineering*, 14/9 (Sep 1988):1366-1371.

[DeFr85] De Francesco, N., D. Latella, and G. Vaglini. 1985. "An Interactive Debugger for a Concurrent Language." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 320-325. Washington, DC: IEEE Computer Society Press.

[DeMi77] DeMillo, R.A., and R.J. Lipton. May 1977. *A Probabilistic Remark on Algebraic Program Testing*. Georgia Institute of Technology. Also published in *Information Processing Letters*, 7/4 (Jun 1978):193-195.

[DeMi78] DeMillo, R.A., R.J. Lipton, and F.G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." *IEEE: Computer*, 11/4 (Apr 1978): 34-41.

[DeMi79a] DeMillo, R.A., R.J. Lipton, and A.J. Perlis. "Social Processes and Proofs of Theorems and Programs." *ACM: Communications of the ACM*, 22/5 (May 1979):271-280.

[DeMi79b] DeMillo, R.A., F.G. Sayward, and R.J. Lipton. 1979. "Program Mutation: A New Approach to Program Testing." In *Infotech State of the Art Report on Program Testing*, 107-126. Infotech International. **

[DeMi81] DeMillo, R.A., D.E. Hocking, and M.J. Merrit. 1981. *A Comparison of Some Reliable Test Data Generation Procedures*. Georgia Institute of Technology. Technical Report GIT-ICS-81/08. **

[DeMi86a] DeMillo, R.A. October 1986. *Functional Capabilities of a Test and Evaluation Subenvironment in an Advanced Software Engineering Environment*. Georgia Institute of Technology. Report GIT-SERC-86/07. **

[DeMi86b] Demillo, R.A., and E.H. Spafford. 1986. "The Mothra Software Testing Environment." In *Proceedings 11th Annual Software Engineering Workshop*, December, Greenbelt, MD. NASA/GSFC. **

[DeMi87a] DeMillo, R.A., W.M. McCracken, R.J. Martin, and J.F. Passafiume. 1987. *Software Testing and Evaluation*. Menlo Park, CA: The Benjamin/Cummings Publishing Co.

[DeMi87b] DeMillo, R.A., D.S. Guindi, K.N. King, and W.M. McCracken. 1987. *An Overview of the Mothra Testing Environment*. Purdue University. Technical Report SERC-TR-3-P.

[DeMi87c] DeMillo, R.A., and A.J. Offutt. 1987. *Constraint Based Automatic Test Data Generation*. Purdue University. Technical Report SERC-TR-5-P.

[DeMi87d] DeMillo, R.A., D. Guindi, K.N. King, E.W. Krauser, W.M. McCracken. A.J. Offutt, and E.H. Spafford. 1987. *Mothra Internal Documentation, Version 1.0*. Georgia Institute of Technology. Technical Report GIT-SERC-87/10.

[DeMi88a] DeMillo, R.A., D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. 1988. "An Extended Overview of the Mothra Software Testing Environment." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 142-151. Washington, DC: IEEE Computer Society Press.

[DeMi88b] DeMillo, R.A., R.J. Martin, and R.N. Meeson. September 1988. *Strategy for Achieving Ada-Based High Assurance Systems*. Alexandria, VA: Institute for Defense Analyses. Draft IDA Paper P-2143.

[DeRe76] DeRemer, F., and H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small." *IEEE: Transactions on Software Engineering*, 2/2 (Jun 1976):80-86.

[Deck82a] Decker, W.J., and W.A. Taylor. May 1982. *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1)*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-78-102. **

[Deck82b] Decker, W.J., W.A. Taylor, and E.J. Smith. February 1982. *Software Engineering Laboratory (SEL) Compendium of Tools*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-81-107. **

[Deke81] Dekert, J.L.F. 1981. "APL and Halstead's Theory of Software Metrics." In *APL81 Conference Proceedings (APL Quote Quad)*, Vol. 12, September, 89-93. ACM. **

[Dela88] Delaney, R.P., and L.F. Summerill. 1988. "Ada Software Metrics." In *Proceedings 6th National Conference on Ada Technology*, March 14-17, Arlington, VA, 24-31. Washington, DC: ACM Ada Technical Committee.

[Dems82] Demshki, M., D. Ligett, B. Linn, G. McCluskey, and R. Miller. June 1982. *Wang Institute Cost Model (WICOMO) Tool User's Manual*. Tyngsboro, MA: Wang Institute for Graduate Studies. **

[Denn78] Denning, P.J., and J.P. Buzen. "Operational Analysis of Queueing Network Models." *ACM: Computing Surveys*, 10/3 (Sep 1978):225-245.

[Dera85] Deransart, P., and J. Maluszynski. "Relating Logic Programs and Attribute Grammars." *Journal of Logic Programming*, 2/2 (Jul 1985):119-155. **

[Deut73] Deutsch, L.P. May 1973. *An Interactive Program Verifier*. Ph.D. diss., University of California at Berkeley. **

[DiMa85] Di Maio, A., S. Ceri, and S.C. Reghizzi. 1985. "Execution Monitoring and Debugging Tool for Ada Using Relational Algebra." In *Proceedings SIGAda International Conference*, May, Paris, France. Published in *ACM: Ada Letters*, V/2 (Sep-Oct 1985).

[DiVi82] DiVito, B.L. 1982. *Verification of Communication Protocols and Abstract Process Models*. University of Texas. Technical Report ICSCA-CMP-25. **

[Dijk68] Dijkstra, E.W. 1968. "A Constructive Approach to the Problem of Program Correctness." *BIT* 8/3 (1968). **

[Dijk76a] Dijkstra, E.W. 1976. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall.

[Dijk76b] Dijkstra, E.W. 1976. "Complexity Controlled by Hierarchical Ordering of Functions and Variability." In *Software Engineering Concepts and Techniques, Proceedings NATO Conference*, October 7-11, Garmisch, Germany, 114-116. New York: Van Nostrand Reinhold.

[Dill81] Dillon, L.K. May 1981. *Constraint Management in the ATTEST System*. University of Massachusetts. Technical Report 81-9. **

[Dill84] Dillon, L.K. September 1984. *Analysis of Distributed Systems using Constrained Expressions*. Ph.D. diss., University of Massachusetts. Technical Report TR-84-18. **

[Dill85] Dillon, L.K., G. Avrunin, and J. Wileden. July 1985. *Constrained Expressions: A General Technique for Describing Behavior of Concurrent Systems*. University of Massachusetts. Technical Report TR-85-27. **

[Dill86] Dillon, L.K. October 1986. *Overview of the Constrained Expression Design Language*. University of California at Santa Barbara. Technical Report TRCS86-21. **

[Dill87] Dillon, L.K. October 1987. *Verification of Ada Tasking Programs Using Symbolic Execution, Part I: Partial Correctness*. University of California at Santa Barbara. **

[Dill88a] Dillon, L.K., R.A. Kemmerer, and L.J. Harrison. 1988. "An Experience with Two Symbolic Execution-Based Approaches to Formal Verification of Ada Tasking Programs." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 114-121. Washington, DC: IEEE Computer Society Press.

[Dill88b] Dillon, L.K. May 1988. "Symbolic Execution-Based Verification of Ada Tasking Programs." In *Proceedings Ada Europe Conference*, June, Munich, Germany . New York: Cambridge University Press.

[Dill88c] Dillon, L. ' G.S. Avrunin, and J.C. Wileden. "Constrained Expressions: Toward Broad Applicability of A..alysis Methods for Distributed Software Systems." *ACM: Transactions on Programming Languages and Systems*, 10/3 (Jul 1988):374-402.

[Dirc81] Dircks, H.F. 1981. "SOFCOST: Grumman's Software Cost Estimating Model." In *Proceedings IEEE NAECON 1981*, May. **

[Doer85] Doerflinger, C.W., and V.R. Basili. "Monitoring Software Development Through Dynamic Variables." *IEEE: Transactions on Software Engineering*, 11/9 (Sep 1985):978-985. Also published in *Proceedings 7th International Computer Software and Applications Conference*, Novemer 7-11, Chicago, IL, 434-445. Los Angeles, CA: IEEE Computer Society.

[Doub87] Doubleday, D.L. August 1987. *ASAP: An Ada Static Source Code Analyzer Program*. University of Maryland. Technical Report TR-1895. **

[Down85a] Downs, T. "An Approach to the Modeling of Software Testing with Some Applications." *IEEE: Transactions on Software Engineering*, 11/4 (Apr 1985):375-386.

[Down85b] Downs, T. "A Review of Some of the Reliability Issues in Software Engineering." *Journal of Electrical and Electronics Engineering. Australia - IE Aust. & IREE Aust.*, 5/1 (Mar 1985). **

[Down86] Downs, T. "Extensions to an Approach to the Modeling of Software Testing with Some Performance Comparisons." *IEEE: Transactions on Software Engineering* 12/9 (Sep 1986):979-987.

[Drap66] Draper, N.R., and H. Smith. 1966. *Applied Regression Analysis*. Wiley Series in Probability and Mathematical Statistics. New York: John Wiley & Sons, Inc.

[Dsso85] Dssouli, R., and G.v. Bochmann. 1985. "Error Detection with Multiple Observers." In *Proceedings 5th IFIP Workshop on Protocols*, June, 483-494. North-Holland. **

[Dsso86] Dssouli, R. December 1986. *Etude des Methodes de Test pour les Implantations de Protocoles de Communication Basees sur Les Specifications Formelles*. Ph.D. thesis, Montreal University. **

[Ducl82] Duclos, L.C. December 1982. *Simulation Cost Model for the Life-Cycle of the Software Product: A Quality Assurance Approach*. Ph.D. diss., University of Southern California. **

[Duke89] Duke, E.L. "V&V of Flight and Mission-Critical Software." *IEEE: Software*, 6/3 (May 1989):39-45.

[Duma83] Dumas, R.L. September 1983. *Final Report: Software Acquisition Resource Expenditure (SARE) Data Collection Methodology*. MITRE Corp. Technical Report MTR 9031. **

[Dunc78] Duncan, A.G. 1978. "Test Grammars: A Method for Generating Program Test Data." *Digest IEEE Workshop on Software Testing and Test Documentation*, December 18-20, Ft. Lauderdale, FL, 270-283. IEEE Computer Society Technical Committee on Software Engineering. **

[Dunc81] Duncan, A.G., and J.S. Hutchison. 1981. "Using Attribute Grammars to Test Designs and Implementations." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 170-178. Washington, DC: IEEE Computer Society Press.

[Dunh81] Dunham, J.R., and J.C. Knight. November 1981. *Production of Reliable Flight-Crucial Software*. NASA. Technical Report 2222. **

[Dunh83] Dunham, J.R., and E. Kruesi. "The Measurement Task Area." *IEEE: Computer*, 16/11 (Nov 1983):47-54.

[Dunh85] Dunham, J.R., and J.L Pierce. March 1985. *An Experiment in Software Reliability*. Greenbelt, MD: NASA/GSFC. NASA Contractor Report 172553. **

[Dunh86] Dunham, J.R. "Experiments in Software Reliability: Life-Critical Applications." *IEEE: Transactions on Software Engineering*, 12/1 (Jan 1986):110-124.

[Dunn74] Dunn, O.J., and V.A. Clark. 1974. *Applied Statistics: Analysis of Variance and Regression*. New York: John Wiley & Sons.

[Dunn82] Dunn, R., and R. Ullman. 1982. *Quality Assurance for Computer Software*. New York: McGraw Hill.

[Dunn84] Dunn, R.H. 1984. *Software Defect Removal*. New York: McGraw-Hill.

[Duns77] Dunsmore, H.E., and J.D. Gannon. 1977. "Experimental Investigation of Programming Complexity." In *Proceedings ACM/NBS 16th Annual Technical Symposium: Systems and Software*, June 2, Gaithersburg, MD, 117-125. Washington, DC: Washington DC Chapter of the ACM.

[Duns78a] Dunsmore, H.E. July 1978. *The Influence of Programming Factors on Programming Complexity*. Ph.D. diss., University of Maryland. **

[Duns78b] Dunsmore, H.E. 1978. "Programming Factors–Language Features that Help Explain Programming Complexity." In *Proceedings 31st ACM Annual National Computer Conference*, December 4-6, Washington, DC, 554-560. New York: Association for Computing Machinery.

[Duns80] Dunsmore, H.E., and J.D. Gannon. "Analysis of the Effects of Programming Factors on Programming Effort." *Journal of Systems and Software*, 1/2 (Feb 1980):265-273. **

[Duns83] Dunsmore, H.E. 1983. "Software Metrics: An Overview of an Evolving Methodology." In *Proceedings Symposium Empirical Foundations of Information and Software Science*, November 3-5. Atlanta, GA. Published in *Information Processing and Management*, (1983). **

[Dura78] Duran, J.W., and J.J. Wiorkowski. 1978. "Toward Models for Probabilistic Program Correctness." In *Proceedings ACM Software Quality Assurance Workshop*, November 15-17, San Diego, CA, 39-44. New York: Association for Computing Machinery.

[Dura80] Duran, J.W., and J.J. Wiorkowski. "Quantifying Software Validity by Sampling." *IEEE: Transactions on Reliability*, R-29/2 (Jun 1980):141-144.

[Dura81a] Duran, J.W., and S.C. Ntafos. 1981. "A Report on Random Testing." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 179-183. Washington, DC: IEEE Computer Society Press. Also published in *IEEE Transactions on Software Engineering*, 10/4 (Jul 1984):438-444.

[Dura81b] Duran, J.W., and J.J. Wiorkowski. "Capture-Recapture Sampling for Estimating Software Error Content." *IEEE: Transactions on Software Engineering*, 7/1 (Jan 1981):147-148.

[Duva80] Duvall, L., J. Martens, D. Swearingen, and J. Donahoo. 1980. "Data Needs for Software Reliability Modeling." In *Proceedings Annual Reliability and Maintainability Symposium*, 200-208.

[Dyer80] Dyer, M., R.C. Linger, H.D. Mills, D. O'Neill, and R.R. Quinnan. "The Management of Software Engineering, Part IV: Software Development Practices." *IBM Systems Journal*, 19/4 (1980):451-465.

[Dyer81a] Dyer, M. October 16, 1981. *Cleanroom Project Management Data*. Bethesda, MD: IBM Federal Systems Division. Internal Memo to H.D. Mills. **

[Dyer81b] Dyer, M., and H.D. Mills. 1981. "The Cleanroom Approach to Reliable Software Development." In *Proceedings Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting: Production of Reliable Flight-Crucial Software*, November 2-4, Research Triangle Institute, NC. **

[Dyer81c] Dyer, M., and H.D. Mills. 1981. "Cleanroom Software Development." In *Proceedings 6th Annual Software Engineering Workshop*, December, Grenbelt, MD. NASA/GSFC. **

[Dyer82a] Dyer, M. 1982. *An Approach to Statistical Testing for Cleanroom Software Development*. Bethesda, MD: IBM Federal Systems Division. Technical Report 86.0002. **

[Dyer82b] Dyer, M., and H.D. Mills. 1982. "Developing Electronic Systems with Certifiable Reliability." In *Proceedings NATO Conference on Electronic Systems Effectiveness and life-Cycle Costing*, NATO Advanced Study Series. Springer-Verlag. **

[Dyer82c] Dyer, M. October 14, 1982. *Cleanroom Software Development Method*. Bethesda, MD: IBM Federal Systems Division. **

[Dyer83] Dyer, M. August 19, 1983. *Software Validation in the Cleanroom Development Method*. Bethesda, MD: IBM Federal Systems Division. Technical Report 86.0003. **

[Dyer85a] Dyer, M. 1985. "Software Verification Through Statistical Testing."

[Dyer85b] Dyer, M. 1985. "Software D[illegible]pment Under Statistical Quality Control." In ?? **

[EPI82] University of Texas. February 1982. *Formal Verification of a Communications Processor*. Final Report, Contract MDA 904-80-C-0481. **

[East72] Easterling, R.G. "Approximate Confidence Limits for System Reliability." *Journal of the American Statistical Association*, 67/337 (Mar 1972):220-222.

[Eckh85] Eckhardt, D.E., and L.D. Lee. January 1985. *A Theoretical Basis for the Analysis of Redundant Software Subject to Coincident Errors*. Hampton, VA: National Aeronautics and Space Administration. Technical Memorandum 86369. Also published in *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1511-1517.

[Eckm83a] Eckmann, S. March 1983. *Symbolic Execution of Concurrent Programs in Gypsy*. University of California at Santa Barbara. **

[Eckm83b] Eckmann, S., and R.A. Kemmerer. December 1983. *A User's Manual for the UNISEX System*. University of California at Santa Barbara. Revised April 1985. **

[Eckm84] Eckmann, S., and R.A. Kemmerer. April 1984. *Preliminary Inatest User's Manual*. University of California at Santa Barbara. **

[Eckm85] Eckmann, S., and R.A. Kemmerer. 1985. "INATEST: An Interactive Environment for Testing Formal Specifications." In *Proceedings 3rd Workshop on Formal Verification*, Pajaro Dunes, CA, February. Also published in *Software Engineering Notes*, 10/4 (Aug 1985). **

[Ehre73] Ehrenberger, W., and H. Schuller. 1973. "Proof of the Correct Performance of a Computerized Reactor Protection System." In *Proceedings 3rd European Seminar on Real-Time Programming*, May, Ispra. **

[Ehre76] Ehrenberger, W., G. Rauch, and K. Okroy. 1976. "Program Analysis–A Method for the Verification of Software for the Control of a Nuclear Reactor." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA, 611-616. Washington, DC: IEEE Computer Society Press.

[Ehre78] Ehrenberger, W., and K. Plogert. 1978. "Statistical Verification of Reactor Protection Software." In *Proceedings International Symposium on Nuclear Power Plant Control*, April, Cannes, France, paper 39. **

[Ehri85] Ehrig, H., and B. Mahr. 1985. *Fundamentals of Algebraic Specifications*. Berlin: Springer-Verlag. **

[Ehrl87] Ehrlich, W.K., and T.J. Emerson. 1987. "Modeling Software Failures and Reliability Growth During System Testing." In *Proceedings 9th International Conference on Software Engineering*, March 30 - Arpil 2, Monterey, CA, 72-82. Washington, DC: IEEE Computer Society Press.

[Ejio87] Ejiogu, L.O. "The Critical Issues of Software Metrics." *ACM: SIGPLAN Notices*, 22/3 (Mar 1987):59-63.

[Elme69] Elmendorf, W.R. "Controlling the Functional Testing of an Operating System." *IEEE: Transactions on System Sciences and Cybernetics*, SSC-5/4 (Oct 1969):284-290.

[Elme71] Elmendorf, W.R. "Disciplined Software Testing." In *Debugging in Large Systems*, R. Rustin (ed.), 137-140. Englewood Cliffs, NJ: Prentice-Hall.

[Elme73] Elmendorf, W.R. November 1973. *Cause-Effect Graphs in Functional Testing*. IBM Technical Report 00.2487.

[Elsh76a] Elshoff, J.L. "Measuring Commercial PL/1 Programs Using Halstead's Criteria." *ACM: SIGPLAN Notices*, 7/5 (May 1976):38-46. **

[Elsh76b] Elshoff, J.L. "An Analysis of Some Commercial PL/I Programs." *IEEE: Transactions on Software Engineering*, 2/2 (Jun 1976):113-120.

[Elsh78a] Elshoff, J.L. 1978. "A Review of Software Measurement Studies at General Motors Research Laboratories." In *Proceedings 2nd Software Life Cycle Measurement Workshop*. New York: IEEE Computer Society. **

[Elsh78b] Elshoff, J.L. "An Investigation into the Effects of the Counting Method Used on Software Science Measurements." **

[Elsh78c] Elshoff, J.L., and M. Marcotty. "On the Use of the Cyclomatic Number to Measure Program Complexity." *ACM: SIGPLAN Notices*, 13/12 (Dec 1978):29-40.

[Elsh84] Elshoff, J.L. 1984. "Characteristic Program Complexity Measures." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 288-293. Washington, DC: IEEE Computer Society Press.

[Elsp72a] Elspas, B., K.N. Levitt, R.J. Waldinger, and A. Waksman. "An Assessment of Techniques for Proving Program Correctness." *ACM: Computing Surveys*, 4/2 (Jun 1972):97-147.

[Elsp72b] Elspas, B., M.W. Green, K.N. Levitt, and R.J. Waldinger. 1972. *Research in Interactive Program Proving Techniques*. Menlo Park, CA: Stanford Research Institute. SRI Report 8398-II. **

[Elsp73] Elspas, B., K.N. Levitt, and R.J. Waldinger. 1973. *An Interactive System for the Verification of Computer Programs*. Menlo Park, CA: Stanford Research Institute. Final Report, SRI Project 1891. **

[Elsp74] Elspas, B. July 1974. *The Semiautomatic Generation of Inductive Assertions for Proving Program Correctness*. Menlo Park, CA: Stanford Research Institute. Interim Report, SRI Project 2686. **

[ElspXX] Elspas, B., M.W. Green, A. Korsak, and P. Wong. *Solving Nonlinear Inequalities Associated with Computer Program Paths*. Menlo Park, CA: Stanford Research Institute. Preliminary draft. **

[**Emde81**] Emden, M.H., and T.S.E. Maibaum. 1981. "Equations Compared with the Clauses for Specification of Abstract Data Types." In *Advances in Database Theory*, H. Gallaire, J. Minker, and J.M. Nicholas (eds.), 159-194. New York: Plenum Press.

[**Emer83**] Emerson, E.A., and J.Y. Halpern. " 'Sometimes' and 'Not' Revisited: On Branching versus Linear Time." In *Proceedings ACM Symposium on the Principles of Programming Languages*, January, Austin, TX, 127-140. **

[**Emer84**] Emerson, T.J. 1984. "A Discriminant Metric for Module Cohesion." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 294-303. Washington, DC: IEEE Computer Society Press.

[**Emer85**] Emerson, E.A., and C.-L. Lei. 1985. "Modalities for Model Checking: Branching Time Strikes Back." In *Proceedings ACM Symposium on the Principles of Programming Languages*, January, New Orleans, LA., 84-96. **

[**Endr75**] Endres, A. "An Analysis of Errors and Their Causes in System Programs." *IEEE: Transactions on Software Engineering*, 1/2 (Jun 1975):140-149.

[**Epp86**] Epp, E.C., and S.J. Zeil. December 1986. *ARIES: A Multi-Lingual Interpreter for a Tool-Fragment Environment*. University of Massachusetts. COINS Technical Report 86-57 (revised May 1987). **

[**Eric85**] Erickson, R.L. 1985. "Overview of Generic Telecommunications Software Reliability and Quality Requirements Proposed by BELLCORE." In *Proceedings IEEE Global Telecommunications Conference*, December 2-5, New Orleans, LA, 65-68. Piscataway, NJ: IEEE Service Center.

[**Evan83a**] Evangelist, W.M. "Software Complexity Metric Sensitivity to Program Structuring Rules." *ACM: The Journal of Systems and Software*, 3/6 (Nov 1983):231-243.

[**Evan83b**] Evangelist, W.M. "Relationships Among Computational, Software and Intuitive Complexity." *ACM: SIGPLAN Notices*, 18/12 (Dec 1983):57-59.

[**Evan84a**] Evangelist, W.M. 1984. "An Analysis of Control Flow Complexity." In *Proceedings 8th International Computer Software and Applications Conference*. Washington, DC: IEEE Computer Society.

[**Evan84b**] Evans, M.W. 1984. *Productive Software Test Management*. New York: John Wiley & Sons.

[**Evan84c**] Evangelist, W.M. 1984. "Program Complexity and Programming Style." In *Proceedings International Conference on Data Engineering*, 534-541.

[**Evan87**] Evans, W.E., and J.J. Marciniak. "Software Quality Metrics." In *Software Quality Assurance and Management*, 157-186. John Wiley & Sons. **

[**FIPS77**] *Guidelines for Benchmarking ADP Systems in the Competitive Procurement Environment*. Gaithersburg, MD: National Bureau of Standards/U.S. Department of Commerce. Federal Information Processing Standards Publication, May 1977. **

[**FSTC83**] *Software Tools Survey*. Falls Church, VA: Federal Software Testing Center, U.S. Office of Software Development. **

[**Faga74**] Fagan, M.E. December 1974. *Design and Code Inspections and Process Control in the Development of Programs*. IBM Corp. Technical Report TR-21-572. **

[**Faga76**] Fagan, M.E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM: Systems Journal*, 15/3 (1976):182-211.

[**Faga86**] Fagan, M.E. "Advances in Software Inspections." *IEEE: Transactions on Software Engineering* 12/7 (Jul 1986):744-751.

[**Fain85**] Fainter, R.G. June 1985. *AdaTAD-A Debugger for the Ada Multi-Task Environment*. Ph.D. diss., Virgina Tech. **

[**Fain86**] Fainter, R.G., and T.E. Lindquist. 1986. "Debugging Tasks with AdaTAD." In *Proceedings 1st International Conference on Ada for the NASA Space Station*. Jun , Houston, TX. **

[**Fair75**] Fairley, R.E. "An Experimental Program-Testing Facility." *IEEE: Transactions on Software Engineering*, 1/4 (Dec 1975):350-357.

[**Fair79**] Fairley, R.E. "ALADDIN: An Assembly Language Assertion Driven Debugging Interpreter." *IEEE: Transactions on Software Engineering*, 5/4 (Jul 1979):426-428.

[**Farr65**] Farr, L., and H.J. Zagorski. 1965. "Quantitative Analysis of Programming Cost Factors: A Progress Report." In *Proceedings 1965 ICC Symposium*, Rome. Published in "Economics of Automatic Data

Processing," A.B. Frielink (ed.), 167-177. Amsterdam: North-Holland. **

[Farr83] Farr, W.H. September 1983. *A Survey of Software Reliability Modeling and Estimation*. Dahlgren, VA: Naval Surface Weapons Center. Technical Report NSWC-TR-82-171.

[Farr88] Farr, W.H., O.D. Smith, and C.L. Schimmelpfenneg. 1988. "A PC Tool for Software Reliability Measurement." In *1988 Proceedings*. Institute of Environmental Sciences.

[Fava79] Favaro, J.M. 1979. "A FORTRAN Symbolic Executor Based on MACSYMA." In *Proceedings 2nd MACSYMA User's Conference*, June. **

[Feat89] Feather, M.S. "Constructing Specifications by Combining Parallel Elaborations." *IEEE: Transactions on Software Engineering*, 15/2 (Feb 1989).

[Feie80] Feiertag, R.J. January 1980. *A Technique for Proving Specifications are Multilevel Secure*. SRI International. Technical Report CSL109. **

[Feld89] Feldman, M.B., and M.L. Moran. "Validating a Demonstration Tool for Graphics-Assisted Debugging of Ada Concurrent Programs." *IEEE: Transactions on Software Engineering*, 15/3 (May 1989):305-313.

[Fern85] Fernandez, J.C., J.L. Richier, and J. Voiron. 1985. "Verification of Protocol Specifications using the CEDAR System." In *Proceedings 5th International Workshop on Protocol Specification, Verification, and Testing*, June, Toulouse, France. **

[Ferr77] Ferrentino, A.B., and H.D. Mills. 1977. "State Machines and Their Semantics in Software Engineering." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 242-251. Long Beach, CA: IEEE Computer Society Press.

[Fetz88] Fetzer, J.H. "Program Verification: The Very Idea." *ACM: Communications of the ACM*, 31/9 (Sep 1988):1048-1063.

[Feue79a] Feuer, A.R., and E.B. Fowlkes. 1979. "Relating Computer Program Maintainability to Software Measures." In *Proceedings AFIPS National Computer Conference*, vol. 48, June 4-7, New York, NY, 1003-1012. Arlington, VA: AFIPS Press.

[Feue79b] Feuer, A.R., and E.B. Fowlkes. 1979. "Some Results from an Empirical Study of Computer Software." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 351-355. Washington, DC: IEEE Computer Society Press. **

[Fink83] Finkel, R.A., M.H. Solomon et al. April 1983. *Charlotte: Part IV of the First Report on the Crystal Project*. University of Wisconsin. Technical Report 501. **

[Fisc77] Fischer, K.F. 1977. "A Test Case Selection Method for the Validation of Software Maintenance Modifications." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 421-426. Long Beach, CA: IEEE Computer Society Press.

[Fits79] Fitsos, G.P. September 1979. *Software Science Counting Rules and Tuning Methodology*. IBM Santa Teresa Laboratory. Technical Report 03-075. **

[Fits80] Fitsos, G.P. January 1980. *Vocabulary Effects in Software Science*. IBM Santa Teresa Laboratory. Technical Report 03-082. **

[Fitz78a] Fitzsimmons, A.B., and T. Love. "A Review and Evaluation of Software Science." *ACM: Computing Surveys*, 10/1 (Mar 1978):3-18.

[Fitz78b] Fitzsimmons, A.B. 1978. "Relating the Presence of Software Errors to the Theory or Software Science." In *Proceedings IEEE 11th Hawaii International Conference on System Sciences*, January, Honolulu, HA. **

[Flon77] Flon, L. 1977. *On the Design and Verification of Operating Systems*. Ph.D. diss., Carnegie-Mellon University.

[Flon78a] Flon, L., and N. Suzuki. 1978. "Non-Determinism and Correctness of Parallel Programs." In *Proceedings Information Processing Working Conference on the Formal Description of Programming Concepts*, August 1-5, St. Andrews, Canada, 589-600. Amsterdam: North Holland.

[Flon78b] Flon, L., and N. Suzuki. November 1978. *Consistent and Complete Proof Rules for the Total Correctness of Parallel Programs*. Xerox Corp. Technical Report CSL-78-6. **

[Flon81] Flon, L., and N. Suzuki. "The Total Correctness of Parallel Programs." In *SIAM: Journal of Computers*, 10/2 (May 1981):227-246.

[Floy67] Floyd, R.W. 1967. "Assigning Meaning to Programs." In *Proceedings American Mathematical Society Symposium in Applied Mathematics*, vol. 19, 19-31. Providence, RI: American Mathematics Society. Also published in *ACM: Communications of the ACM*, 14/1 (Jan 1971):39-45. **

[Forg87] Forghani, B., and B. Sarikaya. November 1987. *CONTEST-FSM A Finite State Machine Based Test Generation Tool for Protocols*. Concordia University. Research Report. **

[Form77] Forman, E.H., and N.D. Singpurwalla. "An Empirical Stopping Rule for Debugging and Testing Computer Software." *Journal of the American Statistical Association*, 72/360 (Dec 1977):750-757.

[Form79] Duran, E.H., and N.D. Singpurwalla. "Optimal Time Intervals for Testing Hypotheses on Computer Software Errors." *IEEE: Transactions on Reliability*, R-28/3 (Aug 1979):250-253.

[Fosd74] Fosdick, L.D. March 1974. "BRNANL-A FORTRAN Program to Identify Basic Blocks in FORTRAN Programs." University of Colorado. Technical Report CU-CS-040-74. **

[Fosd76a] Fosdick, L.D., and L.J. Osterweil. "Data Flow Analysis in Software Reliability." *ACM: Computing Surveys*, 8/3 (Sep 1976):305-330.

[Fosd76b] Fosdick, L.D., and L.J. Osterweil. 1976. "The Detection of Anomalous Interprocedural Data Flow." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA. Washington, DC: IEEE Computer Society Press.

[Fost80] Foster, K.A. "Error Sensitive Test Cases Analysis (ESTCA)." *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):258-264.

[Fost83] Foster, K.A. "Comments on the Application of Error-Sensitive Testing Strategies to Debugging." *ACM: SIGSOFT Software Engineering Notes*, 8/5 (Oct 1983):40-42.

[Fost84] Foster, K.A. "Sensitive Test Data for Logical Expressions." *ACM: Software Engineering Notes*, 9/2 (Apr 1984):120-125.

[Fost85] Foster, K.A. "Revision of an Error Sensitive Test Rule." *ACM: Software Engineering Notes*, 10/1 (Jan 1985):62-67.

[Fran79] Francez, N., C.A.R. Hoare, D.J. Lehmann, and W.P. de Roever. "Semantics of Concurrency, Nondeterminism and Communication." *Journal of Computer and System Sciences*, 12 (1979). **

[Fran80] Francez, N. "Distributed Termination." *ACM: Transactions on Programming Languages and Systems*, 2/1 (Jan 1980):42-55.

[Fran85a] Frankl, P.G., S.N. Weiss, and E.J. Weyuker. 1985. "ASSET: A System To Select and Evaluate Tests." In *Proceedings IEEE Conference on Software Tools*, April, New York, 72-79.

[Fran85b] Frankl, P.G., and E.J. Weyuker. 1985. "A Data Flow Testing Tool." In *Proceedings SoftFair II: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, December 2-5, San Francisco, CA. **

[Fran86] Frankl, P.G., and E.J. Weyuker. 1986. "Data Flow Testing in the Presence of Unexecutable Paths." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 4-13. Washington, DC: IEEE Computer Society Press.

[Fran88] Frankl, P.G., and E.J. Weyuker. "An Applicable Family of Data Flow Testing Criteria." *IEEE: Transactions on Software Engineering*, 14/10 (Oct 1988):1483-1498.

[Freb79] Freburger K., and V.R. Basili. May 1979. *The Software Engineering Laboratory: Relationship Equations*. University of Maryland. Technical Report TR-764. **

[Frei79] Freiman, F.R., and R.D. Park. 1979. "PRICE Software Model–Version 3: An Overview." In *Proceedings IEEE-PINY Workshop on Quantitative Software Models*, October, 32-41. IEEE Cat. TH0067-9. **

[Freu84] Freudenberger, S.M, 1984. *On the Use of Global Optimization Algorithms for the Detection of Semantic Programming Errors*. Courant Institute of Mathematical Sciences. Technical Report NSO-24. **

[Frye81] Fryer, S., and D.M. Weiss. 1981. "Evaluation of the A-7E Software Requirements Document by Analysis of Change Data: Two years of Change Data." In *Proceedings 15th Annual Asilomar Conference on Circuits, Systems, and Computers*, November. **

[Fuji77] Fujii, M.S. 1977. "Independent Verification of Highly Reliable Programs." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 38-44. Long Beach, CA: IEEE Computer Society Press.

[Funa75] Funami, Y., and M.H. Halstead. 1975. *A Software Physics Analysis of Akiyama's Debugging Data*. Purdue University. Technical Report CSD-TR-144. Also publsihed in *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press. **

[Fung85] Fung, C.K-C. 1985. *A Methodology for the Collection and Evaluation of Software Error Data*. Ph.D. diss., Ohio State University.

[Furt81] Furtek, F.C., J.B. DeWolf, and P. Buchan. 1981. "DARTS: A Tool for Specification and Simulation of Real-Time Systems." In *Proceedings AIAA Conference on Computers in Aerospace*, October. **

[GE77a] *Fortran Test Procedure Language--Programmer Reference Manual*. Schenectady, NY: General Electric Co., 1977. **

[GE77b] *Test Procedure Processor--User Guide*. Schenectady, NY: General Electric Co., 1977. **

[GRC79] *Fortran Automated Verification System (FAVS), Vol. I, User's Manual*. Santa Barbara, CA: General Research Corp., January 1979. **

[Gabo76] Gabow, H.N., S.N. Maheshward, and L.J. Osterweil. "On Two Problems in the Generation of Program Test Data." *IEEE: Transactions on Software Engineering*, 2/3 (Sep 1976):227-231.

[Gaff79] Gaffney, J.E. 1979. "A Comparison of a Complexity-Based and Halstead Program Size Estimates." In *Proceedings ACM Computer Science Conference*, February, Dayton, OH, 35-36.

[Gaff80] Gaffney, J.E., and G.L. Heller. 1980. "Macro Variable Software Models for Application to Improved Software Development Management." In *Proceedings Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society. **

[Gaff81a] Gaffney, J.E. 1981. "Software Metrics: A Key to Improved Software Development Management." In *Conference on Computer Science and Statistics: Proceedings 13th Annual Symposium on the Interface*, March, Carnegie-Mellon, PA, 211-220. Springer-Verlag.

[Gaff81b] Gaffney, J.E. 1981. "Metrics in Software Quality Assurance." In *Proceedings ACM Annual National Computer Conference*, November 9-11, Los Angeles, CA, 126-130. Baltimore, MD: ACM Order Department.

[Gaff88] Gaffney, J.E., and C.F. Davis. March 1988. *An Approach to Estimating Software Errors and Availability*. Software Productivity Consortium. Technical Report SPC-TR-88-007.

[Gali87a] Galiano, E. June 1987. *Vectorization Over Multiple Data Sets*. Georgia Institute of Technology. Design Project Report. **

[Gali87b] Galiano, E. 1987. *Program Execution Over Multiple Data Sets*. Georgia Institute of Technology. **

[Gall81] Gallier, J.H. "Nondeterministic Flowchart Programs with Recursive Procedures: Semantics and Correctness." *Theoretical Computer Science*, 13/3 (Mar 1981):239-270.

[Gann75] Gannon, J.D., and J.J. Horning. "Language Design for Programming Reliability." *IEEE: Transactions on Software Engineering*, 1/2 (1975):179-191.

[Gann76] Gannon, J.D. 1976. "Data Types and Programming Reliability: Some Preliminary Evidence." In *P eedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for ew York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press.

[Gann77] Gannon, J.D. "An Experimental Evaluation of Data Type Conventions." *ACM: Communications of the ACM*, 20/8 (Aug 1977):584-595.

[Gann79] Gannon, C. "Error Detection Using Path Testing and Static Analysis." *IEEE: Computer*, 12/8 (Aug 1979):26-31.

[Gann80] Gannon, J.D., P.R. McMullin, R.G. Hamlet, and M. Ardis. "Testing Traversible Stacks." *ACM: SIGPLAN Notices*, 15/1 (Jan 1980):58-65.

[Gann81] Gannon, J.D., P.R. McMullin, and R.G. Hamlet. "Data Abstraction Implementation, Specification and Testing." *ACM: Transactions on Programming Languages and Systems*, 3/3 (Jul 1981):211-223.

[Gann83] Gannon, J.D., E.E. Katz, and V.R. Basili. August 1983. *Characterizing Ada Programs: Packages*. The Measurement of Computer Software Performance. Los Alamos National Laboratory. **

[Gann85] Gannon, J.D., E.E. Katz, and V.R. Basili. 1985. "Metrics for Characterizing Ada Packages." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England. Washington, DC: IEEE Computer Society Press.

[**Gann86**] Gannon, J.D., E.E. Katz, and V.R. Basili. "Metrics for Ada Packages: An Initial Study." *ACM: Communications of the ACM*, 29/7 (Jul 1986):616-623.

[**Garc83**] Garcia-Molina, H., F. Germano, and W. Kohler. 1983. "Architectural Overview of a Distributed Software Testbed." In *Proceedings IEEE 16th Hawaii International Conference on System Sciences*, January, Honolulu, HA, 310-319. **

[**Garc84**] Garcia-Molina, H., F. Germano, and W.H. Kohler. "Debugging a Distributed Computing System." *IEEE: Transactions on Software Engineering*, 10/2 (Mar 1984):210-219.

[**Gare78**] Garey, M., and D. Johnson. 1978. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman. **

[**Garm81**] Garman, J.R. "The Bug Heard 'Round the World." *ACM: Software Engineering Notes*, 6/5 (Oct 81):3-10.

[**GaudXX**] Gaudel, M.C., and B. Marre. *Algebraic Specifications and Software Testing: Theory and Applications*. Rapport LRI. Report 407. **

[**Geig79**] Geiger, W., L. Gmeiner, H. Trauboth, and U. Voges. "Program Testing for Nuclear Reactor Protection Systems." *IEEE: Computer*, 12/8 (Aug 1979):10-18.

[**Gell78**] Geller, M. "Test Data as an Aid to Proving Program Correctness." *ACM: Communications of the ACM*, 21/5 (May 1978):368-375.

[**Gelp79**] Gelpert, D. "Testing Maintainability." *ACM: SIGSOFT Software Engineering Notes*, 4/2 (Apr 1979):7-12.

[**Gelp88**] Gelperin, G., and B. Hetzel. "The Growth of Software Testing." *ACM: Communications of the ACM*, 31/6 (Jun 1988):687-695.

[**Gerh76a**] Gerhart, S.L., and L. Yelowitz. "Observations of Fallibility in Applications of Modern Programming Methodologies." *IEEE: Transactions on Software Engineering*, 2/3 (Sep 1976):195-207.

[**Gerh76b**] Gerhart, S.L., and L. Yelowitz. "Control Structure Abstractions of the Backtracking Technique." *IEEE: Transactions on Software Engineering*, 2/4 (Dec 1976).

[**Gerh78**] Gerhart, S.L. August 1978. *Program Verification in the 1980's: Problems, Perspectives, and Opportunities*. Marina del Ray, CA: Information Sciences Institute. Report ISI/RR-78-71. **

[**Gerh79**] Gerhart, S.L. 1979. "Program Validation." In *Computing Systems Reliability*, T. Anderson and B. Randers (eds.), 66-108. Cambridge University Press. **

[**Gerh80**] Gerhart, S.L., et al. 1980. *An Overview of AFFIRM: A Specification and Verification System*. University of Southern California. Technical Report PR-79-81. Also published in *Proceedings Information Processing (IFIP) Congress '80*343-347. Tokyo, Japan.

[**Gerh84**] Gerhart, S.L. 1984. "Application of Axiomatic Methods to a Specification Analyzer." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 441-451. Washington, DC: IEEE Computer Society Press.

[**Gerh85**] Gerhart, S. 1985. *A Test Data Generation Method Using Prolog*. Wang Institute of Graduate Studies. Technical Report 85-02. **

[**Gerh88a**] Gerhart, S.L. 1988. "A Broad Spectrum Toolset for Upstream Testing, Verification, and Analysis." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 4-12. Washington, DC: IEEE Computer Society Press.

[**Gerh88b**] Gerhart, S.L. 1988. Position Statements from Panel on: Logic-Based and Constraint-Based Techniques for Software Analysis. In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 137-140. Washington, DC: IEEE Computer Society Press.

[**Germ82a**] German, S.M., D.P. Helmbold, and D.C. Luckham. October 1982. "Monitoring for Deadlocks in Ada Tasking." In *Proceedings AdaTEC Conference on Ada*, October, Arlington, VA, 10-25.

[**Germ82b**] German, S.M. 1982. *A Finely Grained Concurrent Algorithm for Deadlock Detection*. Unpublished manuscript. **

[**Germ84**] German, S.M. "Monitoring for Deadlock and Blocking in Ada Tasking." *IEEE: Transactions on Software Engineering*, 10/6 (Nov 1984):764-777.

[**Gerr85**] Gerrard, C.P., D. Coleman, and R. Gallimore. June 1985. *Formal Specification and Design Time Testing*. Software Sciences Ltd. Internal Report. **

[Gert84] Gerth, R.B., and W.P. de Roever. 1984. "A Proof System for Concurrent Ada Programs." In *Science of Computer Programming 4*, 159-204. Elsevier North-Holland. **

[Getz83] Getz, S.L., G. Kalligiannis, and S.R. Schach. "A Very High-Level Interactive Graphical Trace for the Pascal Heap." *IEEE: Transactions on Software Engineering*, 9/2 (1983):179-185.

[Giam86] Giammo, T. 1986. "Relaxation of the Common Failure Rate Assumption in Modeling Software Reliability." In *Reliability: State of the Art*, A. Bendell and P. Mellor (eds.), 31-44. Oxford: Pergamon Infotech. **

[Gibs89] Gibson, V.R., and J.A. Senn "System Structure and Software Maintenance Performance." *ACM: Communications of the ACM*, 32/3 (Mar 1989):347-358.

[Gide74] Gideadi, A.N., and H. Ledgard. "On a Proposed Measure of Program Structure." *ACM: SIGPLAN Notices*, 9/5 (May 1974):31-36. **

[Gilb76] Gilb, T. 1976. *Software Metrics*. Winthrop Computer Systems Series. Englewood Cliffs, NY: Winthrop. **

[Gilb79] Gilb, T. "A Comment on The Definition of Maintainability." *ACM: SIGSOFT Software Engineering Notes*, 4/3 (Jul 1979):32-33.

[GilkXX] Gilkey, T.J., J.R. White, and T.L. Booth. *Performance Analysis as a Software Design Tool*. University of Connecticut.

[Gill88] Gilles, J., and R. Ford. May 1988. "A Guided Tour through a Window Oriented Debugging Environment." In *Proceedings Ada Europe Conference*, June, Munich, Germany . New York: Cambridge University Press.

[Ginz65] Ginzberg, M.G. "Notes on Testing Real-Time System Programs." *IBM Systems Journal*, 4/1 (1965):58-73.

[Gira73] Girard, E., and J.-C. Rault. 1973. "A Programming Technique for Software Reliability." In *Conference Record 1973 IEEE Symposium on Computer Software Reliability*, April 30 - May 2, New York, 44-50. **

[Girg85] Girgis, M.R., and M.R. Woodward. 1985. "An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 313-319. Washington, DC: IEEE Computer Society Press.

[Girg86a] Girgis, M.R., and M.R. Woodward. 1986. "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria." In *Proceeding: Workshop on Software Testing*, July 15-17, Banff, Canada, 64-73. Washington, DC: IEEE Computer Society Press.

[Girg86b] Girgis, M.R. March 1986. *Studies of Program Test Coverage and the Development of an Automated Support System*. Ph.D. thesis, Liverpool University. **

[Glas79] Glass, R.L. 1979. *Software Reliability Guidebook*. Englewood Cliffs, NJ: Prentice-Hall.

[Glas80] Glass, R.L. "A Benefit Analysis of Some Software Reliability Methodologies." *ACM: SIGSOFT Software Engineering Notes*, 5/2 (Apr 1980):26-33.

[Glas81] Glass, R.L. "Persistent Software Errors." *IEEE: Transactions on Software Engineering*, 7/2 (Mar 1981):162-168.

[Glig87] Gligor, V.D., C.S. Chandersekaran, W. Jiang, A. Johri, G.L. Luckenbaugh, and L.E. Reich. "A New Security Testing Method and Its Application to the Secure Xenix Kernel." *IEEE: Transactions on Software Engineering*, 13/2 (Feb 1987):169-183.

[Gmei79] Gmeiner, L., and U. Voges. 1979. "Software Diversity in Reactor Protection Systems: An Experiment." In *Proceedings Safety of Computer Control Systems (SAFECOM) '79*, L. Lauber (ed.), , 75-79. New York: Pergamon. **

[Godf87] Godfrey, S., C. Brophy, et al. July 1987. *Assessing the Ada Design Process and Its Implications: A Case Study*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-87-004. **

[Godo77] Godoy and Engels. 1977. "Software Sneak Analysis." In *Proceedings AIAA Conference on Computers in Aerospace: Exploration of the Outer Solar System*, vol. 50, November, Los Angeles, CA. New York: American Institute of Aeronautics and Astronautics. **

[Goel78] Goel, A.L., and K. Okumoto. 1978. "An Analysis of Recurrent Software Errors in a Real-Time Control System." In *Proceedings 31st ACM Annual National Computer Conference*, December 4-6, Washington, DC, 496-501. New York: Association for Computing Machinery. **

[Goel79] Goel, A.L., and K. Okumoto. "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures." *IEEE: Transactions on Reliability*, R-28/3 (1979):206-211.

[Goel80a] Goel, A.L., and K. Okumoto. March 1980. *A Time Dependent Error Detection Rate Model for Software Performance Assessment with Applications*. Syracuse University. Annual report to RADC. **

[Goel80b] Goel, A.L. "Software Error Detection Model with Applications." *Journal of Systems and Software*, 1/3 (1980):243-249. **

[Goel80c] Goel, A.L. "A Summary of the Discussion on "An Analysis of Competing Software Reliability Models." *IEEE: Transactions on Software Engineering*, 6/5 (Sep 1980):501-502.

[Goel81] Goel, A.L., and K. Okumoto. 1981. "When to Stop Testing and Start Using Software?" In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March:131-138.

[Goel82] Goel, A.L. October 1982. *Software Reliability and Estimation Techniques*. Griffis Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-82-263. **

[Goel83] Goel, A. April 1983. *A Guidebook for Software Reliability Assessment*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-83-176.

[Goel85] Goel, A.L. "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1411-1423.

[Goel88] Goel, A.L. October 1988. *An Experimental Investigation into Software Reliability*. Griffis Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-88-213.

[Goel89] Goel, A. 1989. "Real-Time Performance Benchmarks for Ada." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 145-153. Washington, DC: ACM Ada Technical Committee. **

[Gogu78] Goguen, J.A., J.W. Thatcher, and E.G. Wagner. 1978. "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types." In *Current Trends in Programming Methodology*, Vol. 4, R. Yeh (ed.), 80-149. Englewood Cliffs, NJ: Prentice-Hall. **

[Gogu79a] Goguen, J.A., and J.J. Tardo. 1979. "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications." In *Proceedings Conference on Specification of Reliable Software*, Cambridge, MA, 170-189. **

[Gogu79b] Goguen, J.A., J.J. Tardo, N. Williamson, and M. Zamfir. "A Practical Method for Testing Algebraic Specifications." *UCLA Computer Science Department Quarterly*, (1979). **

[Gogu80] Goguen, J.A. "Thoughts on Specification, Design, and Verification." *ACM: Software Engineering Notes*, 5/3 (Jul 1980):29-33.

[Gold80] Goldberg, J. 1980. "SIFT: A Provable Fault-Tolerant Computer for Aircraft Flight Control." In *Proceedings Information Processing (IFIP) Congress '80*, 151-156. Tokyo, Japan. **

[Good70] Good, D.I., and R.L. London. "Computer Interval Arithmetic: Definition and Proof of Correct Implementation." *ACM: Journal of the ACM*, 17/4 (Oct 1970):603-612.

[Good75a] Goodenough, J.B., and S.L. Gerhart. "Toward a Theory of Test Data Selection." *IEEE: Transactions on Software Engineering*, 1/2 (Jun 1975):156-173.

[Good75b] Goodenough, J.B., and S.L. Gerhart. "Correction to 'Toward a Theory of Test Data Selection.' " *IEEE: Transactions on Software Engineering*, 1/4 (Dec 1975):425. **

[Good75c] Good, D.I., R.L. London, and W.W. Bledose. 1975. "An Interactive Program Verification System." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 482-492. IEEE Cat. No. 75CH0940-7CSR.

[Good75d] Goodenough, J.B. "Exception Handling: Issues and a Proposed Notation." *ACM: Communications of the ACM*, 18/12 (Dec 1975):683-696.

[Good75e] Good, D.I. 1975. "Provable Programming." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 411-419. IEEE Cat. No. 75CH0940-7CSR.

[**Good79a**] Goodenough, J.B. 1979. "A Survey of Program Testing Issues." In *Research Directions in Software Technology*, P. Wegner (ed.), 316-340. Cambridge, MA: MIT Press.

[**Good79b**] Good, D.I., R.M. Cohen, and J. Keeton-Williams. 1979. "Principles of Proving Programs Correct." In *Proceedings 6th ACM Annual Symposium on Principles of Programming Languages*, San Antonio, TX, 42-52. **

[**Good82a**] Good, D.I. 1982. *The Proof of a Distributed System in Gypsy*. University of Texas at Austin. Technical Report ICSCA-CMP-30. **

[**Good82b**] Good, D.I., A.E. Siebert, and L.M. Smith. December 1982. *Message Flow Modulator - Final Report*. University of Texas at Austin. Technical Report ICSCA-CMP-34. **

[**Good84a**] Good, D.I. January 1984. *Structuring a System for A1 Certification*. University of Texas at Austin. Internal Note #145. **

[**Good84b**] Good, D.I., L. DeViot, and M.K. Smith. June 1984. *Using the Gypsy Methodology*. University of Texas at Austin. **

[**Good86a**] Good, D.I. 1986. *Report on Gypsy 2.05 - January 1986*. University of Texas at Austin. **

[**Good86b**] Goodenough, J. 1986. *Ada Programmer Errors*. Unpublished manuscript. **

[**Good88**] Good, D.I. May 1988. *Predicting Computer Behavior*. Computational Logic, Inc. Technical Report CLI-20. **

[**Gord76**] Gordon, R.D., and M.H. Halstead. 1976. "An Experiment Comparing Fortran Programming Times with the Software Physics Hypothesis." In *Proceedings AFIPS National Computer Conference*, vol. 45, June 7-10, New York, NY, 935-937. Montvale, NJ: AFIPS Press. Also published as Purdue University Technical Report CSD-TR-167.

[**Gord77**] Gordon, R.D. 1977. *A Measure of Mental Effort Related to Program Clarity*. Ph.D. diss., Purdue University. **

[**Gord79a**] Gordon, R.D. "Measuring Improvements in Program Clarity." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):79-90.

[**Gord79b**] Gordon, R.D. "A Qualitative Justification for a Measure of Program Clarity." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):121-127.

[**Gord85a**] Gordon, K. October 1985. *Technical Note On Software Metrics*. Mitre Corp.

[**Gord85b**] Gordon, A.J. August 1985. *Ordering Errors in Distributed Programs*. University of Wisconsin. Technical Report 611. **

[**Gord86**] Gordon, A.J., and R.A. Finkel. 1986. "TAP: A Tool to Find Timing Errors in Distributed Programs." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 154-163. Washington, DC: IEEE Computer Society Press.

[**Gord88**] Gordon, A.J., and R.A. Finkel. "Handling Timing Errors in Distributed Programs." *IEEE: Transactions on Software Engineering*, 12/10 (Oct 1988):1525-1535.

[**Gorl87**] Gorlick, M.M., C.F. Kesselman, D.A. Marotta, and D.S Parker. May 1987. *Mockingbird: A Logical Methodology for Testing*. Computer Science Laboratory.

[**Gors80**] Gorsline G.W., and R.G. Fainter. 1980. "Program Complexity Measures." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 161-162.

[**Goul72**] Gould, J.D., and P. Drongowski. 1972. *A Controlled Psychological Study of Program Debugging*. Yorktown Heights, NY: IBM Corp. IBM Report RC4083. **

[**Goul74**] Gould, J.D., and P. Drongowski. "An Exploratory Study of Computer Program Debugging." *Human Factors*, 16/3 (May 1974):258-277.

[**Goul75**] Gould, J.D. "Some Psychological Evidence on How People Debug Computer Programs." *International Journal of Man-Machine Studies*, 7/2 (1975):151-182. **

[**Gour81**] Gourlay, J.S. 1981. *Theory of Testing Computer Programs*. Ph.D. diss., University of Michigan.

[**Gour83**] Gourlay, J.S. "A Mathematical Framework for the Investigation of Testing." *IEEE: Transactions on Software Engineering*, 9/6 (Nov 1983):686-709.

[**Grad87a**] Grady, R.B. "Measuring and Managing Software Maintenance." *IEEE: Software*, 4/5 (Sep 1987):35-45.

[**Grad87b**] Grady, R.B., and D. Caswell. 1987. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ: Prentice-Hall. **

[**Gran72**] Grant, E.L., and R.S. Leavenworth. 1972. *Statistical Quality Control*. McGraw-Hill Kogakusha Ltd. **

[**Gree76**] Green, T.F., N.F. Schneidewind, G.T. Howard, and R.J. Pariseau. 1976. "Program Structures, Complexity and Error Characteristics." In *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York, 139-154. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press. **

[**Gree81**] Green, A.L., W.J. Decker, and F.E. McGarry. September 1981. *Automated Collection of Software Engineering Data in the Software Engineering Laboratory ((SEL)*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-81-014. **

[**Gree87**] Greenlaw, T. June 1987. *Concatenation of Multiple Program Instances by Exploiting Vector Architectures*. Georgia Institute of Technology. Design Project Report. **

[**Grem84**] Gremillion, L.L. "Determinants of Program Repair Maintenance Requirements." *ACM: Communications of the ACM*, 27/8 (Aug 1984):826-832.

[**Grie76**] Gries, D. "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs." *IEEE: Transactions on Software Engineering*, 2/4 (Dec 1976):238-244.

[**Grie77**] Gries, D. "An Exercise in Proving Parallel Programs Correct." *Communications of the ACM*, 20/12 (Dec 1977):921-930.

[**Grie79**] Gries, D. "Is Sometimes Ever Better than Always?" *ACM: Transactions on Programming Languages and Systems*, 1/2 (1979):258-265.

[**Grie81**] Gries, D. 1981. *The Science of Programming*. New York: Springer-Verlag. **

[**Grif72**] Griffith, P.F., and R.M. Henry. "An Investigatory Study into Human Problem Solving Capabilities as They Relate to Programmer Efficiency." *Computer Personnel*, 3/3 (1972):10-15. **

[**Grna80a**] Grnarov, A., J. Arlat, and A. Avizienis. 1980. "On the Performance of Software Fault-Tolerance Strategies." In *Digest Papers, 10th International Conference on Fault-Tolerant Computing*, October 1-3, Kyoto, Japan, 251-255.

[**Grna80b**] Grnarov, A., J. Arlat, and A. Avizienis. 1980. "Modeling of Software Fault-Tolerance Strategies." In *Proceedings 1980 Pittsburgh Modeling and Simulation Conference*, May, Pittsburgh, PA. **

[**Grov80**] Groves, L.J., and W.J. Rogers. "The Design of a Virtual Machine for Ada." *ACM: SIGPLAN Notices*, 15/11 (Nov 1980):223-234.

[**Guin87**] Guindi, D.S. and C.A. Budinger. 1987. *MUM: Mothra's User Manual*. Georgia Institute of Technology. Technical Report GIT-SERC-87/11.

[**Guin89**] Guindi, D.S., W.M. McCracken, and S. Rugaber. 1989. "Reuse and the Software Life Cycle." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 463-467. Washington, DC: ACM Ada Technical Committee. **

[**Gutt75**] Guttag, J.V. 1975. *The Specification and Application to Programming of Abstract Data Types*. University of Toronto. Technical Report CSRG-59. **

[**Gutt77**] Guttag, J.V. "Abstract Data Types and the Development of Data Structures." *ACM: Communications of the ACM*, 6/20 (Jun 1977):396-404.

[**Gutt78a**] Guttag, J.V., E. Horowitz, and D.R. Musser. "Abstract Data Types and Software Validation." *ACM: Communications of the ACM*, 21/12 (Dec 1978):1048-1064.

[**Gutt78b**] Guttag, J.V., and J.J. Horning. "The Algebraic Specification of Abstract Data Types." *Acta Informatica*, 10/1 (1978):27-52.

[**Gutt80**] Guttag, J.V., and J.J. Horning. 1980. "Formal Specification as Design Tool." In *Proceedings 7th ACM Annual Symposium on Principles of Programming Languages*, January 28-30, Las Vegas, NV, 251-261. Baltimore, MD: ACM Order Department.

[**Gutt85**] Guttag, J.V., J.J. Horning, and J.M. Wing. "The Larch Family of Specification Languages." *IEEE: Software*, (Sep 1985). **

[**HCP82**] *Symbolic Debug/1000 User's Manual*. Cupertino, CA: Hewlett-Packard Co. **

[HONE80] *Formal Definition of the Ada Programming Language (preliminary version).* Honeywell Inc., Cii Honeywell Bull and INRIA, November 1980. **

[Habe75] Habermann, A.N. 1975. *Path Expressions*. Carnegie-Mellon University. **

[Hale82] Haley, A., S.H. Zweben. 1982. "Development and Application of a White Box Approach to Integration Testing." In *Proceedings Workshop on Effectiveness of Testing and Proving Methods*, May, Avalon, CA. **

[Hall73] Haller, A.P., G. Lasseter, R.E. Meeker, and J. Turner. 1973. *Final Report -- FORTRAN Automatic Code Evaluation System*. Austin, TX: Information Research Associates. **

[Hall74] Haller, A.P., 1974. *Automatic Program Analysis*. University of Texas. Technical Report 38. **

[Hall80] Hall, M.L. 1980. "Data Processing Security: Core Concepts." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 51-54.

[Hall86] Hall, W.E., and S.H. Zweben. "The Cloze Procedure and Software Comprehensibility Measurement." *IEEE: Transactions on Software Engineering*, 12/5 (May 1986):608-623.

[Halp87] Halpern, J.D., S. Owre, N. Proctor, and W.F. Wilson. "Muse - A Computer Assisted Verification Systems." *IEEE: Transactions on Software Engineering*, 13/2 (Feb 1987):151-156.

[Hals72a] Halstead, M.H. "Natural Laws Controlling Algorithm Structure?" *ACM: SIGPLAN Notices*, 7/2 (Feb 1972):19-26. **

[Hals73a] Halstead, M.H. "Language Level: A Missing Concept in Information Theory." *ACM: SIGME: Performance Evaluation Review*, 2/3 (Mar 1973):7-9. **

[Hals73b] Halstead, M.H., and R. Bayer. 1973. "Algorithm Dynamics." In *Proceedings 28th ACM Annual National Computer Conference*, August 27-29, Atlanta, GA, 126-135. New York: Association for Computing Machinery.

[Hals75a] Halstead, M.H. May 1975. *Software Physics: Basic Principles*. San Jose, CA: IBM Research Laboratories. Report RJ 1582. **

[Hals75b] Halstead, M.H. 1975. "Toward a Theoretical Basis for Estimating Programming Effort." In *Proceedings ACM Annual National Computer Conference*, October 20-22, Minneapolis, MN, 222-224. **

[Hals76] Halstead, M.H. 1976. *Using the Methodology of Natural Science to Understand Software*. Purdue University. Technical Report CSD-TR-67. **

[Hals77a] Halstead, M.H. 1977. *Elements of Software Science*. New York: Elsevier North-Holland Publishing.

[Hals77b] Halstead, M.H. August 1975. "Potential Impacts of Software Science on Life Cycle Management." In *Software Phenomenology*, 385-400. Washington, DC: U.S. Army Institute for Research in Management Information and Computer Science. **

[Hals77c] Halstead, M.H. August 1977. *A Software Science Analysis of the Writing of a Technical Paper*. Purdue University. Technical Report 242. **

[Hals77d] Halstead, M.H. "On Lines of Code and Programmer Productivity." Letter in *IBM Systems Journal*, 4 (1977). **

[Hals78] Halstead, M.H. 1978. "Software Science - A Progress Report." In *Proceedings U.S. Army Computer Systems Command Software Life Cycle Management Workshop*, August 21-22. **

[Hame82] Hamer, P.G., and G.D. Frewin. 1982. "M.H. Halstead's Software Science - A Critical Examination." In *Proceedings 6th International Conference on Software Engineering*, September 13-16, Tokyo, Japan, 197-206. Washington, DC: IEEE Computer Society Press.

[Haml77a] Hamlet, R.G. "Testing Programs with the Aid of a Compiler." *IEEE: Transactions on Software Engineering*, 3/4 (Jul 1977):279-290.

[Haml77b] Hamlet, R.G. "Testing Programs with Finite Sets of Data." *Computer Journal*, 20/3 (Aug 1977):232-237.

[Haml78a] Hamlet, R.G. 1978. "Test Reliability and Software Maintenance." In *Proceedings 2nd International Computer Software and Applications Conference*, November 13-16, Chicago, IL, 315-320. Long Beach, CA: IEEE Computer Society Press.

[Haml78b] Hamlet, R.G. "Critique of Reliability Theory." In *Digest IEEE Workshop on Software Testing and Test Documentation*, December 18-20, Ft. Lauderdale, FL, 57-69. IEEE Computer Society Technical Committee on Software Engineering. **

[Haml78c] Hamlet, P.A., and J.D. Musa. 1978. "Measuring Reliability of Computation Center Software." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 28-36. Washington, DC: IEEE Computer Society Press.

[Haml79] Hamlet, R.G., M. Ardis, J.D. Gannon, and P.R. McMullin. May 1979. *Testing Data Abstractions through their Implementations*. University of Maryland. Technical Report TR-761. **

[Haml86] Hamlet, D. 1986. "Testing For Probable Correctness." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 92-97. Washington, DC: IEEE Computer Society Press.

[Haml87] Hamlet, D. "Probable Correctness Theory." *Information Processing Letters*, 25/1 (Apr 1987):17-25.

[Haml88] Hamlet, D., and R. Taylor. 1988. "Partition Testing Does Not Inspire Confidence." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 206-216. Washington, DC: IEEE Computer Society Press.

[Han76] Han, Y.W. February 1976. *A Systematic Study of Computer System Reliability*. Ph.D. diss., University of California at Berkeley. **

[Hane72] Haney, H.M. 1972. "Module Connection Analysis--A Tool for Scheduling Software Debugging." In *Proceedings AFIPS Fall Joint Computer Conference*, vol. 40, May 16-18, Atlantic City, NJ. Montvale, NJ: AFIPS Press.

[Hanf70] Hanford, K.V. "Automatic Generation of Test Cases." *IBM Systems Journal*, 9/4 (Dec 1970):242-257.

[Hank80] Hanks, J.M. 1980. *Testing Cobol Programs by Mutation: Volume I - Introduction to the CMS.1 System, Volume II - CMS.1 System Documentation*. Georgia Institute of Technology. Technical Report GIT-ICS-80/04. **

[Hans73] Hansen, P.B. "Testing a Multiprogramming System, " *Software, Practice and Experience*, (Apr-Jun 1973):272-279.

[Hans78] Hansen, W.J. "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)." *ACM: SIGPLAN Notices*, 13/3 (1978):29-33.

[Hans84] Hansen, H.L. (ed.). 1984. *Software Validation*. New York: Elsevier Science Publishers.

[Hant76] Hantler, S., and J.C. King. "An Introduction to Proving the Correctness of Programs." *ACM: Computing Surveys*, 8/3 (Sep 1976):331-353.

[Hara83] Harandi, M.T. 1983. "Knowledge-Based Program Debugging." In *Proceedings SoftFair I: 1st Conference on Software Development Tools, Techniques, and Alternatives*, July 25-28, Arlington, VA, 282-288. Los Angeles: IEEE Computer Society. **

[Hare82] Harel, E., and E.R. McLean. November 1982. *The Effects of Using a Nonprocedural Computer Language on Programmer Productivity*. University of California at Los Angeles. Working Paper 3-83. **

[Harr81a] Harrison, W., and K. Magel. "A Complexity Measure Based on Nesting Level." *ACM: SIGPLAN Notices*, 16/3 (Mar 1981):63-74.

[Harr81b] Harrison, W., and K. Magel. "A Topological Analysis of Computer Programs with Less Than Three Binary Branches." *ACM: SIGPLAN Notices*, 16/4 (Apr 1981):51-63.

[Harr82] Harrison, W., K. Magel, R. Kluczny, and A. DeKock. "Applying Software Complexity Metrics to Program Maintenance." *IEEE: Computer*, 15/9 (Sep 1982):65-79.

[Harr85] Harrison, W., and C.R. Cook. "A Method of Sharing Software Complexity Data." *ACM: SIGPLAN Notices*, 20/2 (Feb 1985):42-51.

[Harr88a] Harrison, L.J. June 1988. *CASEX: A Concurrent Ada Symbolic Executor*. M.S. thesis, University of California at Santa Barbara. **

[Harr88b] Harrison, W. "How Complex is Your Software?" *Computer Language*, (Jan 1988):73-75.

[Harr88c] Harrison, L.J., and R.A. Kemmerer. 1988. "An Interleaving Symbolic Execution Approach for the Formal Verification of Ada Programs with Tasking." In *Proceedings Ada Europe Conference*, June, Munich, Germany , 15-27. New York: Cambridge University Press.

[Hart71] Hartmanis, J., and J.E. Hopcroft. "An Overview of the Theory of Computational Complexity." *ACM: Journal of the ACM*, 18/3 (Jul 1971):444-475.

[Hart79] Hart, J.J. "The Advanced Interactive Debugging System (AIDS)." *ACM: SIGPLAN Notices*, 14/12 (Dec 1979):110-121.

[Hart84] Harter, P.K. July 1984. *Response Times in Level Structured Systems*. University of Colorado. **

[Harv82] Harvey, P. 1982. *Fault-Tree Analysis of Software*. M.S. thesis, University of California at Irvine. **

[Hass80] Hassell, J., L.A. Clarke, and D.J. Richardson. 1980. *A Close Look at Domain Testing*. University of Massachusetts. COINS Technical Report 80-16. Also published in *IEEE: Transactions on Software Engineering*, 8/4 (Jul 1982):380-390.

[Hech72] Hecht, M., and J. Ullman. "Flow Graph Reducibility." *SIAM: Journal of Applied Mathematics*, 1 (1972):188-202. **

[Hech75] Hecht, M.S., and J.D. Ullman. "A Simple Algorithm for Global Data Flow Analysis Problems." *SIAM: Journal of Computing*, 4 (Dec 1975):519-532.

[Hech76] Hecht, H. 1976. "Fault-Tolerant Software: Motivation and Capabilities." *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press.

[Hech77a] Hecht, M.S. 1977. *Flow Analysis of Computer Programs*. Amsterdam: North Holland.

[Hech77b] Hecht, H. January 1977. *Measurement, Estimation and Prediction of Software Reliability*. NASA Langley Research Center. Report NASA-CR-145205. **

[Hech77c] Hecht, H., W.A. Sturm, and S. Trattner. 1977. "Reliability Measurement During Software Development." In *Proceedings AIAA Conference on Computers in Aerospace: Exploration of the Outer Solar System*, vol. 50, November, Los Angeles, CA. New York: American Institute of Aeronautics and Astronautics. Also published as Aerospace Report N78-10487/4, September 1977. **

[Hech79] Hecht, H. "Fault-Tolerant Software." *IEEE: Transactions on Reliability*, R-28/3 (Aug 1979):227-232.

[Hech80] Hecht, H. 1980. "Mini-Tutorial on Software Reliability." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 383-385. Los Alamitos, CA: IEEE Computer Society Press.

[Heid82] Heidler, W., et al. May 1982. *Software Testing Measures*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-82-135. **

[Hell72] Hellerman, L. "A Measure of Computational Work." *IEEE: Transactions on Computers*, C-21/5 (May 1972):439-446.

[Hell87] Heller, G.L. October 1987. *Data Collection Procedures for the Rehosted SEL Database*. NASA Software Engineering Laboratory Series. Technical Report SEL-87-008.

[Helm83] Helmbold, D.P. and D.C. Luckham. November 1983. *Runtime Detection and Description of Deadness Errors in Ada Tasking*. Stanford University. Program Analysis and Verification Group Report no. 22. Technical Report CSL-TR-83-249.

[Helm84a] Helmbold, D.P., and D.C. Luckham. 1984. "Debugging Ada Tasking Programs." Stanford University Technical Report CSL-TR-84-262. Published in *Proceedings IEEE Computer Society Conference on Ada Applications and Environments*, October 15-18, St. Paul, MN, 96-110. Also published in *IEEE: Software*, 2/2 (Mar 1985):255-274.

[Helm84b] Helmbold, D.P. 1984. *Distributed Deadness Monitoring in Ada*. Stanford University. **

[Helm85] Helmbold, D.P., and D.C. Luckham. 1985. *TSL: Task Sequencing Language*. Stanford University Technical Report. Also in *Proceedings SIGAda International Conference*, May, Paris, France. Published in *ACM: Ada Letters*, V/2 (Sep-Oct 1985):255-274.

[Hend75] Henderson, P. 1975. "Finite State Modeling in Program Development." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 221-227. IEEE Cat. No. 75CH0940-7CSR.

[Heng87] Hengeveld, W., and J. Kroon. 1987. "Using Checking Sequences for OSI Session Layer Conformance Testing." In *Proceedings 7th IFIP Protocol Symposium*, May, Zurich, Switzerland. **

[Henn76a] Hennell, M.A., D. Hedley, and M.R. Woodward. "Experience with an Algol68 Numerical Algorithms Testbed." In *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York, 171-179. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press. **

[Henn76b] Hennell, M.A., M.R. Woodward, and D. Hedley. "On Program Analysis." *Information Processing Letters*, 5/5 (Nov 1976):136-140.

[Henn78] Hennell, M.A. "An Experimental Testbed for Numerical Software." *Computer Journal*, 21/4 (Nov 1978):333-336. Also published in *Information Processing Letters*, 22/2 (Feb 1979):53-56.

[Henn79] Hennell, M.A., M.R. Woodward, and D. Hedley. 1979. "The Testing of a Software Tool." In *Proceedings International Symposium on Applications and Software Engineering*, September, Montreal, Canada. Acta Press. **

[Henn81] Hennell, M.A., I.J. Riddell, and M.R. Woodward. "A Mutation Analysis of Numerical Software." *ACM: SIGNUM Newsletter*, (Jul 1981). **

[Henn84] Hennell, M.A., D. Hedley, and I.J. Riddell. 1984. "Assessing a Class of Software Tools." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 266-277. Washington, DC: IEEE Computer Society Press.

[HennXX] Hennell, M.A., P. Fairfield, and M.U. Shaikh. *Functional Testing*. University of Liverpool, Statistical and Computational Mathematics Dept.

[Henr79] Henry, S.M. 1979. *Information Flow Metrics for the Evaluation of Operating Systems' Structure*. Ph.D. diss., Iowa State University. **

[Henr81a] Henry, S.M., D.G. Kafura, and K. Harris. 1981. "On the Relationships Among Three Software Metrics." In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March:81-88.

[Henr81b] Henry, S.M., and D.G. Kafura. "Software Structure Metrics Based on Information Flow." *IEEE: Transactions on Software Engineering*, 7/5 (Sep 1981):510-518.

[Henr84] Henry, S.M., and D.G. Kafura. "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics." *Software Practice and Experience*, 14/6 (Jun 1984):561-573. **

[Henr85] Henry, S.M., J.D. Arthur, and R.E. Nance. March 1985. *A Procedural Approach to Evaluating Software Development Methodologies*. Virginia Polytechnic Institute. TR-85-20.

[Henr88a] Henry, S.M., and S. Wake. 1988. *Predicting Maintainability with Software Quality Metrics*. Virginia Polytechnic Institute. TR-88-46.

[Henr88b] Henry, S.M., D. Kafura, K. Mayo, A. Yerneni, and S. Wake. 1988. *A Reliability Model Incorporating Software Quality Factors*. Virginia Polytechnic Institute. TR-88-45.

[HenrXX] Henry, S.M., and R. Goff. *Comparison of a Graphical and a Textual Design Language Using Software Quality Metrics*. Virginia Polytechnic Institute.

[Herd79] Herd, J.R., J.N. Postak, W.E. Russell, and K.R. Stewart. June 1977. *Software Cost Estimation Study--Study Results*. Rockville, MD: Doty Associates. Final Technical Report RADC-TR-77-220. **

[Herm76] Herman, P.M. *The Australian Computer Journal*, 8/3 (Nov 1976):347-354. **

[Hess88] Hess, J.A. 1988. "Measuring Software for Its Reuse Potential." In *Proceedings Annual Reliability and Maintainability Symposium*, January, 202-206. **

[Hetz73] Hetzel, W.C. 1973. *Program Test Methods*. A collection of papers based on *Proceedings Computer Program Test Methods Symposium*, W.C. Hetzel (ed.), University of North Carolina, Chapel Hill. Englewood Cliffs, NJ: Prentice-Hall. **

[Hetz76] Hetzel, W.C. 1976. *An Experimental Analysis of Program Verification Methods*. Ph.D. thesis, University of North Carolina. **

[Hetz84] Hetzel, B. (ed.). 1984. *A Complete Guide to Software Testing*, 2nd edition. Wellesley, MA: QED Information Sciences.

[Hewi76] Hewitt, C., and A. Yonezawa. December 1976. *Symbolic Evaluation Using Conceptual Representation for Programs with Side-Effects*. MIT Artificial Intelligence Laboratory. Memo 399. **

[Hibb82] Hibbard, P.G., and T.L. Rodeheffer. 1982. "Optimizing for a Multiprocessor: Balancing Synchronization Cost Against Parallelism." In *Proceedings International Symposium on Programming, 5th Colloqium*, April, Turin, Italy, 194-211. New York: Springer-Verlag.

[Hill83] Hill, C.R. 1983. *A Real-Time Microprocessor Debugging Technique*. Briarcliffe Manor, NY: Computer Systems Research Philips Labs.

[Hite88] Hite, L.A and D.P. Miller. 1988. *Designing a Testing Strategy for Expert Systems*. Virginia Polytechnic Institute. TR-88-41.

[Ho78] Ho, S.-B.F. November 1978. *A Systematic Approach to the Development and Validation of Software for Critical Applications*. Ph.D. diss., University of California at Berkeley. In *Proceedings 4th*

International Conference on Software Engineering, September 27-29, Munich, Germany, 231-240. Washington, DC: IEEE Computer Society Press. **

[Ho79] Ho, P. November 1979. *A DC DYMOL to DC Constrained Expressions Translator*. M.S. thesis, University of Massachusetts. **

[Hoar69] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming." *ACM: Communications of the ACM*, 12/10 (1969):576-583.

[Hoar71a] Hoare, C.A.R. 1971. "Procedures and Parameters: An Axiomatic Approach." In *Lecture Notes in Mathematics*, Vol. 188. New York: Springer-Verlag. **

[Hoar71b] Hoare, C.A.R. "Proof of a Program: FIND." *ACM: Communications of the ACM*, 14/1 (Jan 1971):39-45.

[Hoar72] Hoare, C.A.R. "Proof of Correctness of Data Representations." *Acta Informatica*, 1/4 (1972):271-281.

[Hoar74] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept." *ACM: Communications of the ACM*, 17/10 (Oct 1974):549-557.

[Hoar75] Hoare, C.A.R. "Parallel Programming: An Axiomatic Approach." *Computing Languages*, 1/2 (Jun 1975):151-160.

[Hoar78] Hoare, C.A.R. "Communicating Sequential Processes." *ACM: Communications of the ACM*, 21/8 (Aug 1978):666-677. Also published by Prentice-Hall International, 1985.

[Hoar81] Hoare, C.A.R. *A Calculus of Total Correctness for Communicating Processes*. Science of Computer Programming-1, 1/1-2 (Oct 1981):49-72.

[Hoar85] Hoare, C.A.R. "Communicating Sequential Processes," Prentice-Hall International, 1985. **

[Hoar87] Hoare, C.A.R. "An Overview of Some Formal Methods for Program Design." *IEEE: Computer*, 20/9 (Sep 1987):85-91.

[Hodg76] Hodges, B.C., and J.P. Ryan. 1976. "A System for Automatic Software Evaluation." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA, 617-623. Washington, DC: IEEE Computer Society Press.

[Hoer74] Hoermann, H.A. 1974. "Principles of Reliability Assessment." In *Proceedings CSNI Specialist Meeting on the Development and Application of Reliability Techniques to Nuclear Plants*, April. **

[Hoff73] Hoffman, R.H. August 1973. *Product Assurance Confidence Evaluator (PACE) Programmer's Guide*. TRW Defense and Space Systems Group. **

[Hoff75] Hoffman, R.H. 1975. "NASA/Johnson Space Center Approach to Automated Test Data Generation." In *4th Conference on Computer Science and Statistics: Proceeedings 8th Symposium on the Interface*, February, Los Angeles, CA. Springer-Verlag. **

[Hoff76] Hoffman, R.H. January 1976. *User Information for the Interactive Automated Test Data Generator (ATDG) System*. Houston, TX: NASA. Report JSC-10832. **

[Hoff77] Hoffman, H.-M. June 1977. *An Experiment in Software Error Occurrence and Detection*. M.S. thesis, Naval Postgraduate School. **

[Holt76] Holthouse, M.A., and E. Cosloy. 1976. "A Practical System for Automatic Testcase Generation." In *Proceedings AFIPS National Computer Conference*, vol. 45, June 7-10, New York, NY. Montvale, NJ: AFIPS Press. **

[Holt78] Holthouse, M.A., and M.J. Hatch. 1978. *Experience with Automated Testing of Elementary Program Functions*. University of Victoria. Technical Report DM-212-IR. **

[Holz82] Holzmann, G. "A Theory for Protocol Validation." *IEEE: Transactions on Computers*, C-31 (Aug 1982):730-738.

[Horn74] Horning, J.J., et al. 1974. "A Program Structure for Error Detection and Recovery." In *Lecture Notes in Computer Science*. Operating Systems, 16. G.Goos and J Hartmanis (eds.), 171-187. New York: Springer Verlag. **

[Hous77] Houssais, B. 1977. "Verification of an Algol 68 Implementation." In *Proceedings Strathclyde Algol 68 Conference*, March, Glasgow, Scotland. Also published in *ACM: SIGPLAN Notices* 12/6 (Jun 1977):117-128.

[Hout81] Houtz, C., and T. Buschbach. March 1981. *Review and Analysis of Conversion Cost-Estimating Techniques*. Falls Church, VA: GSA Federal Conversion Support Center. Technical Report GSA/FCSC-81/001. **

[Howa73] Howard, J.H., and W.P. Alexander. 1973. "Analyzing Sequences of Operations Performed by Programs." In *Program Test Methods*, W.C. Hetzel (ed.). Englewood Cliffs, NJ: Prentice-Hall. **

[Howa85] Howatt, J.W. 1985. *A Quantitative Characterization of Control Flow Context: Software Measures for Programming Environments*.
Ph.D. diss., Iowa State University. **

[Howd74a] Howden, W.E., and L.G. Stucki. January 1974. *A Methodology for Effective Test Case Selection -- Phase I*. Huntington Beach, CA: McDonnell Douglas Astronautics. Technical Report MDC G5301. **

[Howd74b] Howden, W.E. November 1974. *Models of Correct Programs and Program Testing*. University of California at San Diego. Technical Report 10. **

[Howd74c] Howden, W.E. 1974. "Automatic Generation of Program Test Data and Proofs of Program Correctness." In *Workshop on the Attainment of Reliable Software*, April, University of Toronto. **

[Howd75a] Howden, W.E. "Methodology for the Generation of Program Test Data." *IEEE: Transactions on Computers*, C-24/5 (May 1975):554-559.

[Howd75b] Howden, W.E., and J. Laub. 1975. "Automatic Case Analysis of Programs." In *4th Conference on Computer Science and Statistics: Proceeedings 8th Symposium on the Interface*, February, Los Angeles, CA, 347-352. Springer-Verlag. **

[Howd76a] Howden, W.E. 1976. *Elementary Algebraic Program Testing Techniques*. UCSD Computer Science Technical Report No. 12. **

[Howd76b] Howden, W.E., and L.G. Stucki. 1976. *A Methodology for Effective Test Case Selection -- Phase III*. Huntington Beach, CA: McDonnell Douglas Astronautics. Technical Report MDC-G6211. **

[Howd76c] Howden, W.E. "Reliability of the Path Analysis Testing Strategy." *IEEE: Transactions on Software Engineering*, 2/3 (Sep 1976):208-215.

[Howd76d] Howden, W.E. 1976. *Algebraic Equivalence of Elementary Computational Structures*. University of California at San Diego. (Revised 1980). **

[Howd76e] Howden, W.E. 1976. "Experiments with a Symbolic Evaluation System." In *Proceedings AFIPS National Computer Conference*, vol. 45, June 7-10, New York, NY, 899-908. Montvale, NJ: AFIPS Press.

[Howd77a] Howden, W.E. May 1977. *Symbolic Testing - Design Techniques, Costs, and Effectiveness*. Gaithersburg, MD: National Bureau of Standards. Technical Report NBS-GCR-77-89.

[Howd77b] Howden, W.E. "Symbolic Testing and the DISSECT Symbolic Evaluation System." *IEEE: Transactions on Software Engineering*, 3/4 (Jul 1977):266-278.

[Howd77c] Howden, W.E. 1977. *An Evaluation of the Effectiveness of Symbolic Testing*. University of California at San Diego. Technical Report 16. Also published in *Software Practice and Experience*, 8/4 (Jul-Aug 1978):381-397.

[Howd78a] Howden, W.E. "Theoretical and Empirical Studies of Program Testing." *IEEE: Transactions on Software Engineering*, 4/4 (Jul 1978):293-298.

[Howd78b] Howden, W.E. "Algebraic Program Testing." *ACTA Informatica*, no. 10 (1978):53-66.

[Howd78c] Howden, W.E., and H.P. Eichhorst. 1978. "Proving Properties of Programs from Program Traces." In *Tutorial: Software Testing and Validation Techniques*, E. Miller and W.E. Howden (eds.), 46-56. New York: IEEE.

[Howd78d] Howden, W.E. 1978. "Introduction to the Theory of Testing." In *Tutorial: Software Testing and Validation Techniques*, E. Miller and W.E. Howden (eds.), 16-19. New York: IEEE.

[Howd78e] Howden, W.E. "Lindenmayer Grammars and Symbolic Testing." *Information Processing Letters* 7/1 (1978):36-39. **

[Howd78f] Howden, W.E. 1978. "Empirical Studies of Software Validation." In *Tutorial: Software Testing and Validation Techniques*, E. Miller and W.E. Howden (eds.), 280-285. New York: IEEE.

[Howd79] Howden, W.E. 1979. *An Analysis of Software Validation Techniques for Scientific Programs*. University of Victoria. Report DM-171-IR. **

[Howd80a] Howden, W.E. "Functional Testing and Design Abstractions." *Journal of Systems and Software*, no. 1 (1980):307-313.

[Howd80b] Howden, W.E. "Applicability of Software Validation Techniques to Scientific Programs." *ACM: Transactions on Programming Languages and Systems*, 2/3 (Jul 1980):307-320. Previously published as University of Victoria Report DM-171-IR.

[Howd80c] Howden, W.E. "Functional Program Testing." *IEEE: Transactions on Software Engineering*, 6/2 (Mar 1980):162-169.

[Howd80d] Howden, W.E. May 1980. *Completeness Criteria for Testing Elementary Program Functions*. University of Victoria. Technical Report DM-212-IR. Also published in *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 235-243. Washington, DC: IEEE Computer Society Press.

[Howd81a] Howden, W.E. 1981. "Errors, Design Properties, and Functional Program Tests." In *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 115-127. North-Holland. **

[Howd81b] Howden, W.E. 1981. "A Survey of Static Analysis Methods." In *Tutorial: Software Testing and Validation Techniques*, 2nd Edition, E. Miller and W.E. Howden (eds.), 101-115. Los Alamitos, CA: IEEE Computer Society Press.

[Howd81c] Howden, W.E. 1981. "A Survey of Dynamic Analysis Methods." In *Tutorial: Software Testing and Validation Techniques*, 2nd Edition, E. Miller and W.E. Howden (eds.), 209-231. Los Alamitos, CA: IEEE Computer Society Press.

[Howd81d] Howden, W.E. July 1981. *Errors in Data Processing Programs and the Refinement of Current Program Test Methodologies*. Gaithersburg, MD: National Bureau of Standards. Final Report NBS Contract NB79BCA0069. **

[Howd82a] Howden, W.E. "Weak Mutation Testing and Completeness of Test Sets." *IEEE: Transactions on Software Engineering*, 8/4 (Jul 1982):371-379.

[Howd82b] Howden, W.E. "Life-Cycle Software Validation." *IEEE: Computer*, (Feb 1982):71-78.

[Howd83] Howden W.E. 1983. "A General Model for Static Analysis." In *Proceedings IEEE 16th Hawaii International Conference on System Sciences*, January, Honolulu, HA, 163-169. **

[Howd85] Howden, W.E. "The Theory and Practice of Functional Testing." *IEEE: Software*, 2/5 (Sep 1985):6-17.

[Howd86] Howden, W.E. "A Functional Approach to Program Testing and Analysis." *IEEE: Transactions on Software Engineering*, 12/10 (Oct 1986):997-1005.

[Howd87] Howden, W.E. 1987. *Functional Program Testing and Analysis*. New York: McGraw-Hill.

[Howd88] Howden, W.E. 1988. *Comments Analysis and Programming Errors*. University of California at San Diego. Technical Report 88-142. **

[Howd89a] Howden, W.E. 1989. "Current Validation Research and Development Activities." In *Towards SDS Testing and Evaluation: A Collection of Relevant Topics*. IDA Memorandum Report M-513. Alexandria, VA: Institute for Defense Analyses. Draft.

[Howd89b] Howden, W.E. 1989. *Verifying Programs without Specifications*. University of California at San Diego. **

[Howe84] Howes, N.R. "Managing Software Development Projects for Maximum Productivity." *IEEE: Transactions on Software Engineering*, 10/1 (Jan 1984):27-35.

[Hsie82] Hsieh, C-C. 1982. *An Approach to Logical Ripple Effect Analysis for Software Maintenance*. Ph.D. diss., Northwestern University.

[Hsie89] Hsieh, C.S. 1989. "Timing Analysis of Cyclic Concurrent Programs." In *Proceedings 11th International Conference on Software Engineering*, May 15-18, Pitsburgh, PA, 312-318. Washington, DC: IEEE Computer Society Press.

[Huan75] Huang, J.C. "An Approach to Program Testing." *ACM: Computing Surveys*, 7/3 (Sep 1975):113-128.

[Huan78] Huang, J.C. "Program Instrumentation and Software Testing." *IEEE: Computer*, 11/4 (Apr 1978):25-32.

[Huan79] Huang, J.C. "Detection of Data Flow Anomalies Through Program Instrumentation." *IEEE: Transactions on Software Engineering*, SE-5 (1979):226-236.

[Huet80] Huet, G., and J.M. Hullot. 1980. "Proofs by Induction of Equational Theories with Constructors." In *Proceedings 21st FOCS*, 96-107. **

[Hump88] Humphrey, W.S. "Characterizing the Software Process." *IEEE: Software*, 5/2 (Mar 1988):73-79.

[Hunt85] Hunt, W.A. Jr. 1985. *FM8501: A Verified Microprocessor*. University of Texas. Technical Report ICSCA-CMP-47. **

[Hunt87] Hunt, W.A. Jr. 1987. *The Mechanical Verification of a Microprocessor Design*. Computational Logic Inc. Technical Report CLI-6. **

[Hutc83] Hutchens, D.H., and V.R. Basili. August 1983. *System Structure Analysis: Clusterings With Data Bindings*. University of Maryland. Technical Report TR-1310. Also published in *IEEE: Transactions on Software Engineering*, 11/8 (Aug 1985):749-757.

[Hwan81] Hwang, S.-S.V. December 1981. *An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspection*. University of Maryland. Scholarly Paper 362. **

[IEEE83a] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standard 729. February 18, 1983. New York: The Institute of Electrical and Electronics Engineers.

[IEEE83b] *IEEE Standard for Software Test Documentation*. IEEE Standard 829-1983. February 18, 1983. New York: The Institute of Electrical and Electronics Engineers.

[IEEE83c] *IEEE Standard for Software Configuration Management Plans*. IEEE Standard 828-1983. 1983. New York: The Institute of Electrical and Electronics Engineers.

[IEEE84] *IEEE Standard for Software Quality Assurance Plans*. IEEE Standard 730-1984. 1983. New York: The Institute of Electrical and Electronics Engineers.

[IEEE87] *IEEE Standard for Measures to Produce Reliable Software*. IEEE Draft P982.1, June 1987. New York: The Institute of Electrical and Electronics Engineers. **

[IEEE88] *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Draft Standard 1061/D15. New York: The Institute of Electrical and Electronics Engineers.

[INFO76] Infotech (eds.) 1976. "Complexity in Programming." In *Structured Programming*, 25-28 and 65-96. Berkshire, England: Infotech International Ltd. **

[INFO79] *Software Testing, INFOTECH State of the Art Report*. London, England: Infotech, 1979. **

[ISO87a] *OSI Conformance Testing Methodology and Framework Part I: General Concepts*. ISO. DP 9646-1, July 1987. **

[ISO87b] *OSI Conformance Testing Methodology and Framework Part II: Abstract Test Suite Specification*. ISO. DP 9646-2, July 1987. **

[ISO87c] *Lotos - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*. ISO. DIS 8807, August 1987. **

[Iann84] Iannino, A., J.D. Musa, K. Okumoto, and B. Littlewood. "Criteria for Software Reliability Model Comparisons." *IEEE: Transactions on Software Engineering*, 10/6 (Nov 1984):687-691.

[Ibar82] Ibarra, O.H., and B.S. Leininger. "The Complexity of the Equivalence Problem for Simple Loop-Free Programs." *SIAM Journal of Computing*, 11/1 (Feb 1982):15-27.

[Igar73] Igarashi, S., R. London, and D. Luckham. May 1973. *Automatic Verification of Programs I: A Logical Basis and Implementation*. Stanford, CA: Stanford Artificial Intelligence Laboratory. Memo AIM-200. **

[Igna71] Ignalls, D.H. February 1971. *FETE: A FORTRAN Execution Time Estimator*. Stanford University. Technical Report STAN-CS-71-204. **

[Ingl86] Inglis, J. 1986. "Standard Software Quality Metrics." *AT&T Technical Journal*, 65/2 (Mar-Apr 1986):113-118.

[Isod87] Isoda, S., T. Shimomura, and Y. Ono. "VIPS: A Visual Debugger." *IEEE: Software*, 4/3 (May 1987):8-19.

[Itak82] Itakura, M., and A. Takayanagi. 1982. "A Model for Estimating Program Size and Its Evaluation." In *Proceedings 6th International Conference on Software Engineering*, September 13-16, Tokyo, Japan, 104-109. Washington, DC: IEEE Computer Society Press.

[Ives83] Ives, B., M.H. Olson, and J.J. Barondi. "The Measurement of User Information Satisfaction." *ACM: Communications of the ACM*, 26/10 (Oct 1983):785-799.

[JLC84] "Independent Verification and Validation." In *Final Report of the Joint Logistics Commanders' Workshop on Post Deployment Support (PDSS) for Mission-Critical Computer Software, Vol. II - Workshop Proceedings*, Orlando I Software Workshop, June 1984. **

[Jach84] Jachner, J., and V.K. Agarwal. "Data Flow Anomaly Detection." *IEEE: Transactions on Software Engineering*, 10/4 (Jul 1984):432-437.

[Jack71] Jackson, R.S., and S.A. Bravdica. 1971. "Software Validation of the Titan IIIC Digital Flight Control System Using a Hybrid Computer." In *Proceedings AFIPS Fall Joint Computer Conference*, vol. 40, May 16-18, Atlantic City, NJ, 225-232. Montvale, NJ: AFIPS Press.

[Jaha86] Jahanian, F., and A.K. Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE: Transactions on Software Engineering* 12/9 (Sep 1986):890-904.

[Jalo89] Jalote., P. "Testing the Completeness of Specifications." *IEEE: Transactions on Software Engineering*, 15/5 (May 1989):526-531.

[Jame77] James, T. 1977. "Software Cost Estimating Methodology." In *Proceedings National Aerospace Electronics Conference*, 22-28. **

[Jard83] Jard, C., and G.V. Bochman. 1983. "An Approach to Testing Specifications." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):53-59. Baltimore, MD: ACM Order Department.

[Jard87] Jard, C., and O. Drissi. February 1987. *Deriving Trace Checkers for Distributed Systems*. Universite de Rennes. Research Report. **

[Jarr84] Jarratt, R.M.A. 1984. *Software Development Tools for Dataflow Machines*, M.S. thesis, University of Manchester. **

[Jeff85] Jeffery, D.R., and M.J. Lawrence. "Managing Programming Productivity." *Journal of Systems and Software*, 5 (1985):49-58.

[Jeli72] Jelinski, J., and P.B. Moranda. 1972. "Software Reliability Research." In *Statistical Computer Performance Evaluation*, W. Freidberger, ed., 465-484. New York: Academic Press.

[Jeli73] Jelinski, J., and P.B. Moranda. 1973. "Applications of a Probability-Based Model to a Code Reading Experiment." In *Conference Record 1973 IEEE Symposium on Computer Software Reliability*, April 30 - May 2, New York, 78-81. **

[Jenk86] Jenkins, J.R. 1986. *Automated Generation of Input/Output Pairs for the CAIS Validation Test Suite*. M.S. thesis. Arizona State University. **

[Jens83a] Jensen, R.W. 1983. "An Improved Macrolevel Software Development Resource Estimation Model." In *Proceedings 5th ISPA Conference*, April, 88-92. **

[Jens83b] Jensen, R.W., and S. Lucas. 1983. "Sensitivity Analysis of the Jensen Software Model." In *Proceedings 5th ISPA Conference*, April, 384-389. **

[John75] Johnson, J.P. December 1975. *Software Reliability Measurement*. Los Angeles Air Force Station, CA: Space and Missile Systems Organization, Air Force Systems Command. Report SAMSO-TR-75-279.

[John77] Johnson, D.B. August 1977. *Program Analysis with the Aid of a Data Management System*. M.A. thesis, University of Texas. **

[John78] Johnson, M.S. 1978. *The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment*. Ph.D. diss., University of British Columbia. **

[John79] Johnson, M.S. "Translator Design to Support Run-Time Debugging." *Software Practice and Experience*, 9/12 (Dec 1979):1035-1041.

[John81] Johnston, D.E., and A.M. Lister. "A Note on the Software Science Length Equation." *Software Practice and Experience*, 11/8 (Aug 1981). **

[John82a] Johnson, M.S. "Some Requirements for Architectural Support of Software Debugging." In *Proceedings ACM Symposium on Architectural Support for Programming Languages and Operating Systems*. Published in *ACM: SIGPLAN Notices*, 17/4 (Apr 1982). **

[**John82b**] Johnson, M.S. "A Software Debugging Glossary." *ACM: SIGPLAN Notices*, 17/2 (Feb 1982):53-70.

[**John83**] Johnson, J.D., and G.W. Kenney. 1983. "Implementation Issues for a Source Level Symbolic Debugger." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):149-151. Baltimore, MD: ACM Order Department.

[**John84**] Johnson, W.L., and E. Soloway. 1984. "PROUST: Knowledge-Based Program Understanding." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 369-380. Washington, DC: IEEE Computer Society Press.

[**JohnXX**] Johnson, W.L., S. Draper, and E. Soloway. "An Effective Bug Classification Scheme Must Take the Programmer into Account." **

[**Jone76**] Jones, C. 1976. *Program Quality and Programmer Productivity*. San Jose, CA: IBM Corp. **

[**Jone78**] Jones, T.C. "Measuring Programming Quality and Productivity." *IBM Systems Journal*, 17/1 (1978):39-63.

[**Jone79**] Jones, T.C. 1979. "The Limits to Programmer Productivity." In *Proceedings GUIDE and SHARE Application Development Symposium*. **

[**Jone80**] Jones, C.B. 1980. *Software Development: A Rigorous Approach*. Englewood Cliffs, NJ: Prentice Hall. **

[**Jone81**] Jones, T.C. November 1981. *Programmer Productivity Issues of the Eighties*. Washington, DC: IEEE Computer Society Press.

[**Jone89**] Jones, A.M., R.E. Bozeman, and W. McIver. 1989. "The Moorehouse Object-Oriented Reuse Library." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 456-462. Washington, DC: ACM Ada Technical Committee. **

[**Joyc87a**] Joyce, E. "Software Bugs: A Matter of Life and Liability." *Datamation*, (May 15, 1987):88-92.

[**Joyc87b**] Joyce, J., G. Lomow, K. Slind, and B. Ungar. "Monitoring Distributed Systems." *ACM: Transactions on Computer Systems*, 5/2 (May 1987):121-150.

[**Kafu81**] Kafura, D.G., and S.M. Henry. "Software Quality Metrics Based on Interconnectivity." *Journal of Systems and Software*, 2/2 (Jun 1981):121-131.

[**Kafu84**] Kafura, D.G., J.T. Canning, and G. Reddy. 1984. "The Independence of Software Metrics Taken at Different Life-Cycle Stages." In *Proceedings 9th Annual Software Engineering Workshop*, November 28, Greenbelt, MD. NASA/GSFC, 213-222. Also published as Virginia Polytechnic Institute, Technical Report TR-85-24. **

[**Kafu85a**] Kafura, D.G., and J.T. Canning. January 1985. *A Validation of Software Metrics Using Many Metrics and Many Resources*. Virginia Polytechnic Institute. TR-85-6.

[**Kafu85b**] Kafura, D.G., and G.R. Reddy. August 1985. *The Use of Software Quality Metrics In Software Maintenance*. Virginia Polytechnic Institute. Technical Report TR-85-33. Also published in *IEEE: Transactions on Software Engineering*, 13/3 (Mar 1987):335-343.

[**Kafu88**] Kafura, D.G., and J.T. Canning. 1988. *Using Group and Subsystem Level Analysis to Validate Software Metrics on Commercial Software*. Virginia Polytechnic Institute. TR-88-13.

[**Kahn77**] Kahn, G., and D. MacQueen. 1977. "Coroutines and Networks of Parallel Processes." In *Proceedings Information Processing (IFIP) Congress '77*, August 8-12, Toronto, Canada, 993-998. Amsterdam: North-Holland.

[**Kami80**] Kamin, S. 1980. "Final Data Type Specifications: A New Data Type Specification Method." In *Proceedings 7th ACM Annual Symposium on Principles of Programming Languages*, January 28-30, Las Vegas, NV. Baltimore, MD: ACM Order Department. **

[**Kant80**] Kant, K. 1980. "Error Recovery in Concurrent Processes." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 608-614. Los Alamitos, CA: IEEE Computer Society Press.

[**Kapp88**] Kappel, M.R., C.D. Ardoin, C.J. Linn, J.L. Linn, and J. Salasin. April 1988. *SAGEN User's Guide: Version 1.5*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2028.

[**Kato86**] Kato, T., K.S. Suzuki, and Y. Turano. 1986. "Conformance Testing for OSI Protocols in the Multiple Layer Environment Based on Automaton Models." In *Proceedings ICCC '86*, September, Munic,

Germany, 519-524. **

[Katz73] Katz, S.M., and Z. Manna. 1973. "A Heuristic Approach to Program Verification." In *Proceedings IFCAI-73*, August. **

[Katz86] Katz, E.E., H.D. Rombach, and V.R. Basili. 1986. "Structure and Maintainability of Ada Programs: Can We Measure the Differences?" In *Proceedings 9th Minnowbrook Workshop on Software Performance Evaluation*, August 5-8, Blue Mountain Lake, NY. **

[Katz87] Katz, E.E., and V.R. Basili. 1987. "Examining the Modularity of Ada Programs." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 390-396. Washington, DC: ACM Ada Technical Committee. Previously published in *IEEE: Computer*, 18/9 (Sep 1985):53-65.

[Kauf87a] Kaufmann, M., and W.D. Young. May 1987. "Comparing Gypsy and the Boyer-Moore Logic." University of Texas at Austin. Technical Report 59. **

[Kauf87b] Kaufmann, M., and W.D. Young. 1987. "Comparing Specification Paradigms for Secure Systems: Gypsy and the Boyer-Moore Logic." In *Proceedings 10th National Computer Security Conference*, September 21-24, Baltimore, MD. **

[Kear85] Kearney, J.K., R.L. Sedlmeyer, W.B. Thompson, M.A. Adler, and M.A. Gray. 1985. "Problems with Software Complexity Measurement." In *Proceedings 1985 ACM Computer Science Conference*, March, Cincinnati, OH, 340-347. **

[Kear86] Kearney, J.K., R.L. Sedlmeyer, W.B. Thompson, M.A. Gray, and M.A. Adler. "Software Complexity Measurement." *ACM: Communications of the ACM*, 29/11 (Nov 1986):1044-1050.

[Keil87] Keiler P.A., et al. 1987. "On the Quality of Software Reliability Prediction." In *Electronic System Effectiveness and Life Cycle Costing*, J.K. Skwirzynski, ed., NATO ASI Series, Vol. F3. Heidelberg: Springer-Verlag.

[Kell76] Keller, R.M. "Formal Verification of Parallel Programs." *ACM: Communications of the ACM*, 19/7 (Jul 1976):371-384.

[Kell82] Kelly, J.P.J. 1982. *Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach*. Ph.D. thesis, University of California at Los Angeles. **

[Kell83] Kelly, J.P, and A. Avizienis. 1983. "A Specification-Oriented Multi-version Software Experiment." In *Digest Papers FTCS-13: 13th International Conference on Fault-Tolerant Computing*, June, Milan, Italy, 120-126.

[Kell85a] Keller, S.E., and J.A. Perkins. 1985. "An Ada Measurement and Analysis Tool." In *Proceedings 3rd Annual National Conference on Ada Technology*, 188-196.

[Kell85b] Keller, S.E., and J.A. Perkins. 1985. "Ada Measurement Based on Software Quality Principles." In *Proceedings Washington Ada Symposium*, March, 195-203. New York: ACM.

[Kemm80] Kemmerer, R.A. 1980. *FDM - A Specification and Verification Methodology.* System Development Corp. Technical Report SP-488.

[Kemm81] Kemmerer, R. 1981. "Status Report on SDC's Formal Development Methodology." In *Proceedings 2nd Verification Workshop*, April, Gaithersburg, MD. **

[Kemm85a] Kemmerer, R.A. "Testing Formal Specifications to Detect Design Errors." *IEEE: Transactions on Software Engineering*, 11/1 (Jan 1985):32-43. Also published as University of California at Santa Barbara Technical Report 84-06, March 1984.

[Kemm85b] Kemmerer, R.A., and S.T. Eckmann. "UNISEX: A Unix-Based Symbolic Executor for Pascal." *Software Practice and Experience*, 15/5 (May 1985):439-457. **

[Kemm86] Kemmerer, R.A. 1986. *Verification Assessment Study Final Report, Vol. I: Overview, Conclusions and Future Directions*. National Computer Security Council. Technical Report C3-CR01-86.

[Kemm87] Kemmerer, R.A. 1987. "Analyzing Encryption Protocols Using Formal Verification Techniques." In *Proceedings CRYPTO '87*, Santa Barbara, CA, August. **

[Kenn75] Kennedy, K.W. 1975. "Node Listings Applied to Data Flow Analysis." In *Proceedings 2nd ACM Annual Symposium on Principles of Programming Languages*, January, Palo Alto, CA, 10-21. Baltimore, MD: ACM Order Department. **

[Kenn80] Kennaway, J.R., and C.A.R. Hoare. 1980. "A Theory of Nondeterminism." In *Automata, Languages and Programming*, J.W. de Bakker and J. van Leeuwen (eds.), *Lecture Notes in Computer Science, Vol. 85*. New York: Springer-Verlag. **

[Kern74a] Kernighan, B.W., and P.J. Plauger. 1974. *The Elements of Programming Style*. New York: McGraw-Hill.

[Kern74b] Kernighan, B.W., and P.J. Plauger. "Programming Style: Examples and Counterexamples." *ACM: Computing Surveys*, 6/4 (Dec 1974):303-319.

[Kern81] Kernighan, B.W., and P.J. Plauger. 1981. *Software Tools in Pascal*. Reading, MA: Addison Wesley.

[Kieb83] Kieburtz, R.B., and A. Silberschatz. "Access-Right Expressions." *ACM: Transactions on Programming Languages and Systems*, 5/1 (Jan 1983):78-96.

[King69] King, J.C. 1969. *A Program Verifier*. Ph.D. diss., Carnegie-Mellon University. **

[King70] King, J.C. 1970. "A Verifying Compiler." In *Debugging Techniques in Large Systems*. R. Rustin (ed.), 17-39. Englewood Cliffs, NJ: Prentice Hall. **

[King75a] King, J.C. 1975. "A New Approach to Program Testing." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 228-233. IEEE Cat. No. 75CH0940-7CSR.

[King75b] King, J.C. 1975. "Program Testing by Symbolic Execution." In *Proceedings Computer Science Conference*, February, 228-233. **

[King76] King, J.C. "Symbolic Execution and Program Testing." *ACM: Communications of the ACM*, 19/7 (Jun 1976):385-394.

[Kitc81] Kitchenham, B.A. "Measures of Programming Complexity." *ICL Technical Journal*, (May 1981):298-316.

[Knig84] Knight, J.C. 1984. "A Large Scale Experiment in N-Version Programming." In *Proceedings 9th Annual Software Engineering Workshop*, November 28, Greenbelt, MD. NASA/GSFC. **

[Knig85a] Knight, J.C., and P.E. Ammann. 1985. "An Experimental Evaluation of Simple Methods for Seeding Program Errors." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 337-342. Washington, DC: IEEE Computer Society Press.

[Knig85b] Knight, J.C., and V.S. Grine. 1985. *Symbolic Execution of Concurrent Ada Programs*. University of Virginia.

[Knig86a] Knight, J.C., and N.G. Leveson. "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming." *IEEE: Transactions on Software Engineering*, 12/1 (Jan 1986):96-109.

[Knig86b] Knight, J.C., and N.G. Leveson. 1986. "An Empirical Study of Failure Probabilities in Multi-Version Software." In *Proceedings 16th International Symposium on Fault-Tolerant Computing*, July, Vienna, Austria, 165-170. **

[Knij78] Knijff, D.J.J. van der. "Software Physics and Program Analysis." *Australian Computer Journal*, 10/3 (Aug 1978). **

[Knut71] Knuth, D.E. "An Empirical Study of FORTRAN Programs." *Software Practice and Experience*, 1/1 (Apr-Jun 1971):105-133.

[Knut73] Knuth, D.E., and F.R. Stevenson. "Optimal Measurement Points for Program Frequency Counts." *BIT*, 13/3 (1973):313-322.

[Koer84] Koerner, K., R. Mital, D.N. Card, and A. Maione. 1984. "An Evaluation of Programmer/Analyst Workstations." In *Proceedings 9th Annual Software Engineering Workshop*, November 28, Greenbelt, MD. NASA/GSFC. **

[Kopp76] Koppang, R.G. 1976. "Process Design System–An Integrated Set of Software Development Tools." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA, 86-90. Washington, DC: IEEE Computer Society Press.

[Kore84] Korelsky, T., and D. Sutherland. 1984. "Formal Specification of a Multi-Level Secure Operating System." In *Proceedings 1984 Symposium on Security and Privacy*, April, 209-218. **

[Kore85] Korel, B., and J.W. Laski. 1985. "A Tool for Data Flow Oriented Program Testing." In *Proceedings SoftFair II: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, December 2-5, San Francisco, CA, 34-37. **

[Kore86a] Korel, B. 1986. "A Program Error Localization Expert System." In *Proceedings Symposium on Application of Artificial Intelligence II*, April 1-3, Orlando, FL. **

[Kore86b] Korel, B. August 1986. *Dependence-Based Modeling in the Automation of Error Localization in Computer Programs*. Ph.d. thesis, Oakland University. **

[Kore87] Korel, B. "The Program Dependence Network in Static Program Testing." *Information Processing Letters*, 24/2 (1987):103-108. **

[Kore88] Korel, B., and J.W. Laski. 1988. "STAD: A System for Testing and Debugging: User Perspective." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 13-20. Washington, DC: IEEE Computer Society Press.

[Kosa72] Kosaraju, R. "Analysis of Structured Programs." *Journal Computer Systems Science*, 9/12 (Dec 1974):232-255. Also published by John Hopkins University, Technical Report 72-11, 1972. **

[Koss88] Koss, W.E. 1988. "Software Reliability Metrics for Military Systems." In *Proceedings Annual Reliability and Maintainability Symposium*, January, 190-194. **

[Kosy73] Kosy, D.W. 1973. "Approaches to Improved Program Validation Through Programming Language Design." In *Program Test Methods*, W.G. Hetzel (ed.). Englewood Cliffs, NJ: Prentice-Hall. **

[Krac78] Kracik, P.J. 1978. "An Example of Software Quality Assurance Techniques Used in a Successful Large Scale Development." In *Proceedings ACM Software Quality Assurance Workshop*, November 15-17, San Diego, CA, 181-186. New York: Association for Computing Machinery. **

[Krau73] Krause, K.W., R.W. Smith, and M.A. Goodwin. "Optimal Testing Through Automated Network Analysis." In *Conference Record 1973 IEEE Symposium on Computer Software Reliability*, April 30 - May 2, New York, 18-22. **

[Krau86] Krauser, E.W., and A.P. Mathur. 1986. "Program Testing on a Massively Parallel Transputer Based System." In *Proceedings ISMM International Symposium on Mini and Microcomputers and their Applications*, November 10-12, Austin, TX,67-71. **

[Krau88] Krauser, E.W., A.P. Mathur, and V. Rego. 1988. "High Performance Testing on SIMD Machines." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 171-177. Washington, DC: IEEE Computer Society Press.

[Krie80] Krieg-Brueckner, B., and D.C. Luckham. 1980. "Anna: Towards a Language for Annotating Ada Programs." In *Proceedings ACM-SIGPLAN Symposium on the Ada Programming Language*, December, Boston. Published in *ACM: SIGPLAN Notices*, 15/11 (Nov-Dec 1980):128-138. Also published in *ACM: SIGPLAN Notices*, 15/11 (Nov 1984):128-138.

[Krie83] Krieg-Brueckner, B. "Consistency Checking in Ada and Anna: A Transformational Approach." *ACM: Ada Letters*, III/2 (Sep-Oct 1983):46-54.

[Krie86] Krieg-Brueckner, B., H. Ganzinger, M. Broy, R. Wilhelm, U. Monche, B. Weisgerber, A.D. McGettrick, I.G. Campbell, and G. Winterstein. 1985. *PROgram Development by SPECification and TRAnsformation*. In *Proceedings Ada Europe Conference*, May, Edinburgh, Scotland, , 249-258. New York: Cambridge University Press

[Krog87] Kroger, F. 1987. *Temporal Logic of Programs*. Vol. 8 of *EATCS Monographs on Theoretical Computer Science*. Berlin: Springer-Verlag. **

[Krug88] Kruger, G.A. "Project Management Using Software Reliability Growth Models." *Hewlett-Packard Journal*, June 1988.

[Krus78] Kruszewski, G. 1978. "Software Reliability Modeling." In *Proceedings NAVSEA SWS RMQ Seminar*, September, 271-296. **

[Kuhn82] Kuhn, W.W. 1982. "A Software Lifecycle Case Study Using the PRICE Model." In *Proceedings IEEE NAECON 1982*, May. **

[Laem78] Laemmel, A., and M.L. Shooman. 1978. *Software Modeling Studies: Statistical (Natural) Language Theory and Computer Program Complexity*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-78-4. **

[Lam84] Lam, S.S., and A.U. Shankar. "Protocol Verification via Projections." *IEEE: Transactions on Software Engineering*, 10/4 (Jul 1984):325-342.

[Lamb78] Lamb, S.S., V.G. Leck, L.J. Peters, and G.L. Smith. 1978. "SAMM: A Modeling Tool for Requirements and Design Specification." In *Proceedings 2nd International Computer Software and Applications Conference*, November 13-16, Chicago, IL, 48-53. Long Beach, CA: IEEE Computer Society Press.

[Lamb83] Lamb, D.A. 1983. *Sharing Intermediate Representations: The Interface Description Language*. Carnegie-Mellon University. Technical Report CMU-CS-83-129. **

[Lamp77] Lamport, L. "Proving the Correctness of Multiprocess Programs." *IEEE: Transactions on Software Engineering*, 3/2 (Mar 1977):125-143.

[Lamp78] Lamport, L. "Time, Clocks and the Ordering of Events in Distributed Systems." *ACM: Communications of the ACM*, 21/7 (Jul 1978):558-564.

[Lamp79a] Lamport, L. "A New Approach to Proving the Correctness of Multiprocess Programs." *ACM: Transactions Programming Languages and Systems*, 1/1 (Jul 1979):84-97.

[Lamp79b] Lamport, L. "On the Proof of Correctness of a Calendar Program." *ACM: Communications of the ACM*, 22/10 (Oct 1979):554-557.

[Lamp80] Lamport, L. "The 'Hoare Logic' of Concurrent Programs." *Acta Informatica*, no. 14 (1980):21-37.

[Lamp82] Lamport, L., R. Shostak, and M. Pease. "The Byzantine Generals Problem." *IEEE: Transactions on Programming Language and Systems*, 4/3 (Jul 1982):382-401.

[Lamp83] Lamport, L. "Specifying Concurrent Program Modules." *ACM: Transactions on Programming Languages and Systems*, 5/2 (Apr 1983):190-222.

[Lamp84] Lamport, L., and F.B. Schneider. "The 'Hoare Logic of CSP' and All That." *ACM: Transactions on Programming Languages and Systems*, 6/2 (Apr 1984):281-296.

[Land77] Landrault, C., and J.-C. Laprie. 1977. "Reliability and Availability Modeling of Systems Featuring Hardware and Software Faults." In *Proceedings 7th International Conference on Fault-Tolerant Computing*, 10-15.

[Land79] Landry, S.P., and B.D. Shriver. "Simulated Execution of Dataflow Programs on Processors Having Finite Resources." *ACM: SIGSIM Simuletter*, 11 (Aug 1979):141-149. **

[Land86] Landwehr, C.E., J. McLean, S.L. Gerhart, Donald I. Good, and Nancy Leveson. "NRL Invitational Workshop on Testing and Proving: Two Approaches to Assurance." *ACM: SIGSOFT Software Engineering Notes*, 11/5 (Oct 1986):63-64.

[Lapr84] Laprie, J.-C. "Dependability Evaluation of Software Systems in Operation." *IEEE: Transactions on Software Engineering*, 10/6 (Nov 1984):701-714.

[Lask79] Laski, J.W. 1979. *A Hierarchical Approach to Program Testing*. University of Waterloo. Technical Report 55CFW130779. Also published in *ACM: SIGPLAN Notices*, 15/1 (Jan 1980):77-85.

[Lask82] Laski, J.W. "On Data Flow Guided Program Testing." *ACM: SIGPLAN Notices*, 17/9 (Sep 1982):62-71.

[Lask83] Laski, J.W., and B. Korel. "A Data Flow Oriented Program Testing Strategy." *IEEE: Transactions on Software Engineering*, 9/3 (May 1983):347-354.

[Lask86] Laski, J.W. "An Algorithm for the Derivation of Codefinitions in Computer Programs." *Information Processing Letters*, 23/2 (Aug 1986):85-90.

[Lask87] Laski, J.W. May 1987. *A Comparative Analysis of Some Data Flow Testing Strategies*. Oakland University, Technical Report TR-CSE-87-05. **

[Lask88a] Laski, J.W. 1988. "Testing in Top-Down Program Development." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 72-79. Washington, DC: IEEE Computer Society Press.

[Lask88b] Laski, J.W. "Data Testing in STAD." *Journal of Systems and Software*. To appear. **

[Lass79] Lassez, J.-L., and D.J.J. Van der Knijff. 1979. "Evaluation of Length and Level for Simple Program Schemes." In *Proceedings 3rd International Computer Software and Applications Conference*, November 6-8, Chicago, IL, 688-694. Long Beach, CA: IEEE Computer Society Press.

[Lass81] Lassez, J.-L., D.J.J. van der Knijff, and J. Sheppard. "A Critical Examination of Software Science." *Journal of Systems and Software*, 2/12 (Dec 1981):105-112.

[Late84] Latella, D. 1984. "User Interface of the ECSP Concurrent Debugger." In *MuTEAM: Distributed Multiprocessor Architecture and ECSP Concurrent Language*. Technoprint, 61-87. Bologna. **

[Lato89] Latour, L. 1989. "Ada, Hypertext, and Reuse." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 434-442. Washington, DC: ACM Ada Technical Committee. **

[Laue79] Lauesen, S. "Debugging Techniques." *Software Practice and Experience*, 9/1 (Jan 1979):51-63.

[Lave88] Lavender, R.G. 1988. *Issues Related to the Explication of Process-Product Relationships in DoD-STD-2167 and DoD-STD-2168* Virginia Polytechnic Institute. TR-85-6.

[Lawr81] Lawrence, M.J. 1981. "Programming Methodology, Organizational Environment, and Programming Productivity." *ACM: The Journal of Systems and Software*, 2 (1981):257-269.

[Lawr87] Lawrynuik, D. 1987. "The T-3 Testing Tool." In *Proceedings CIPS Edmonton Fall Conference*, Edmonton, Alberta, November 16-18. **

[Laws83] Lawson, D.J. 1983. "Failure Mode, Effect, and Criticality Analysis." *Electronic Systems Effectiveness and Life Cycle Costing*, J.K. Skwirzynski, ed., NATO ASI Series, Vol. F3, 55-74. Heidelberg: Springer Verlag. **

[LeDo85] LeDoux, C.H., and D.S. Parker. 1985. "Saving Traces for Ada Debugging." In *Proceedings SIGAda International Conference*, May, Paris, France. Published in *ACM: Ada Letters*,V/2 (Sep-Oct 1985):97-108.

[Leac87] Leach, R.J. 1987. "Ada Software Metrics and Their Limitations." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 285-293. Washington, DC: ACM Ada Technical Committee.

[Leac89] Leach, R.J. 1989. "Software Metric Analysis of the Ada Repository." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 270-277. Washington, DC: ACM Ada Technical Committee. **

[Lee88] Lee, J.A.N, and X. He. September 1988. *A Methodology for Test Selection*. Virginia Polytechnic Institute. TR-88-29.

[Lee89a] Lee, P.-N., and A. Tamboli. 1989. "Ada Implementation of Sequential Correspondent Operations for Software Fault Tolerance." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 278-283. Washington, DC: ACM Ada Technical Committee. **

[Lee89b] Lee, A.J., W.R. Macre, and D.K. Doi. 1989. "Benchmarking the Real-Time Performance of Dynamic Ada Processes." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 132-138. Washington, DC: ACM Ada Technical Committee. **

[Lefk89] Lefkowitz, S., H. Greene, and M. Bender. 1989. "Establish and Evaluate Ada Runtime Features of Interest for Real-Time Systems." In **

[Lehm80] Lehman, M.M. "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle." *Journal of Systems and Software*, 1/3 (1980). **

[Less80] Lesser, V.R., P.C. Bates, R. Brooks, D. Corkill, L. Lefkowitz, R. Mukunda, J. Pavlin, S. Reed, and J.C. Wileden. 1981. *A High Level Simulation Testbed for Cooperative Distributed Problem Solving*. University of Massachusetts. Technical Report TR-81-16. **

[Less81] Lesser, V.R., and D.D. Corkill. "Functionally Accurate, Cooperative Distributed Systems." *IEEE: Transactions on Systems, Man and Cybernetics*, SMC-11/1 (Jan 1981):81-96.

[Leun88] Leung, H.K., and L.J. White. September 1988. *An Study of Regression Testing*. University of Alberta. Technical Report 88-15.

[Leve83a] Leveson, N.G., T. Shimeall, J. Stolzy, and J. Thomas. 1983. "Design for Safe Software." In *Proceedings AIAA Space Science Meeting*, January, Reno, NV.

[Leve83b] Leveson, N.G., and P.R. Harvey. "Analyzing Software Safety." *IEEE: Transactions on Software Engineering*, 9/5 (Sep 1983):569-579.

[Leve83c] Leveson, N.G., and J.L. Stolzy. "Safety Analysis of Ada Programs Using Fault Trees." *IEEE: Transactions on Reliability*, R-32/5 (Dec 1983):479-484.

[Leve83d] Leveson, N.G. "Verification of Safety." In *Proceedings International IFAC Workshop on Achieving Safe Real-time Computer Systems*. September, Cambridge, England. **

[Leve86a] Levendel, Y., and P.R. Menon. 1986. "Fault Simulation." In *Fault-Tolerant Computing*, D.K. Pradham (ed.) Vol. 1, Chapter 3, 184-264. Englewood Cliffs, NJ: Prentice Hall. **

[Leve86b] Leveson, N.G. "Software Safety: Why, What, and How." *ACM: Computing Surveys*, 18/2 (Jun 1986):125-163.

[Leve87] Leveson, N.G., and J.L. Stolzy. "Safety Analysis Using Petri Nets." *IEEE: Transactions on Software Engineering*, 13/3 (Mar 1987):386-397.

[Leve89a] Leveson, N.G. "Safety as a Software Quality." *IEEE: Software*. 6/3 (May 1989):88-89. **

[Levi78] Levitt, K.N. 1978. "A Panel Session - Formal Methods in Programming–When will they be practical?" In *Proceedings AFIPS National Computer Conference*, vol. 47, June 5-8, Anaheim, CA, 665-668. Arlington, VA: AFIPS Press.

[Levi80] Levin, G.M. August 1980. *Proof Rules for Communicating Sequential Processes*. Ph.D. diss., Cornell University.

[Levi81] Levin, G.M., and D. Gries. "A Proof Technique for Communicating Sequential Processes." *Acta Informatica*, no. 15 (1981):281-302.

[Levy84] Levy, M.R. 1984. "Type Checking, Separate Compilation and Reusability." In *Proceedings ACM-SIGPLAN '84 Symposium on Compiler Construction*, June, Montreal. Published in *ACM: SIGPLAN Notices*, 19/6 (Jun 1984):285-289.

[Lew88] Lew, K.S., T.S. Dillon, and K.E. Forward. "Software Complexity and Its Impact on Software Reliability." *IEEE: Transactions on Software Engineering*, 14/11 (Nov 1988):1645-1655.

[Li87] Li, H.F., and W.K. Cheung. "An Empirical Study of Software Metrics." *IEEE: Transactions on Software Engineering*, 13/6 (Jun 1987):697-708.

[Lieb80] Lieberman, H., and C. Hewitt. 1980. "A Session with TINKER: Interleaving Program Testing with Program Design." In *Proceedings 1980 LISP Conference*, August, Stanford University. **

[Ligh76] Light, W. 1976. "Software Reliability/Quality Assurance Practices." In *Proceedings AIAA Conference on Computers in Aerospace*. **

[Ligo87] Ligon, W.E. 1987. *An Efficient Method for Executing Multiple Mutants on a Vector Processor*. Georgia Institute of Technology. **

[Lin85] Lin, H. June 1985. *Software for Ballistic Missile Defense*. Center for International Studies. Massachusetts Institute of Technology. **

[Lind76] Linden, T.A. 1976. "The Use of Abstract Data Types to Simplify Program Modifications." In *Proceedings Conference on Data-Abstraction, Definition and Structure*. Published in *ACM: SIGPLAN Notices* 11 (1976):12-23. **

[Lind85] T.E. Lindquist, and J.L. Facemire. 1985. "Using an Ada-Based Abstract Machine Description of CAIS to Generate Validation Tests." In *Proceedings Washington Ada Symposium*, March 24-26, John Hopkins Applied Physics Laboratory, Laurel, MD, 173-178. Washington, DC: ACM Ada Technical Committee: ACM.

[Lind87] T.E. Lindquist, P.K. Lawlis, and D.P. Levine. 1987. "Typing Information in a Software Engineering Environment." In *Proceedings 6th International Conference on Entity-Relationship Approach*. November. **

[Lind88a] Lindquist, T.E., and J.R. Jenkins. "Test-Case Generation with IOGen." *IEEE: Software*, 5/1 (Jan 1988):72-79.

[Lind88b] Lindquist, T.E., I.S. Kwon, and V.L. Wood. *Test Case Generation for Ada Exceptions and Tasking*. In preparation. **

[Lind88c] Lindquist, T.E. 1988. Methods and Tools for Increasing Reliability of Embedded Ada Systems." In *Towards SDS Testing and Evaluation: A Collection of Relevant Topics*. IDA Memorandum Report M-513. Alexandria, VA: Institute for Defense Analyses. Draft.

[Lind88d] Lindsay, P.A. "A Survey of Mechanical Support for Formal Reasoning." *Software Engineering Journal*, 3/1 (Jan 1988).

[Lind89] Lind, R.K., and K. Vairavan. "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort." *IEEE: Transactions on Software Engineering*, 15/5 (May 1989):649-653.

[Ling79] Linger, R.C., H.D. Mills, and B.I. Witt. 1979. *Structured Programming: Theory and Practice*. The Systems Programming Series. Reading, MA: Addison Wesley.

[Linn88] Linn, J.L, C.D. Ardoin, C.J. Linn, S.H. Edwards, M.R. Kappel, and J. Salasin. April 1988. *Strategic Defense Initiative Architecture Dataflow Modeling Technique: Version 1.5*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2035.

[Lipo73] Lipow, M. May 1973. *Application of Algebraic Methods to Computer Program Analysis*. TRW Software Series. Report TRW-SS-73-10. **

[Lipo77] Lipow, M., and T.A. Thayer. "Prediction of Software Failures." In *Proceedings Annual Reliability and Maintainability Symposium*, 489-494. **

[Lipo79] Lipow, M. "Prediction of Software Errors." *Journal of Systems and Software*, 1 (1979):71-75. **

[Lipt78] Lipton, R.J., and F.G. Sayward. 1978. "The Status of Research on Program Mutation." In *Digest IEEE Workshop on Software Testing and Test Documentation*, December 18-20, Ft. Lauderdale, FL, 355-378. IEEE Computer Society Technical Committee on Software Engineering. **

[Lisk75] Liskov, B.H, and S.N. Zilles. "Specification Techniques for Data Abstractions." *IEEE: Transactions on Software Engineering*, 1/1 (1975):7-19.

[Lisk79] Liskov, B.H., and V. Berzins. 1979. "An Appraisal of Program Specifications." In *Research Directions in Software Technology*, P. Wegner (ed.), 276-301. Cambridge, MA: MIT Press. **

[List82] Lister, A.M. "Software Science–The Emperor's New Clothes?" *Australian Computer Journal*, 14/2 (May 1982):66-71. **

[Lite76] Litecky, C.R., and G.B. Davis. "A Study of Errors, Error-Proneness, and Error Diagnosis in Cobol." *ACM: Communications of the ACM*, 19/1 (Jan 1976):33-37.

[Litt73] Littlewood, B., and J.L. Verrall. "A Bayesian Reliability Growth Model for Computer Software." *The Journal of the Royal Statistical Society*, Series C, 22/3 (1973):332-346.

[Litt75] Littlewood, B. 1975. "A Reliability Model for Markov Structured Software." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 204-207. IEEE Cat. No. 75CH0940-7CSR.

[Litt76] Littlewood, B. 1976. "A Semi-Markov Model for Software Reliability with Failure Costs." In *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York218-300. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press. **

[Litt78] Littlewood, B. 1978. "How to Measure Software Reliability, and How Not To..." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 37-45. Washington, DC: IEEE Computer Society Press.

[Litt79] Littlewood, B. "Software Reliability Model for Modular Program Structure." *IEEE: Transactions on Reliability*, R-28/3 (Aug 1979):241-246.

[Litt80a] Littlewood, B. 1980. "What Makes a Reliable Program – Few Bugs, or a Small Failure Rate?" In *Proceedings AFIPS National Computer Conference*, vol. 49, May 19-22, Anaheim, CA, 707-713. Arlington, VA: AFIPS Press.

[Litt80b] Littlewood, B. "Theories of Software Reliability: How Good Are They and How Can They Be Improved?" *IEEE: Transactions on Software Engineering*, 6/5 (Sep 1980):498-500.

[Litt80c] Littlewood, B. 1980. "A Bayesian Differential Debugging Model for Software Reliability." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 511-519. Los Alamitos, CA: IEEE Computer Society Press. **

[Litt81a] Littlewood, B. "Stochastic Reliability Growth: A Model for Fault Removal in Computer Programs and Hardware Designs." *IEEE: Transactions on Reliability*, R-30/4 (1981):313-320.

[Litt81b] Littlewood, B. 1981. "A Critique of the Jelinski-Moranda Model for Software Reliability." In *Proceedings Annual Reliability and Maintainability Symposium*, 357-364. **

[Lo83] Lo, P., and D. Wyckoff. July 1983. *Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-81-102. **

[Loca80] Locasso, R., J. Scheid, V. Shorre, and P. Eggert. November 1980. *The Ina Jo Specification Language Reference Manual*. Santa Monica, CA: System Development Corp. SDC Document TM-6889/000/01. **

[Lohs84] Lohse, J.B., and S.H. Zweben. "Experimental Evaluation of Software Design Principles." *Journal of Systems and Software*, 4/4 (Nov 1984):301-308.

[Lond70] London, R. "Bibliography on Proving the Correctness of Computer Programs." *Machine Intelligence*, (1970). **

[Lond71] London, R.L. 1971. "Software Reliability through Proving Programs Correct." In *Proceedings IEEE International Symposium on Fault-Tolerant Computing*, March. **

[Lond75] London, R.L. 1975. "A View of Program Verification." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 534-545. IEEE Cat. No. 75CH0940-7CSR.

[Lond85] London, R.L., and R.A. Duisberg. "Animating Programs using Smalltalk." *IEEE: Computer*, 18/8 (Aug 1985):61-71.

[Long77] Long, A.B., C.V. Ramamoorthy, S.F. Ho, H.H. So, H.L. Reeves, and E.A. Straker. 1977. "A Methodology for the Development and Validation of Critical Software for Nuclear Power Plants." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 620-627. Long Beach, CA: IEEE Computer Society Press.

[Long88] Long, D.L., and L.A. Clarke. 1988. "Task Interaction Graphs for Concurrency Analysis." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 132-133. Washington, DC: IEEE Computer Society Press.

[Love76] Love, L.T., and A.B. Bowman. "An Independent Test of the Theory of Software Physics." *ACM: SIGPLAN Notices*, 11/11 (Nov 1976):42-49.

[Love77a] Love, L.T. 1977. *Relating Individual Differences in Computer Programming Performance to Human Information Processing Abilities*. Ph.D. diss., University of Washington, Dept. Psychology. **

[Love77b] Love, L.T. 1977. "An Experimental Investigation of the Effect of Program Structure on Program Understanding." *ACM: SIGPLAN Notices*, 12/3 (Mar 1977):105-113.

[Luck77] Luckham, D.C. 1977. "Program Verification and Oriented Programming." In *Proceedings Information Processing (IFIP) Congress '77*, August 8-12, Toronto, Canada, 783-793. Amsterdam: North-Holland.

[Luck79a] Luckham, D.C., and N. Suzuki. "Verification of Array, Record and Pointer Operations in Pascal." *ACM: Transactions on Programming Languages and Systems*, 1/2 (Oct 1979):226-244.

[Luck79b] Luckham, D.C., S.M. German, F.W. von Henke, R.A. Karp, P.W. Milne, D.C. Oppen, W. Polak, and W.L. Scherlis. March 1979. *Stanford Pascal Verifier User Manual*. Stanford University. Technical Report Program Verification Report PV-11, CSD Report STAN-CS-79-731. **

[Luck80a] Luckham, D.C., and W. Polak. 1980. "A Practical Method of Documenting and Verifying Ada Programs with Packages." In *Proceedings ACM-SIGPLAN Symposium on the Ada Programming Language*, December, Boston. Published in *ACM: SIGPLAN Notices*, 15/11 (Nov-Dec 1980):113-122.

[Luck80b] Luckham, D.C., and W. Polak. "Ada Exception Handling: An Axiomatic Approach." *Transactions on Programming Languages and Systems*, 2/2 (Apr 1980):225-233.

[Luck80c] Luckham, D.C., and W. Polak. February 1980. *Ada Exceptions: Specification and Proof Techniques*. Stanford University. Program Verification Group Report PVG-16, CSD Report STAN-CS-80-789. **

[Luck81] Luckham, D.C., H.J. Larsen, D.R. Steveson, and F.W. von Henke. July 1981. *ADAM -- An Ada Based Language for Multi-Processing*. Stanford University. Technical Report STAN-CS-81-867, updated and republished as Technical Report CSL-TR-83-240, May 1983. Also published in *Software Practice and Experience*, (Jul 1984):605-642. **

[Luck84a] Luckham, D.C., and F.W. von Henke. September 1984. *An Overview of ANNA A Specification Language for Ada*. Stanford University. Technical Report CSL-TR-84-265. Also published in *IEEE: Software*, 2/2 (Mar 1985):9-24.

[Luck84b] Luckham, D.C., F.W. von Henke, B. Krieg-Brueckner, and O. Owe. July 1984. *ANNA A Language for Annotating Ada Programs*. Stanford University. **

[Luck85] Luckham, D.C. December 1985. "ANNA, A Specification Language for Ada." In *Proceedings 1st IDA Workshop on Formal Specifications and Verification of Ada*. Alexandria, VA: Institute for Defense Analyses. IDA Memorandum Report M-146.

[Luck86a] Luckham, D.C., R. Neff, and D. Rosenblum. August 1986. *An Environment for Ada Software Development Based on Formal Specification*. Stanford University. Technical Report CSL-TR-86-305. Also published in *ACM: Ada Letters*, VII/3 (May-June 1987):94-106.

[Luck86b] Luckham, D.C., Y. Huh, S. Ghosh, and A. Stanculescu. 1986. *Analysis of the VHSIC Hardware Description Language*. Stanford University. **

[Luck86c] Luckham, D.C., A. Stanculescu, Y. Huh, and S. Ghosh. 1986. "The Semantics of Timing Constructs in Hardware Description Languages." In *Proceedings IEEE International Conference on Computer Design ICCD '86*, October 10-14. Also published as Stanford University Technical Report PAVG-32. **

[Luck87] Luckham, D.C., D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. July 1987. "Task Sequencing Language for Specifying Distributed Ada Systems." In *Proceedings of CRAI Workshop on Software Factories and Ada*, Capri, Italy, A.N. Habermann and U. Montanari (eds). Lecture Notes on Computer Science, 275. Springer-Verlag, 249-305. Also published as Stanford University Technical Report CSL-TR-87-334.

[Luke80] Lukey, F.J. "Understanding and Debugging Programs." *International Journal on Man-Machine Studies*, 12/2 (Feb 1980):189-202. **

[Lync81] Lynch, W.C., and J.C. Browne. 1981. "Performance Evaluation: A Software Metrics Success Story." In *Software Metrics*, Perlis et al (ed.), 171-183. MIT Press. **

[MIL85] Military Standard. 1985. *Technical Reviews and Audits for Systems, Equipment, and Computer Programs*. MIL-STD-1521.

[MacL82] MacLean, J. September 1982. *A Formal Foundation for the Trace Method of Software Specification*. Washington, DC: Naval Research Laboratory. NRL Memorandum Report 4874. **

[Mait80] Maitland, R. 1980. "NODAL." In *NBS Software Tools Database*, R. Houghton, and K. Oakley (eds.). Gaithersburg, MD: National Bureau of Standards. **

[Majo83] Majoros, M., and H.M. Sneed. 1983. "Testing Programs Against a Formal Specification." In *Proceedings 7th International Computer Software and Applications Conference*, Novemer 7-11, Chicago, IL, 512-519. Los Angeles, CA: IEEE Computer Society.

[Manc83] Mancarella, P., and F. Turini. 1983. "A High Level Analysis Tool for Concurrent Programs." In *Proceedings 1983 International Conference on Parallel Processing*, August, Bellaire, MI. **

[Mand85] Mandriolo, D., R. Zicari, C. Ghezzi, and F. Tisato. "Modeling the Ada Task System by Petri Nets." *Computer Languages*, 10/1 (1985):43-61.

[Mann70] Manna, Z., and A. Pnueli. "Formalization of Properties of Functional Programs." *Journal of the ACM*, 17/3 (1970):555-569.

[Mann74] Manna, Z. 1974. *Mathematical Theory of Computation*. New York: McGraw-Hill.

[Mann78] Manna, Z. and R. Waldinger. "Is 'Sometime' Sometimes Better than 'Always'?: Intermittent Assertions in Proving Program Correctness." *ACM: Communications of the ACM*, 21/2 (Feb 1978):159-172.

[Mart70] Martyn, J., and B.C. Vickery. "The Complexity of the Modeling of Information Systems." *Journal of Documentation*, 26/3 (Sep 1970):204-220. **

[Mart83] Martin, D.J. 1983. "Dissimilar Software in High Integrity Applications in Flight Controls." In *Software in Avionics: AGARD Conference Proceedings 330*, January, The Hague, The Netherlands, 36.1-36.9. **

[Math86] Mathur, A.P., and E.W. Krauser. 1986. *Modeling Mutation on a Vector Processor*. Georgia Institute of Technology. Technical Report GIT-SERC-87/07. **

[Math87a] Mathur, A.P., E. Galiano, W. Ligon, and T. Greenlaw. 1987. *Concurrent Execution Over Multiple Data Sets on Vector Processors*. Purdue University. Technical Report SERC-TR-7-P.

[Math87b] Mathur, A.P., and E. Galiano. November 1987. *Inducing Vectorization: A Formal Analysis*. Purdue University. Technical Report SERC-TR-6-P.

[Math88a] Mathur, A.P., and E.W. Krauser. April 1988. *Mutant Unification for Improved Vectorization*. Purdue University. Technical Report SERC-TR-14-P.

[Math88b] Mathur, A.P. *An Empirical Basis for Program and Mutant Unification*. Under preparation. **

[Maug85] Mauger, C., and K. Pammett. 1985. "An Event-Driven Debugger for Ada." In *Proceedings SIGAda International Conference*, May, Paris, France. Published in *ACM: Ada Letters*,V/2 (Sep-Oct 1985):124-134.

[Maye89] Mayes, L., R.W. Aragon, D. Terrien, and J. Trost. 1989. "Automatic Test Data Generation and Assertion Testing for Ada Program Units." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 537-547. Washington, DC: ACM Ada Technical Committee. **

[Mayf85] Mayfield, W.T., and S.R. Welke, eds. November 1985. *Proceedings 2nd IDA Workshop on Formal Specifications and Verification of Ada*. Alexandria, VA: Institute for Defense Analyses. IDA Memorandum Report M-135.

[Mayf86] Mayfield, W.T., et al, eds. August 1986. *Proceedings 3rd IDA Workshop on Formal Specifications and Verification of Ada*. Alexandria, VA: Institute for Defense Analyses. IDA Memorandum Report M-241.

[McCa76] McCabe, T.J. "A Complexity Measure." *IEEE: Transactions on Software Engineering*, 2/4 (Dec 1976):308-320.

[McCa77a] McCall, J.A., P.K. Richards, and G.F. Walters. November 1977. *Factors in Software Quality, Vols. I, II and III*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-77-369.

[McCa77b] McCall, J.A., P. Richards, and G. Walters. 1977. "Metrics for Software Quality Evaluation and Prediction." In *Proceedings 2nd Annual Software Engineering Workshop*. Greenbelt, MD. NASA/GSFC. **

[McCa78] McCall, J.A. 1978. "The Utility of Software Quality Metrics in Large-Scale Software System Development." In *Proceedings U.S. Army Computer Systems Command Software Life Cycle Management Workshop*, August 21-22. **

[McCa79] McCall, J.A. 1979. "An Introduction to Software Quality Metrics." In *Software Quality Management*, J.D. Cooper and M.J. Fisher (eds.), 127-142. Petrocelli Books, Inc.

[McCa80a] McCall, J.A., and M. T. Matsumoto. April 1980. *Software Quality Metrics Enhancements, Vol. I.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-80-109.

[McCa80b] McCall, J.A., and M. T. Matsumoto. April 1980. *Software Quality Measurement Manual, Vol. II.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-80-109.

[McCa82a] McCabe, T.J. (ed.) 1982. *Structured Testing*. IEEE Computer Society Press, IEEE Catalog No. EH02006.

[McCa82b] McCabe, T.J., and G.G. Schulmeyer. 1982. "System Testing Aided by Structured Analysis (A Practical Experience)." In *Proceedings 6th International Computer Software and Applications Conference*, March 9-12, San Diego, CA. Los Angeles, CA: IEEE Computer Society.

[McCa82c] McCabe, T.J. 1982. *Structured Testing: A Testing Methodology Using the McCabe Complexity Metric*. Gaithersburg, MD: National Bureau of Standards. Special Publication 500-99. **

[McCa84] McCall, J.A., and M.A. Herndon. 1984. "Controlling the Reliability of Software During the O&M Phase." In *Proceedings Annual Reliability and Maintainability Symposium*, 275-281. **

[McCa87a] McCall, J.A., W. Randall, C. Bowen, N. McKelvey, R. Senn, J. Morris, H. Hecht, S. Fenwick, P. Yates, M. Hecht, and R. Vienneau. November 1987. *Methodology for Software Reliability Prediction, Vol. I.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-87-171.

[McCa87b] McCall, J.A., W. Randall, C. Bowen, N. McKelvey, R. Senn, J. Morris, H. Hecht, S. Fenwick, P. Yates, M. Hecht, and R. Vienneau. November 1987. *Methodology for Software Reliability Prediction, Vol. II*, Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TC-87-171.

[McCl76] McClure, C.L. 1976. *Formalization and Application of Structured Programming and Program Complexity*. Ph.D. thesis, Illinois Institute of Technology. **

[McCl78a] McClure, C.L. 1978. "A Model for Program Complexity Analysis." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 149-157. Washington, DC: IEEE

Computer Society Press.

[McCl78b] McClure, C.L. 1978. *Reducing COBOL Complexity through Structured Programming*. New York: Van Nostrand Reinold.

[McDa77] McDaniel, G. 1977. "METRIC: A Kernel Instrumentation System for Distributed Environments." In *Proceedings 6th ACM Symposium on Operating System Principles*, November, West Lafayette, IN. **

[McGa82] McGarry, F.E. 1982. "Measuring Software Development Technology." In *Proceedings 7th Annual Software Engineering Workshop*, Greenbelt, MD. NASA/GSFC. **

[McGa84] McGarry, F.E., G. Page, D.N. Card, et al. February 1984. *An Approach to Software Cost Estimation*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-83-001. **

[McGa85a] McGarry, F.E. 1985. "Recent SEL Studies." In *Proceedings 10th Annual Software Engineering Workshop*, December, Greenbelt, MD. NASA/GSFC. **

[McGa85b] McGarry, F.E., J. Valett, and D. Hall. January 1985. "Measuring the Quality Impact of Computer Resource Quality on the Software Development Process and Product." In *Proceedings IEEE 18th Hawaii International Conference on System Sciences*, January, Honolulu, HA, 533-541.

[McGe82] McGettrick, A.D. 1982. *Program Verification Using Ada*. Cambridge Computer Science Texts No. 13. Cambridge University Press. **

[McGi77] McGibbon, T.L., et al. October 1977. *Pattern Recognition Methods for Determining Software Quality*. Griffiss Air Force Base, NY: Rome Air Development Center. **

[McIn83] McIntree, J.W. Jr. March 1983. *Fault Tree Techniques as Applied to Software (Soft Tree)*. USAF. **

[McMu80] McMullin, P.R., and J.D. Gannon. December 1980. *Evaluating a Data Abstraction Testing System Based on Formal Specifications*. University of Maryland. Technical Report TR-993. Also published in *Journal of Systems and Software*, 2 (1981):177-186.

[McMu82] McMullin, P.R. 1982. *DAISTS: A System for Using Specifications to Test Implementations*. Ph.D. diss., University of Maryland.

[McMu83] McMullin, P.R., and J.D. Gannon. "Combining Testing with Formal Specifications: A Case Study." *IEEE: Transaction on Software Engineering*, 9/3 (May 1983):328-334.

[McTaXX] McTap, J.L. "The Complexity of an Individual Program." **

[McWe84] McWethy, S., and J. Radatz. August 1984. *Software Quality Engineering Handbook*. USACSC. **

[Mear81] Mearns, I. October 1981. *A Message-Based Run-Time System and Proof Rules for Ada Tasks*. M.S. diss., University of Manchester. **

[Mear83] Mearns, I. October 1983. *A Denotational Semantics for Concurrent Ada Programs*. Ph.D. diss, University of Manchester. **

[Medi81] Medina-Mora, R., and P.H. Feiler. "An Incremental Programming Environment." *IEEE Transactions on Software Engineering*, 7/5 (May 1981):472-482.

[Meld88] Meldal, S., D. Luckham, and M.A. Haberler. 1988. "Specifying Ada Tasking Using Patterns of Behavior." In *Proceedings IEEE 21st Hawaii International Conference on System Sciences*, January, Honolulu, HA, 129-134. **

[Mell82] Melliar-Smith, P.M., and R.L. Schwartz. "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight-Control System." *IEEE Transactions on Computers*, C-31/7 (Jul 1982):616-630.

[Mend79] Mendis, K.S., and M.L. Gollis. 1979. "Categorizing and Predicting Errors on Software Programs." In *Proceedings 2nd AIAA Conference on Computers in Aerospace*, October, Los Angeles, CA, 300-308. **

[Meye67] Meyer, A.R., and D.M. Ritchie. 1967. "The Complexity of Loop Programs." In *Proceedings ACM 22nd Annual National Computer Conference*, 465-470. Washington, DC: Thompson Books.

[Miar83] Miara, R.J., J.A. Musselman, J.A. Navarro, and B. Shneiderman. "Program Indentation and Comprehensibility." *ACM: Communications of the ACM*, 26/11 (Nov 1983):861-867.

[Mign82] Migneault, G.E. September 1982. *The Cost of Software Fault Tolerance Techniques*. NASA Technical Memorandum 84546. Also published in *Software in Avionics: AGARD Conference Proceedings 330*, January, The Hague, The Netherlands, 37:1-37:8. **

[Mili84] Mili, A., and J. Desharnais. 1984. "A System for Classifying Program Verification Methods: Assessing Meaning of Program Verification Methods." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 499-509. Washington, DC: IEEE Computer Society Press.

[Mill71] Mills, H.D. 1971. "Top Down Programming in Large Systems." In *Debugging Techniques in Large Systems, 1st Courant Computer Science Symposium*. NYU Ed. Randell Rustin (ed.). Englewood Cliffs, NJ: Prentice-Hall. **

[Mill72a] Mills, H.D. 1972. *Chief Programmer Teams: Principles and Procedures*. Gaithersburg, MD: IBM Corp. Technical Report FSC-71-6012. **

[Mill72b] Mills, H.D. 1972. *Mathematical Foundations for Structural Programming*. Gaithersburg, MD: IBM Corp. Technical Report FSL-72-6021. **

[Mill72c] Miller, E. Jr., et al. October 1972. *A Survey of Major Techniques for Program Validation*. GRC RM-1731. **

[Mill72d] Mills, H.D. 1972. *On Statistical Validation of Computer Programs*. Gaithersburg, MD: IBM Federal Systems Division. IBM Report FSC72-6015. **

[Mill74a] Miller, E.F., M.R. Paige, J.P. Benson, and W.R. Wisehart. 1974. "Structural Techniques of Program Validation." In *Proceedings 19th IEEE Computer Society International Conference*, 161-164.

[Mill74b] Miller, E. Jr., et al. 1974. "Structurally Based Automatic Program Testing." In *Proceedings EASCON-74*, October 7-9, Washington, DC. **

[Mill74c] Miller, E.F. October 1974. *Overview and Status -- Program Validation Project*. General Research Corp. **

[Mill74d] Miller, E.F. Jr. 1974. *RXVP, FORTRAN Automated Verification System*. Santa Barbara, CA: General Research Corp., Program Validation Project Report. **

[Mill75a] Mills, H. "The New Math of Computer Programs." *ACM: Communications of the ACM*, 18/1 (Jan 1975):43-48.

[Mill75b] Miller, E.F. 1975. "Engineering Software for Testability." In *Proceedings 20th IEEE Computer Society International Conference*, 7-10.

[Mill75c] Miller, E.F., and R.A. Melton. 1975. "Automated Generation of Testcase Datasets." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA. IEEE Cat. No. 75CH0940-7CSR. Also published in *ACM: SIGPLAN Notices*, 10/6 (Jun 1975):51-58.

[Mill75d] Mills, H.D. 1975. "How to Write Correct Programs and Know It." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 363-370. IEEE Cat. No. 75CH0940-7CSR.

[Mill75e] Miller, E.F. Jr. June 1975. *Methodology for Comprehensive Software Testing*. Santa Barbara, CA: General Research Corp. **

[Mill75f] Miller, E.F. 1975. "RXVP: An Automated Verification System for FORTRAN." In *4th Conference on Computer Science and Statistics: Proceeedings 8th Symposium on the Interface*, February, Los Angeles, CA, 328. Springer-Verlag. **

[Mill77a] Miller, E. "Program Testing: Art Meets Theory." *IEEE: Computer*, 10/7 (Jul 1977):42-51.

[Mill77b] Miller, E. "Program Testing Tools – A Survey" 1977. In *Proceedings MIDCON '77*, 1-14.

[Mill79a] Miller, E. 1979. "Program Testing Technology in the 1980's." The Oregon Report: In *Proceedings Conference on Computing in the 1980's*, 72-79. Washington, DC: IEEE Computer Society Press.

[Mill79b] Miller, E.F. "Some Statistics from the Software Testing Service." *ACM: SIGSOFT Software Engineering Notes*, 4/1 (Jan 1979):8-11.

[Mill79c] Miller, E. "Software Testing and Test Documentation." *IEEE: Computer*, 12/3 (Mar 1979):98-107.

[Mill80a] Miller, E. "Coverage Measure Definitions Revisited." *Testing Technical Newsletter*, 3/4 (1980):6. **

[Mill80b] Mills, H.D. "Function Semantics for Sequential Programs." *Information Processing*, Vol. 80 (1980). **

[Mill80c] Miller, A.M. November 1980. *A Study of the Musa Reliability Model*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-80-005. **

[Mill80d] Miller, R.E., and C.K. Yap. "On Formulating Simultaneity Studying Parallelism and Synchronization." *Journal of Computer Systems Science*. 20/2 (Apr 1980):203-218. **

[Mill81a] Miller, E., and W.E. Howden, eds. 1981. *Tutorial: Software Testing and Validation*, 2nd Edition. Los Alamitos, CA: IEEE Computer Society Press.

[Mill81b] Millen, J.K., and D.L. Drake. "An Experiment with AFFIRM and HDM." *Journal of Systems and Software*, 2 (1981):159-175.

[Mill83] Mills, H.D. 1983. "Software Productivity in the Enterprise." In *Software Productivity*, 265-270. New York: Little, Brown. **

[Mill84] Miller, B.P. 1984. *Performance Characterization of Distributed Programs*. Ph.D. diss., University of California (Berkely).

[Mill85] Miller, D.R., and A. Sofer. 1985. "Completely Monotone Regression Estimates of Software Failures Rates." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 343-348. Washington, DC: IEEE Computer Society Press.

[Mill86] Miller, D.R., and A. Sofer. "A Non-Parametric Approach to Software Reliability, Using Complete Monotonicity." In *Reliability: State of the Art*, A. Bendell and P. Mellor (eds.), 31-44. Oxford: Pergamon Infotech. **

[Mill87a] Mills, H.D., M. Dyer, and R.C. Linger. "Cleanroom Software Engineering." *IEEE: Software*, 4/5 (Sep 1987):19-25.

[Mill87b] Millen, J.K., S.C. Clark, and S.B. Freedman. "The Interrogator: Protocol Security Analysis." *IEEE: Transactions on Software Engineering*, 13/2 (Feb 1987):274-285.

[Mins83] Minsky, N.H. 1983. "Locality in Software Systems." In *Conference Record of 10th Annual ACM Symposium on Principles of Programming Languages*, January, Austin, TX, 299-312. **

[Misr81] Misra, J., and K.M. Chandy. "Proof of Networks of Processes." *IEEE: Transactions on Software Engineering*, 7/4 (Jul 1981):417-426.

[Misr82] Misra, J., K.M. Chandy, and T. Smith. 1982. "Proving Safety and Liveness of Communicating Processes with Examples." In *Proceedings ACM Symposium on Principles of Distributed Computing*. **

[Misr83] Misra, P.N. 1983. "Software Reliability Analysis." *IBM: Systems Journal*, 22/3 (1983):262-270.

[Mitt82] Mittermeir, R.T. 1982. "Optimal Test Efforts for Software Systems." *Reliability in Electrical and Electronic Components and Systems*, E. Lauger and J. Moltof, eds., 650-654. Amsterdam: Elsevier North-Holland Publishing Co. **

[Miya85] Miyazaki, Y., and K. Mori. 1985. "COCOMO Evaluation and Tailoring." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 292-299. Washington, DC: IEEE Computer Society Press. **

[Miya87] Miyazaki, Y., and N. Murakami. 1987. "Software Metrics Using Deviation Value." In *Proceedings 9th International Conference on Software Engineering*, March 30 - Arpil 2, Monterey, CA, 83-91. Washington, DC: IEEE Computer Society Press.

[MiyaXX] Miyamoto, I. "Toward an Effective Software Reliability Evaluation."

[Mizu83] Mizumo, Y. "Software Quality Improvements." *IEEE: Computer;* 16/3 (Mar 1983):66-72.

[Moha76a] Mohanty, S.N., and M. Adamowicz. 1976. "Proposed Measures for the Evaluation of Software." In *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press. **

[Moha76b] Mohanty, S.N. June 1976. *Automatic Program Testing*. Ph.D. diss., Polytechnic Institute of New York. **

[Moha79] Mohanty, S.N. "Models and Measurement for Quality Assessment of Software." *ACM: Computing Surveys*, 11/3 (Sep 1979):251-257.

[Mohe82] Moher, T., and G.M. Schneider. "Methodology and Experimental Research in Software Engineering." *International Journal of Man-Machine Studies*, 16/1 (1982):65-87. **

[Moor88] Moore, J.S. 1988. *PITON: A Verified Assembly Level Language*. Computational Logic Inc. Technical Report CLI-22. **

[Mora72] Moranda, P.B., and J. Jelinski. 1972. *Final Report on Software Reliability Study,* McDonnell Douglas Astronautics Co. MDC Report 63921. **

[Mora75] Moranda, P.B. 1975. "Predictions of Software Reliability During Debugging." In *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, 327-332, Washington, DC, 327-332.

[Mora78a] Moranda, P.B. "Software Reliability Revisited." *IEEE: Computer*, 11/4 (Apr 1978):92-94.

[Mora78b] Moranda, P.B. July 1978. *Critique of: An Analysis of Computing Software Reliability Models by Schick and Wolverton*. Computer Repository. Report R78-B1. **

[Mora78c] Moranda, P.B. "Is Software Science Hard?" In Surveyors' Forum. *ACM: Computing Surveys*, 10/4 (Dec 1978):503-504.

[Mora80] Moranda P.B. 1980. "Error Detection Models for Application During Program Development." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 75-78.

[Mora85] Moran, M.L. 1985. *A Graphical Debugger for Concurrent Ada*. Ph.D. diss., George Washington University.

[More81] Morell, L.J., and R.G. Hamlet. July 1981. *Error Propagation and Elimination in Computer Programs*. University of Maryland. Technical Report 81-1065. **

[More84] Morell, L. 1984. *A Theory of Error-Based Testing*. Ph.D. thesis, University of Maryland. Technical Report TR-1395. **

[More87] Morell, L.J. 1987. "A Model for Assessing Code-Based Testing Techniques." In *Proceedings 5th Annual Pacific Northwest Software Quality Conference: Effective Software Practices*, October, Portland, OR, 309-326.

[More88] Morell, L.J. 1988. "Theoretical Insights into Fault-Based Testing." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 45-62. Washington, DC: IEEE Computer Society Press.

[Morg84] Morgan, E.T. December 1984. *RGA Users Manual*. University of California at Irvine. Technical Report 243. **

[Morg86] Morgan, E.T., and R.R. Razouk. 1985. "Computer-Aided Analysis of Concurrent Systems." In *Proceedings 6th International Workshop on Protocol Specification, Testing, and Verification*, no. V, M. Diaz, ed., June, Toulouse, France, 49-58. North-Holland: Elsevier Science Publishers.

[Morg87] Morgan, E.T., and R.R. Razouk. "Interactive State-Space Analysis of Concurrent Systems." *IEEE: Transactions on Software Engineering*, 13/10 (Oct 1987):1080-1091.

[Mori83] Moriconi, M. 1983. "PegaSys: An Environment for Displaying, Animating, and Reasoning About Graphical Descriptions of Systems." In *Proceedings Symposium on Software Validation*, H.-L. Hansen (ed.), September, Darmstadt. Amersterdam: North-Holland.

[Morr71] Morris, J.H. "Another Recursion Induction Principle." *ACM: Communications of the ACM*, 14/5 (1971):351-354.

[Morr77] Morris, J.H., and B. Wegbreit. 1977. "Program Verification by Subgoal Induction." In *Current Trends in Programming Methodology*, no. 2, Ch. 8. Englewood Cliffs, NJ: Prentice Hall.

[Motl76] Motley, R.W., and W.D. Brooks. November 1976. *Statistical Prediction of Programming Errors*. Griffiss Air Force Base, NY: Rome Air Development Center. **

[Muno88] Munoz, C.U. "An Approach to Software Product Testing." *IEEE: Transactions on Software Engineering*, 14/11 (Nov 1988):1589-1596.

[Muns89] Munson, J.C. and T.M. Khoshgoftaar. 1989 "The Dimensionality of Program Complexity." In *Proceedings 11th International Conference on Software Engineering*, May 15-18, Pitsburgh, PA, 245-253. Washington, DC: IEEE Computer Society Press.

[Mura89] Murata, T., B. Shenker, and S.M. Shatz. "Detection of Ada Static Deadlocks Using Petri Net Invariants." *IEEE: Transactions on Software Engineering*, 15/3 (May 1989):314-325.

[Musa75] Musa, J.D. "A Theory of Software Reliability and Its Application." *IEEE: Transactions on Software Engineering*, 1/1 (Mar 1975):312-327.

[Musa76] Musa, J.D. 1976. "An Exploratory Experiment with "Foreign" Debugging of Programs." *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York.

MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press.

[Musa77] Musa, J.D. 1977. "Measuring Software Reliability." In *Proceedings TIMS-ORSA Joint National Meeting*, May 9-11, San Francisco, CA. **

[Musa79a] Musa J.D. "Validity of Execution-Time Theory of Software Reliability." *IEEE: Transactions on Reliability*, R-28/3 (Aug 1979):181-191.

[Musa79b] Musa, J.D. 1979. "Software Reliability Measures Applied to System Engineering." In *Proceedings AFIPS National Computer Conference*, vol. 48, June 4-7, New York, NY, 941-946. Arlington, VA: AFIPS Press.

[Musa80a] Musa, J.D. "Software Reliability Measurement." *Journal of Systems and Software*, 1/3 (1980):223-241. **

[Musa80b] Musa, J.D. "The Measurement and Management of Software Reliability." In *Proceedings of the IEEE*, 68/9 (Sep 1980):1131-1141.

[Musa84] Musa, J.D., and K. Okumaoto. 1984. "A Logarithmic Poisson Execution Time Model for Software Reliability Measurements." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 230-238. Washington, DC: IEEE Computer Society Press.

[Musa87] Musa, J., A. Iannino, K. Okumoto. 1987. *Software Reliability: Measurement, Prediction, Application*. New York: McGraw Hill.

[Musa89] Musa, J.D, and R.U. Fujii. "Quantifying Software Validation: When to Stop Testing." *IEEE: Software*, 6/3 (May 1989):19-27.

[Muss79] Musser, D.R. 1979. "Abstract Data Type Specification in the AFFIRM System." In *Proceedings International Conference on Specification of Reliable Software*, 47-57. Also published in *IEEE: Transactions on Software Engineering*, 6/1 (Jan 1980):24-32.

[Myer76] Myers, G.J. 1976. *Software Reliability: Principles and Practices*. New York: John Wiley & Sons. **

[Myer77] Myers, G.J. "An Extension to the Cyclomatic Measure of Program Complexity." *ACM: SIGPLAN Notices*, 12/12 (Oct 1977):61-64.

[Myer78a] Myers, G.J. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections." *ACM: Communications of the ACM*, 21/9 (Sep 1978):760-768.

[Myer78b] Myers, G.J. "Software Reliability is Not an Equation." *IEEE: Computer*, 11/6 (Jun 1978):82-83.

[Myer79] Myers, G.J. 1979. *The Art of Software Testing*.

[Myer83] Myers, B.A. "Incense: A System for Displaying Data Structures." *Computer Graphics*, 17/3 (Jul 1983):115-125.

[Myer86] Myers, W. "Can Software for the Strategic Defense Initiative ever be Error-Free?" *IEEE: Computer*, 19/10 (Nov 1986):61-67. **

[Myhr68] Myhre, J.M., and S.C. Saunders. "Comparison of Two Methods of Obtaining Approximate Confidence Intervals for System Reliability." *Technometrics*, 10/2 (Feb 1968):37-49.

[NASA81] National Aeronautics and Space Administration. September 1981. "Software Engineering Laboratory (SEL) Data Base Organization and User's Guide." In *Software Engineering Laboratory Series*, Greenbelt, MD: NASA. Report SEL-81-002. **

[NBS74] *NBS FORTRAN Test Programs, Vols. 1-3*. Gaithersburg, MD: National Bureau of Standards. Special Publication 399, October 1974. **

[NBS82a] *Planning for Software Validation, Verification, and Testing*. Gaithersburg, MD: National Bureau of Standards. Special Publication 500-98, September 1982.

[NBS82b] *Software Validation, Verification, and Testing Technique and Tool Reference Guide*. Gaithersburg, MD: National Bureau of Standards. Special Publication 500-93, November 1982.

[Nage82] Nagel, P.M., and J.A. Skrivan. February 1982. *Software Reliability: Repetitive Run Experimentation and Modeling*. Seattle, WA: Boeing Computer Services. Technical Report BCS-40366. **

[Nage84] Nagel, P.M., F.W. Scholz, and J.A. Skrivan. June 1984. *Software Reliability: Additional Investigations Into Modeling with Replicated Experiments*. NASA Contractor Report 172378.

[Najm87] Najm, E. 1987. "A Verification Oriented Specification in Lotos of the Transport Protocol." In *Proceedings 7th IFIP Protocol Symposium*, May, Zurich, Switzerland, 181-203. **

[Naka89] Nakagawa, Y. and S. Hanata. 1989. "An Error Complexity Model for Software Reliability Measurement." In *Proceedings 11th International Conference on Software Engineering*, May 15-18, Pitsburgh, PA, 123-236. Washington, DC: IEEE Computer Society Press.

[Naur69] Naur, P. "Programming by Action Clusters." *BIT*, 9/3 (1969):250-258.

[Nels66] Nelson, E.A. October 1966. *Management Handbook for the Estimation of Computer Programming Costs*. System Development Corp. Report AD-A648750. **

[Nels73] Nelson, E.C. 1973. *A Statistical Basis for Software Reliability Assessment*. Redondo Beach, CA: TRW. TRW Software Series TRW-SS-73-03. **

[Nels78] Nelson, E.C. "Estimating Software Reliability from Test Data." *Microelectronics and Reliability*, Vol. 17 (1978):67-74.

[Neum75] Neumann, P.G., L. Robinson, K. Levitt, R.S. Boyer, and A.R. Saxema. 1975. *A Provably Secure Operating System*. Menlo Park, CA: SRI International. SRI Project 2581. **

[Ng78] Ng, P.H., and G. Young. "A 1900 Fortran Postmortem Dump System." *Software Practice and Experience*, 8/4 (Jul 1978):421-428.

[Nguy86] Nguyen, V., A. Demers, D. Gries, and S. Owicki. "A Model and Temporal Proof System for Networks of Processes." *Distributed Computing*, January 1986, 7-25. **

[Nico87] Nicola, V.F., V.G. Kulkarni, and K.S. Trivedi. "Queueing Analysis of Fault-Tolerant Computer Systems." *IEEE: Transactions on Software Engineering*, 13/3 (Mar 1987):363-375.

[Noon75] Noonan, R.E. "Structured Programming and Formal Specification." *IEEE: Transactions on Software Engineering*, 1/4 (Dec 1975).

[Ntaf79] Ntafos, S.C., and S.L. Hakimi. "On Path Cover Problems in Digraphs and Applications to Program Testing." *IEEE: Transactions on Software Engineering*, 5/5 (Sep 1979):520-529.

[Ntaf81a] Ntafos, S.C. 1981. *On Testing with Required Elements*. University of Texas at Dallas. Technical Report 90. Also published in *Proceedings 5th International Computer Software and Applications Conference*, November 18-20, Chicago, IL, 132-139. Los Alamitos, CA: IEEE Computer Society Press.

[Ntaf81b] Ntafos, S.C., and S.L. Hakimi. "On Structured Diagraphs and Program Testing." *IEEE: Transactions on Computing*, C-30/1 (Jan 1981):67-71.

[Ntaf82] Ntafos, S.C. November 1984. *On Required Element Testing*. University of Texas at Dallas. Technical Report No. 123. Also published in *IEEE: Transactions on Software Engineering*, 10/6 (Nov 1984):795-803.

[Ntaf84] Ntafos, S.C. 1984. "An Evaluation of Required Element Testing Strategies." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 250-256. Washington, DC: IEEE Computer Society Press. **

[Ntaf85] Ntafos, S.C. June 1985. *A Comparison of Some Structural Testing Strategies*. University of Texas. Technical Report No. 210. Also published in *IEEE: Transactions on Software Engineering*, 14/6 (Jun 1988):868-874.

[Offu87] Offutt, A.J., and K.N. King. 1987. "A Fortran 77 Interpreter for Mutation Analysis." In *Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, June, St. Paul, MN. Published as *ACM: SIGPLAN Notices*, 22/7 (Jul 1987):177-188.

[Ohba84] Ohba, M. "Software Reliability Analysis Models." *IBM: Journal of Research and Development*, 28/4 (Jul 1984):428-443.

[Ohba89] Ohba, M. and X. Chou. 1989. "Does Imperfect Debugging Affect Software Reliability Growth?" In *Proceedings 11th International Conference on Software Engineering*, May 15-18, Pitsburgh, PA, 237-244. Washington, DC: IEEE Computer Society Press.

[Okad82] Okada, M., and M. Azuma. 1982. "Software Development Estimation Study–A Model from CAD/CAM System Development Experiences." In *Proceedings 6th International Computer Software and Applications Conference*, March 9-12, San Diego, CA, 555-564. Los Angeles, CA: IEEE Computer Society.

[Olde77] Oldehoeft, R.R. "A Contrast Between Language Level Measures." A Short Note in *IEEE: Transactions on Software Engineering*, 3/6 (Nov 1977):476-478.

[Olde83] Oldehoeft, R.R. "Program Graphs and Execution Behavior." *IEEE: Transactions on Software Engineering*, 9/1 (Jan 1983):103-108.

[Olen86] Olender, K.M., and L.J. Osterweil. 1986. "Specification and Static Evaluation of Sequencing Constraints in Software." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 14-22. Washington, DC: IEEE Computer Society Press.

[Oste75a] Osterweil, L.J., and L.D. Fosdick. 1975. "DAVE–A FORTRAN Program Analysis Systems." In *4th Conference on Computer Science and Statistics: Proceeedings 8th Symposium on the Interface*, February, Los Angeles, CA, 329-335. Springer-Verlag. **

[Oste75b] Osterweil, L.J., and L.D. Fosdick. September 1975. *Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection*. University of Colorado. Technical Report 15. **

[Oste76a] Osterweil, L.J., and L.D. Fosdick. "DAVE – A Validation Error Detection and Documentation System for Fortran Programs." *Software Practice and Experience*, 6/4 (Oct-Dec 1976):473-486.

[Oste76b] Osterweil, L.J., and L.D. Fosdick. 1976. "Some Experience with DAVE–A Fortran Program Analyzer." In *Proceedings AFIPS National Computer Conference*, vol. 45, June 7-10, New York, NY, 909-915. Montvale, NJ: AFIPS Press.

[Oste77] Osterweil, L.J. 1977. "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis." In *Proceedings Ist International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 406-413. Long Beach, CA: IEEE Computer Society Press.

[Oste80] Osterweil, L.J. July 1980. *A Strategy for Effective Integration of Verification and Testing Techniques*. University of Colorado. Technical Report CU-CS-181-80.

[Oste81a] Osterweil, L.J. 1981. "Using Data Flow Tools in Software Engineering." In *Program Flow Analysis: Theory and Applications*, Muchnick and Jones (eds.). Englewood Cliffs, NJ: Prentice Hall. **

[Oste81b] Osterweil, L.J. 1981. "A Strategy for Integrating Program Testing and Analysis." In *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 187-229. North-Holland. **

[Oste83] Osterweil, L.J. "Toolpack – An Experimental Software Development Environment Research Project." *IEEE: Transactions on Software Engineering*, 9/6 (Nov 1983):673-685.

[Oste84] Osterweil, L.J. 1984. "Integrating the Testing, Analysis, and Debugging of Programs." In *Software Validation*, H.-L. Hausen (ed.), 73-102. North Holland.

[Oste86a] Osterweil, L.J. July 1986. *Notes on Object Management in Arcadia*. University of Colorado. Technical Report CU-86-04. **

[Oste86b] Osterweil, L.J. 1986. *Software Environment Architecture*. University of Colorado. Technical Report CU-CS-332-86. **

[Oste87] Osterweil, L.J. March 1987. "Software Processes are Software Too." In *Proceedings 9th International Conference on Software Engineering*, March 30 - Arpil 2, Monterey, CA, 2-13. Washington, DC: IEEE Computer Society Press.

[Ostr78] Ostrand, T.J., and E.J. Weyuker. 1978. "Remarks on the Theory of Test Data Selection." In *Digest IEEE Workshop on Software Testing and Test Documentation*, December 18-20, Ft. Lauderdale, FL, 1-18. IEEE Computer Society Technical Committee on Software Engineering. **

[Ostr79] Ostrand, T.J., and E.J. Weyuker. 1979. "Error-Based Program Testing." In *Proceedings 1979 Conference on Information Sciences and Systems*, March, Baltimore, MD, 444-449. **

[Ostr80] Ostrand, T.J., and E.J. Weyuker. 1980 "Current Directions in the Theory of Testing." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 386-389. Los Alamitos, CA: IEEE Computer Society Press.

[Ostr84] Ostrand, T.J., and E.J. Weyuker. "Collecting and Categorizing Software Error Data in an Industrial Environment." *Journal of Systems and Software*, 4/4 (1984):289-300.

[Ostr86] Ostrand, T.J., R. Sigal, and E.J. Weyuker. 1986. "Design for a Tool to Manage Specification-Based Testing." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 41-50. Washington, DC: IEEE Computer Society Press.

[Ostr88] Ostrand, T.J., and M.J. Balcer. "The Category-Partition Method for Specifying and Generating Functional Tests." *ACM: Communications of the ACM*, 31/6 (Jun 1988):676-686.

[Otte76] Ottenstein, K.J. 1976. *A Program to Count Operators and Operands for ANSI-FORTRAN Modules*. Purdue University. Technical Report CSU-TR-196. **

[Otte78] Ottenstein, L.M. August 1978. *Predicting Parameters of the Software Validation Effort*. Ph.D. diss., Purdue University. **

[Otte79] Ottenstein, L.M. "Quantitative Estimates of Debugging Requirements." *IEEE: Transactions on Software Engineering*, 5/5 (Sep 1979):504-514.

[Otte81] Ottenstein, L. 1981. "Predicting Numbers of Errors Using Software Science." In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March:157-167.

[Owic75] Owicki, S.S. 1975. *Axiomatic Proof Techniques for Parallel Programs*. Ph.D. diss., Cornell University. Also published in *Acta Informatica*, no. 6 (1976):319-340.

[Owic76] Owicki, S., and D. Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach." *ACM: Communications of the ACM*, 19/5 (May 1976):279-285.

[Owic82] Owicki, S.S., and L. Lamport. "Proving Liveness Properties of Concurrent Programs." *ACM: Transactions on Programming Languages and Systems*, 4/3 (Jul 1982):445-495.

[Oxma78] Oxman, S.W. 1978. "The Testing of the TRIDENT Command and Control System." In *Digest IEEE Workshop on Software Testing and Test Documentation*, December 18-20, Ft. Lauderdale, FL, 284-295. IEEE Computer Society Technical Committee on Software Engineering. **

[Page74] Page, M.P., and J.P. Benson. "The Use of Software Probes in Testing FORTRAN Programs." *IEEE: Computer*, Vol. 7 (1974):18-25. **

[Page82] Page, G.T., D.N. Card, and F.E. McGarry. September 1982. *Evaluation of Management Measure of Software Development*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-82-001, Vols. 1 and 2. **

[Page84] Page, G.T., F.E. McGarry, and D.N. Card. November 1985. "A Practical Experience with Independent Verification and Validation." In *Proceedings 8th International Computer Software and Applications Conference*. Washington, DC: IEEE Computer Society. **

[Page85] Page, G.T., F.E. McGarry, and D.N. Card. June 1985. *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-81-110. **

[Paig72] Paige, M.R., and E.F. Miller. 1972. "A Method for Ranking Priorities in Testing Computer Programs." In *Proceedings Computer Systems Design Conference*.

[Paig75] Paige, M.R. "Program Graphs, an Algebra, and Their Implication for Programming." *IEEE: Transactions on Software Engineering*, 1/3 (Sep 1975):286-291.

[Paig77a] Paige M.R. 1977. "Software Testing Principles and Practice Using a Testing Coverage Analyzer." In *Proceedings Software '77 Conference*, October. **

[Paig77b] Paige, M.R. "On Partitioning Program Graphs." *IEEE: Transactions on Software Engineering*, 3/6 (Nov 1977):386-393.

[Paig78a] Paige, M. 1978. "An Analytical Approach to Software Testing." In *Proceedings 2nd International Computer Software and Applications Conference*, November 13-16, Chicago, IL, 527-531. Long Beach, CA: IEEE Computer Society Press.

[Paig78b] Paige, M. 1978. "Software Design for Testability." In *Proceedings IEEE 11th Hawaii International Conference on System Sciences*, January, Honolulu, HA. **

[Paig81] Paige, M. 1981. "Data Space Testing." In *Proceedings ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March:117-127.

[Panz76] Panzl, D.J. 1976. "Test Procedures--A New Approach to Software Verification." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA, 477-485. Washington, DC: IEEE Computer Society Press.

[Panz78a] Panzl, D.J. "Automatic Software Test Drivers." *IEEE: Computer*, 11/4 (Apr 1978):44-50.

[Panz78b] Panzl, D.J. 1978. "A Language for Specifying Software Tests." In *Proceedings AFIPS National Computer Conference*, vol. 47, June 5-8, Anaheim, CA, 5-8. Arlington, VA: AFIPS Press.

[Panz78c] Panzl, D.J. 1978. "Automatic Revision of Formal Test Procedures." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 320-326. Washington, DC: IEEE

Computer Society Press.

[Panz81a] Panzl, D.J. 1981. "Experience with Automatic Program Testing." In *Proceedings NBS Trends and Applications*, May 28, 25-28. Gaithersburg, MD: National Bureau of Standards. **

[Panz81b] Panzl, D. "A Method for Evaluating Software Development Techniques." *ACM: SIGSOFT STME* (1981). **

[Pari76] Pariseau, R.J. November 1976. *A Screening Criteria for Delivered Source in Military Software*. Naval Air Development Center. Technical Report NADC-79163-50, Vol. I. **

[Parn72a] Parnas, D.L. "Some Conclusions from an Experiment in Software Engineering Techniques." In *Proceedings AFIPS Fall Joint Computer Conference*, vol. 41, December 5-7, Anaheim, CA, 325-329. Montvale, NJ: AFIPS Press.

[Parn72b] Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules." *ACM: Communications of the ACM*, 15/12 (Dec 1972):1053-1058.

[Parn72c] Parnas, D.L. "A Technique for Software Module Specification with Examples." *ACM: Communications of the ACM*, 15/5 (1972):330-336.

[Parn74] Parnas, D. "On a 'Buzzword': Hierarchical Structure." In *Proceedings Information Processing (IFIP) Congress '74*, August 5-10, Stockholm, Sweden. Amsterdam: North-Holland.

[Parn77] Parnas, D.L. 1977. "The Use of Precise Specifications in the Development of Software." In *Proceedings Information Processing (IFIP) Congress '77*, August 8-12, Toronto, Canada, 861-867. Amsterdam: North-Holland.

[Parn78] Parnas, D.L. 1978. "Designing Software for Ease of Extension and Contraction." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 264-277. Washington, DC: IEEE Computer Society Press.

[Parn79] Parnas, D.L. 1979. "The Role of Program Specification." In *Research Directions in Software Technology*, P. Wegner (ed.), 364-370. MIT Press.

[Parn85] Parnas, D.L. "Software Aspects of Strategic Defense Systems." *ACM: SIGSOFT Software Engineering Notes*, 10/5 (Oct 1985):15-23.

[Parn88] Parnas, D.L., A.J. van Schouwen, and S.P. Kwan. May 1988. *Evaluation Standards for Safety Critical Software*. Queens University. Technical Report 88-220.

[Parr80] Parr, F.N. "An Alternative to the Rayleigh Curve Model for Software Development Effort." *IEEE: Transactions on Software Engineering*, 6/5 (May 1980):291-296.

[Pase87a] Pase, B., M. Saaltink, and S. Kromodimoeljo. November 1987. *m-EVES User's manual*. I.P. Sharp Associates. Technical Report TR-87-5402-14. **

[Pase87b] Pase, B., and S. Kromodimoeljo. January 1987. *NEVER: An Interactive Theorem Prover*. I.P. Sharp Associates. Conference Paper CP-87-5402-20. **

[Pate89] Pate, S., R.A. Orr, and M.T. Norris. May 1989. "Tools to Support Formal Methods." In *Proceedings 11th International Conference on Software Engineering*, May 15-18, Pitsburgh, PA, 123-132. Washington, DC: IEEE Computer Society Press.

[Payt82] Payton, T., S. Keller, J. Perkins, S. Rowan, and S. Mardinly. 1982. "SSAGS: A Syntax and Semantics Analysis and Generation System." In *Proceedings 6th International Computer Software and Applications Conference*, March 9-12, San Diego, CA, 424-433. Los Angeles, CA: IEEE Computer Society.

[Pear84] Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison Wesley.

[Pere85] Perera, I.A., and L.J. White. 1985. *Selecting Test Data for the Domain Testing Strategy*. University of Alberta. Technical Report TR85-5. **

[Perk86] Perkins, J., D.M. Lease, and S.E. Keller. 1986. "Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada." In *Proceedings 4th Annual National Conference on Ada Technology*, March, Atlanta, GA, 67-74.

[Perk87] Perkins, J.A., and R.S. Gorzela. 1987. "Experience Using an Automated Metric Framework to Improve the Quality of Ada Software." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 277-284.

Washington, DC: ACM Ada Technical Committee.

[Perl81] Perlis, A.J., F.G. Sayward, and M. Shaw (eds.). 1981. *Software Metrics: An Analysis and Evaluation*. Cambridge, MA: MIT Press. **

[Perr83] Perry, W.E. 1983. *A Structured Approach to Systems Testing*. Wellesley, MA: QED Information Sciences, Inc.

[Perr86] Perry, W.E. 1986. *How to Test Software Packages*. New York: John Wiley & Sons.

[Perr87] Perry, S., et al. March 1987. *Product Assurance Policies and Procedures for Flight Dynamics Software Development*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-87-001. **

[Perr88] Perry, W.E. 1988. "A Structured Approach to Systems Testing." In Welllesley, MA: QED Information Sciences, Inc. **

[Pesc85] Pesch, H., H. Schaller, P. Schnupp, and A.P. Spirk. 1985. "Test Case Generation Using Prolog." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 252-[illegible] Washington, DC: IEEE Computer Society Press.

[Pete76] Peterson, R.J. 1976. "TESTER/1: An Abstract Model for the Automatic Synthesis of Program Test Case Specifications." In *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press. **

[Pete77] Peterson, J. "Petri Nets." *ACM: Computing Surveys*, 9/3 (Sep 1977):223-252.

[Pete81] Peterson, J. 1981. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall. **

[Pets85] Petschenik, N.H. "Practical Priorities in System Testing." *IEEE: Software*, 2/5 (Sep 1985):18-23.

[Piat80] Piatkowski, T.F. 1980. "Remarks on the Feasibility of Validating and Testing ADCCP Implementations." In *Proceedings Trends and Applications Symposium*. **

[Pica81] Picasso, G.O. December 1981. *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*. Greenbelt, MD: NASA/GSFC. Technical Report SEL-81-012. **

[Piku76] Pikul, R.A., and R.T. Wojcik. 1976. "Software Effectiveness: A Reliability Growth Approach." In *Proceedings Symposium on Computer Software Engineering*, April 20-22, Polytechnic Institute for New York. MRI Symposia Series, vol. XXIV, J. Fox (ed.). New York: Polytechnic Press.

[Pimo75] Pimont, S., and J.-C. Rault. "A Software Reliability Assessment Based on a Structural and Behavioral Analysis of Programs." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA, 486-491. Washington, DC: IEEE Computer Society Press.

[Pipp78] Pippenger, N. "Complexity Theory." *Scientific American* (Jun 78):114-124.

[Piwo82] Piwowarski, P. "A Nesting Level Complexity Measure." *ACM: SIGPLAN Notices*, 17/9 (Sep 1982):44-50.

[Ploe79] Ploedereder. E. 1979. "Pragmatic Techniques for Program Analysis and Verification." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 63-72. Washington, DC: IEEE Computer Society Press. **

[Pnue77] Pnuenuli, A. 1977. "The Temporal Logic of Programs." In *Proceedings of the 18th Annual Symposium on Foundations of Computer Sciences*, Oct 31 - Nov 2. **

[Pola81] Polak, W. 1981. "Compiler Specification and Verification." In *Lecture Notes in Computer Science*. Cimpiler Specification and Verification, 124. G. Goos and J. Hartmanis (eds.) New York: Springer-Verlag. **

[Pooc74] Pooch, U.W. "Translation of Decision Tables." *ACM: Computing Surveys*, 6/2 (Jun 1974):125-151.

[Pool73] Poole, P.C. 1973. "Debugging and Testing." In *Advanced Course on Software Engineering*, F.L. Bauer (ed.), 278-318. New York: Springer-Verlag. **

[Popk78] Popkin, G.S., and M.L. Shooman. November 1978. *On the Number of Tests Necessary to Verify a Computer Program*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-78-229

[Post87] Poston, R.M., and M.W. Bruen. "Counting Down to Zero Software Failures." *IEEE: Software*, 4/5 (Sep 1987):54-61.

[Pout87] Poutanen, O. 1987. "Two Portable Ada Testing Tools TBGEN and TCMON." In *Proceedings Ada Europe Conference*, May 26-28, Stockholm, Sweden, :197-208. New York: Cambridge University Press

[Prat80] Prather, R.C. 1980. "Model Evaluation Strategy." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 137-142.

[Prat83] Prather, R.E. "Theory of Program Testing." *Bell System Technical Journal*, 10/2/v62 (Dec 1983):3073-3105. **

[Prat87] Prather, R.E., and J.P. Myers, Jr. "The Path Prefix Software Testing Strategy." *IEEE: Transactions on Software Engineering*, 13/7 (Jul 1987):761-766.

[Press83] Presson, P.E., J. Tsai, T.P. Bowen, J.V. Post, and R.L. Schmidt. July 1983. *Software Interoperability and Reusability, Vols. I and II.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-83-174.

[Prin78] Principato, R.N. 1978. *A Formalization of the State Machine Specification Technique*. MIT Laboratory of Computer Science. Report MIT/hcs/TR-2-2. **

[Prob80] Probert, R.L. 1980. *New and Old Test Techniques: Grey Box Testing and Software Instrumentation*. University of Ottowa. Technical Report 80-13. **

[Prob82a] Probert, R.L. 1982. "Grey-Box (Design-Based) Testing Techniques." In *Proceedings 15th Hawaii International Conference on System Sciences*, January, Honolulu, Hawaii, 94-102. **

[Prob82b] Probert, R.L., and H. Ural. 1982. "Incremental Improvement of Specifications by Testing." In *Proceedings Workshop on Effectiveness of Testing and Proving Methods*, May, Avalon, CA, 37-49. **

[Prob82c] Probert, R.L. "Optimal Insertion of Software Probes in Well-Delimited Programs." *IEEE: Transactions on Software Engineering*, 8/1 (Jan 1982):34-42.

[Prob83] Probert, R.L., D.R. Skuce, and H. Ural. 1983. "Specification of Representative Test Cases Using Logic Programming." In *Proceedings IEEE 16th Hawaii International Conference on System Sciences*, January, Honolulu, HA, 190-196. **

[Prob84] Probert, R.L., and H. Ural. "High-Level Testing and Example-Directed Development of Software Specifications." *Journal of Systems and Software*, 4/4 (Nov 1984):317-325.

[Prot88] Protzel, P.W. 1988. "Automatically Generated Acceptance Test: A Software Reliability Experiment." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 196-203. Washington, DC: IEEE Computer Society Press.

[Pryc82] Prycker, M. de. "On the Development of a Measurement System for High Level Language Program Statistics." *IEEE: Transactions on Software Engineering*, C-31 (Sep 1982):883-891. **

[Purd72] Purdom, P. "A Sentence Generator for Testing Parsers." *BIT* 12/3 (Jul 1972):366-375.

[Putn77] Putnam, L.H., and R.W. Wolverton. 1977. "Quantitative Management: Software Cost Estimating." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL. Long Beach, CA: IEEE Computer Society Press. **

[Putn78] Putnam, L.H. "A General Empirical Solution to the Macro Software Sizing and Estimation Problem." *IEEE: Transactions on Software Engineering*, 4/4 (Jul 1978):301-316.

[Putn79] Putnam, L.H., and A. Fitzsimmons. "Estimating Software Costs." *Datamation*, (Sep 1979):189-198, continued in *Datamation*, (Oct 1979):171-178 and (Nov 1979):137-140.

[Putn82] Putnam, L.H. 1982. "The Real Economics of Software Development." In *The Economics of Information Processing*, R. Goldberg and H. Lorin. New York: John Wiley. **

[Quir85] Quirk, W.J., ed. 1985. *Verification and Validation of Real-Time Software*. New York: Springer-Verlag.

[RADC76a] Rome Air Development Center. August 1976. *Software Reliability Study.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-76-238.

[RADC76b] *JAVS: Jovial Automated Verification System*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-76-20, February 1976. **

[RADC86] Rome Air Development Center. 1986. *Ada Test and Verification System (ATVS)*. Griffiss Air Force Base, NY: Rome Air Development Center. RADC Contract F30602-86-C-0192. **

[Rabi77] Rabin, M.O. "Complexity of Computations." *ACM: Communications of the ACM*, 20/9 (Sep 1977):625-633.

[Rama73] Ramamoorthy, C.V., R.E. Meeker, and J. Turner. 1973. "Design and Construction of an Automated Software Evaluation System." In *Conference Record 1973 IEEE Symposium on Computer Software Reliability*, April 30 - May 2, New York, 28-37. **

[Rama74a] Ramamoorthy, C.V., and S.F. Ho. August 1974. *FORTRAN Automatic Code Evaluation System*. University of California at Berkeley. Technical Report ERL-M-466. **

[Rama74b] Ramamoorthy, C.V., R. Cheung, and K.H. Kim. March 1974. *Reliability and Integrity of Large Computer Programs*. University of California at Berkeley. Technical Report (ERL-M430). **

[Rama75a] Ramamoorthy, C.V., and S-B.F. Ho. "Testing Large Software with Automated Software Evaluation Systems." *IEEE: Transactions on Software Engineering*, 1/1 (Mar 1975):187-199.

[Rama75b] Ramamoorthy, C.V., K.H. Kim, and W.T. Chen. "Optimal Placement of Software Monitors Aiding Systematic Testing." **

[Rama76] Ramamoorthy, C.V., S.-B.F. Ho, and W.T. Chen. "On the Automated Generation of Program Test Data." *IEEE: Transactions on Software Engineering*, 2/4 (Dec 1976):293-300.

[Rama80] Ramamoorthy, C.V., and S.F. Ho. 1980. "Modeling of the Software Reliability Growth Process." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 161-169. Los Alamitos, CA: IEEE Computer Society Press. **

[Rama81] Ramamoorthy, C.V., Y.R. Mok, F.B. Bastani, G.H. Chin, and K.S. Suzuki. "Application of a Methodology for the Development and Validation of Reliable Process Control Software." *IEEE: Transactions on Software Engineering*, 7/6 (Nov 1981):537-555.

[Rama82] Ramamoorthy, C.V., and F.B. Bastani. "Software Reliability – Status and Perspectives." *IEEE: Transactions on Software Engineering*, 8/4 (Jul 1982):354-367.

[Rand75] Randell, B. "System Structure for Software Fault Tolerance." *IEEE: Transactions on Software Engineering*, 1/2 (Jun 1975):220-232.

[Rapp80] Rapps, S., and E.J. Weyuker. August 1980. *Data Flow Analysis Techniques for Program Test Data Selection*. New York University, Courant Institute of Mathematical Sciences. Technical Report 023. Published in *Proceedings 6th International Conference on Software Engineering*, September 13-16, Tokyo, Japan, 272-278. Washington, DC: IEEE Computer Society Press. Also published in *IEEE: Transactions on Software Engineering*, 11/4 (Apr 1985):367-375.

[Raul73] Rault, J.-C. 1973. "Extension of Hardware Fault Detection Models to the Verification of Software." In *Program Test Methods*, W.C. Hetzel (ed.), 255-262. Englewood Cliffs, NJ: Prentice-Hall. **

[Razo85] Razouk, R.R., and C.V. Phelps. 1985. "Performance Analysis Using Timed Petri Nets." In *Protocol Specification, Verification, and Testing, IV*, Y. Yemini, R. Strom, and S. Yemini (eds.), 561-576. Amersterdam: North-Holland. **

[Redd84a] Reddy, G.R. May 1984. *Application of Software Quality Metrics to a Relational Data Base System*. M.S. thesis. Virginia Polytechnic Institute. **

[Redd84b] Reddy, G. June 1984. *Analysis of a DataBase Management System Using Software Metrics*. M.S. thesis, Virginia Polytechnic. **

[Redw83] Redwine, S.T. Jr. "An Engineering Approach to Software Test Data Design." *IEEE: Transactions on Software Engineering*, 9/2 (Mar 1983):191-200.

[RegoXX] Rego, V., and A.P. Mathur. *Stochastic Models of a Program Unification Technique for Concurrent Enhancement*. Purdue University. **

[Reif75] Reifer, D.J. 1975. "Automated Aids for Reliable Software." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 131-142. IEEE Cat. No. 75CH0940-7CSR.

[Reif78] Reifer, D.J. September 1978. *Verification, Validation, and Certification: A Software Acquisition Guidebook*. Redondo Beach, CA: TRW Defense and Space Systems. Report TRW-SS-78-05. **

[Reif79a] Reifer, D.J. "Software Failure Modes and Effects Analysis." *IEEE: Transactions on Reliability*, R-28/3 (Aug 1979):247-249. Previously published in *Proceedings Industry/SAMSO Conference and Workshop on Mission Assurance*, April, Los Angeles, CA.

[Reif79b] Reifer, D.J., and S. Trattner. "A Glossary of Software Tools and Techniques." *IEEE: Computer*, 10/7 (Jul 1977):6-14.

[Reif79c] Reif, J.H. 1979. "Data Flow Analysis of Communicating Processes." In *Proceedings 6th ACM Annual Symposium on Principles of Programming Languages*, San Antonio, TX, 257-268. **

[Reit79] Reiter, R.W. Jr. December 1979. *Empirical Investigation of Computer Program Development Approaches and Computer Programming Metrics*. Ph.D. diss., University of Maryland. **

[Reyn86] Reynolds, R.G., and D. Roberts. 1986. "PARTIAL: A Tool to Support the Metrics Driven Design of Ada Programs." In *Proceedings 15th ACM Computer Science Conference*, February, 213-219. **

[Reyn87] Reynolds, R.G. "The Partial Metrics System: Modeling the Stepwise Refinement Process Using Partial Metrics." *ACM: Communications of the ACM*, 30/11 (Nov 1987):956-963.

[Reyn89] Reynolds, R.G. "The Partial Metrics System: A Tool to Support the Metrics-Driven Design of Psuedocode Programs." *Journal of Systems and Software*, no. 9 (1989):287-295.

[Rich76] Richards, P., and P. Chang. December 1976. *Localization of Variables: A Measure of Complexity*. GE Technical Information Series 76CIS07. **

[Rich78] Richardson, D.J., L.A. Clarke, and D.L. Bennett. July 1978. *SYMPLR, SYmbolic Multivariate Polynomial Linearization and Reduction*. University of Massachusetts. Technical Report 78-16. **

[Rich81a] Richardson, D.J., and L.A. Clarke. 1981. "A Partition Analysis Method to Increase Program Reliability." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 244-253. Washington, DC: IEEE Computer Society Press.

[Rich81b] Richardson, D.J. August 1981. *Examples of the Application of the Partition Analysis Method*. University of Massachusetts. Technical Report TN-48. **

[Rich81c] Richardson, D.J. September 1981. *A Partition Analysis Method to Demonstrate Program Reliability*. Ph.D. diss., University of Massachusetts. **

[Rich81d] Richardson, D.J. September 1981. *Specifications for Partition Analysis*. University of Massachusetts. Technical Report TR-81-35. **

[Rich81e] Rich, C., and R.C. Waters. June 1981. *Abstraction, Inspection, and Debugging in Programming*. MIT Artificial Intelligence Laboratory. Memo 634. **

[Rich82] Richardson, D.J., and L.A. Clarke. 1982. "On the Effectiveness of the Partition Analysis Method." In *Proceedings 6th International Computer Software and Applications Conference*, March 9-12, San Diego, CA, 529-538. Los Angeles, CA: IEEE Computer Society.

[Rich85a] Richardson, D.J., and L.A. Clarke. 1985. "Testing Techniques Based on Symbolic Evaluation." In *Software: Requirements, Specification, and Testing*, T. Anderson (ed.), 93-110. Blackwell Scientific Publications.

[Rich85b] Richardson, D.J., and L.A. Clarke. "Partition Analysis: A Method of Combining Testing and Verification." *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1477-1490.

[Rich86a] Richardson, D.J., and M.C. Thompson. December 1986. *An Analysis of Test Data Selection Criteria Using the RELAY Model of Error Detection*. University of Massachusetts. Technical Report 86-65. **

[Rich86b] Richardson, D.J., and M.C. Thompson. December 1986. *A New Model for Error Detection*. University of Massachusetts. COINS Technical Report 86-64. **

[Rich87a] Richardson, D.J., and M.C. Thompson. December 1987. *Testing Based on the RELAY Model of Error Detection*. University of Massachusetts. Technical Report 87-119.

[Rich87b] Richier, J.L., et al. "Verification in XESAR of the Sliding Window Protocol." In *Proceedings 7th IFIP Protocol Symposium*, May, Zurich, Switzerland, 235-248. **

[Rich88a] Richardson, D.J., and M.C. Thompson. 1988. "The RELAY Model of Error Detection and Its Application." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 223-230. Washington, DC: IEEE Computer Society Press.

[Ridd78] Riddle, W.E., J. Wileden, J. Sayler, A. Segal, and A. Stavely. "Behavior Modeling During Software Design." *IEEE: Transactions on Software Engineering*, 4/7 (Jul 1978):283-292.

[Ridd79] Riddle, W.E. "An Approach to Software System Behavior Modeling." *Comput. Lang.*, 4 (1979):29-47. **

[Ridd80] Riddell, I.J., M.A. Hennell, M.R. Woodward, and D. Hedley. 1980. *Practical Aspects of Program Mutation*. University of Liverpool. **

[Roac80] Roach, M.G. 1980. "A Comparison of Cost Estimation Techniques for Software Development Projects." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 229-235.

[Robi77] Robinson, L., and O. Roubine. January 1977. *SPECIAL - A Specification and Assertion Language*. Menlo Park, CA: SRI International. Technical Report CSL-46. **

[Robi79] Robinson, L., K.N. Levitt, and B.A. Silverburg. 1979. *The HDM Handbook*. SRI International. Project No. 4628. **

[Roby85] Roby, C.G., ed. December 1985. *Proceedings 1st IDA Workshop on Formal Specifications and Verification of Ada*. Alexandria, VA: Institute for Defense Analyses. IDA Memorandum Report M-146.

[Roe87] Roe, R.P., and J.H. Rowland. "Some Theory Concerning Certification of Mathematical Subroutines by Black Box Testing." *IEEE: Transactions on Software Engineering*, 13/6 (Jun 1987):677-682.

[Rogg80] Roggio R.F. 1980. *A Code-Based Model for Predicting Path Faults in COBOL Programs*. Ph.D. diss., Auburn University.

[Rola86] Rolandelli, C., T.J. Shimeall, C. Genung, and N. Leveson. February 1986. *Software Fault Tree Analysis Tool User's Manual*. University of California at Irvine. Technical Report 86-06 **

[Romb84] Rombach, H.D. 1984. "Software Design Metrics for Maintenance." In *Proceedings 9th Annual Software Engineering Workshop*, November 28, Greenbelt, MD. NASA/GSFC, 100-135.

[Romb85a] Rombach, H.D., and V.R. Basili. 1985. "A Methodology for Evaluating and Improving Life Cycle Support by Techniques." In *Proceedings 8th Minnowbrook Workshop on Software Performance Evaluation*, July 30 - August 2, Blue Mountain Lake, NY. **

[Romb85b] Rombach, H.D., and R.W. Selby Jr. 1985. "The Use and Interpretation of Characteristics Metrics Sets with Change, Error, and Fault Data." In *Proceedings NSIA National Joint Conference on Software Quality and Productivity*, Williamsburg, VA, March. **

[Romb87a] Rombach, H.D. 1987. "A Controlled Experiment on the Impact of Software Structure on Maintainability." *IEEE: Transactions on Software Engineering*, 12/3 (Mar 1987):344-354.

[Romb87b] Rombach, H.D., and V.R. Basili. 1987. "A Quantitative Assessment of Software Maintenance: An Industrial Case Study." In *Proceedings IEEE Conference on Software Maintenance*, September 21-24, Austin, TX, 134-144. **

[Romb87c] Rombach, H.D., and L. Mark. July 1987. *A Meta Information Base for Software Engineering*. University of Maryland. Technical Report TR-1765. **

[Romb88a] Rombach, H.D., and L. Mark. July 1988. *Software Process and Product Specifications: A Basis for Generating Customized Software Engineering Information Bases*. University of Maryland. Technical Report CS-TR-2062/UMIACS-TR-88-51. **

[Romb88b] Rombach, H.D., V.R. Basili, K. Reed, L. Mark, D. Stotts, and other members of the TAME project. October 1988. *TAME: Requirements and System Architecture*. University of Maryland. Technical Report TAME-TR-3-1988. **

[Romb88c] Rombach, H.D. 1988. *Software Specification: A Framework*. Carnegie-Mellon University. Published as CMU/SEI TR (curriculum module). **

[Romb88d] Rombach, H.D. 1988. "A Specification Language for Software Engineering Processes and Products." In *Proceedings 4th Software Process Workshop*, London, May 11-13. **

[Romb88e] Rombach, H.D. 1988. "What Data are Needed for Meaningful Reliability Assessment and Prediction?" In *Proceedings Annual National Joint Conference on Software Quality and Productivity*, Arlington, VA, March 1-3. **

[Romb88f] Rombach H.D. 1988 "The Evolution of Design Metrics Research: A Subjective View." In *Proceedings Design Metrics Workshop*, sponsored by the SPS and US **

[Romb88g] Rombach H.D., V.R. Basili, J. Bailey, and A. Delis. 1988. "Ada Reusability Analysis and Measurement." In *Proceedings 6th Symposium on Empirical Foundations of Information and Software Sciences*, Atlanta, GA, October 19-21. **

[Romb88h] Rombach H.D., V.R. Basili, J. Bailey, A. Delis, and F. Farhat. 1988. "Ada Reuse Metrics." In *Proceedings AIRMICS Workshop on Ada Reusability and Metrics*, Atlanta, GA, October 19-21. **

[Romb89a] Rombach, H.D., and T. Ulery. 1989. "Improving Software Maintenance Through Measurement." *IEEE Proceedings*, special issue on Software Maintenance, to be published in April 1989. **

[Romb89b] Rombach, H.D., and L. Mark. July 1989. *Generating Customized Software Engineering Information Bases from Software Process and Product Specifications*. University of Maryland. Technical Report CS-TR-2063. **

[Roqu86] Roquet, J.C., and P.J. Traverse. 1986. "Safe and Reliable Computing Onboard the Airbus and ATR Aircraft." In *Proceedings Safety of Computer Control Systems (SAFECOM) '86*, October, Sarlat, France, 93-97. **

[Rose75] Rosen, B. 1975. *Data Flow Analysis for Recursive PL/I Programs*. Yorktown Heights, NY: IBM T.J. Watson Research Center. Report RC5211. **

[Rose84] Rosenblum, D.S. "A Method of Designing Ada Tools using DIANA Trees as an Internal Form." In *Proceedings IEEE Computer Society Conference on Ada Applications and Environments*, October 15-18, St. Paul, MN, 63-70. Also published in *IEEE: Software*, 2/2 (Mar 1985):24-33.

[Rose85a] Rosenblum, D.S. "A Methodology for the Design of Ada Transformation Tools in a DIANA Environment." *IEEE: Transactions on Software Engineering*, 2/2 (Mar 1985):24-33.

[Rose85b] Rosenthal, L.S. January 1985. *Guidance on Planning and Implementing Computer System Reliability*. Gaithersburg, MD: National Bureau of Standards. NBS Special Publication 500-12.

[Ross88] Rosson, C.V. 1988. *Management Indicators: Assessing Product Reliability and Maintainability*. Virginia Polytechnic Institute. TR-88-40.

[Roub77] Roubine, O., and L. Robinson. 1977. *SPECIAL (SPECIfication and Assertion Language): Reference Manual*. Menlo Park, CA. SRI International. Technical Report TR-CSG-45. **

[Rowl81a] Rowland, J.H., and P.J. Davis. "On the Use of Transcendentals for Program Testing." *Journal of the ACM*, 28/1 (Jan 1981):181-190.

[Rowl81b] Rowland, J.H., and P.J. Davis. "On the Selection of Test Data for Recursive Mathematical Subroutines." *SIAM: Journal on Computing*, 10/1 (Feb 1981):59-72. **

[Rowl88] Rowland, J.H. 1988. "Artificial Systems for Software Engineering Studies." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 80-88. Washington, DC: IEEE Computer Society Press.

[Rube75] Rubey, R.J., J.A. Dana, and P.W. Biche. "Quantitative Aspects of Software Validation." *IEEE: Transactions on Software Engineering*, 1/2 (Jun 1975):150-155.

[Rubi82] Rubin, J., and C.H. West. "An Improved Protocol Validation Technique." *Computer Networks*, 6 (1982):65-73. **

[Rums77] Rumsey, J.R., and D.W. Abmayr. 1977. "An Effective Method for Measurement and Analysis of System Software Performance." In *Proceedings AFIPS National Computer Conference*, vol. 46, June 13-16, Dallas, TX, 523-527. Arlington, VA: AFIPS Press.

[Rush84] Rushby, J. 1984. *Mathematical Foundations of the MLS Tool for Revised Special*. Menlo Park, CA: SRI International. Draft Internal Note. **

[Russ83] Russinoff D.M. 1983. *An Experiment with the Boyer-Moore Program Verification System: A Proof of Wilson's Theorem*. M.S. thesis, University of Texas. **

[Rust71] Rustin, R. (ed.). 1971. *Debugging Techniques in Large Systems*. Englewood Cliffs, NJ: Prentice-Hall.

[SAMS77] *Failure Modes and Effects Analysis for Satellite, Launch Vehicle and Reentry Systems*. SAMSO-STD 77-2, November 1977. **

[SDIO87] Strategic Defense Initiative Organization. 30 June 1987. *Strategic Defense System Test and Evaluation Master Plan (TEMP)*.

[SDIO88a] Strategic Defense Initiative Organization. 16 November 1988. *Strategic Defense System Software Policy*.

[SDIO88b] Strategic Defense Initiative Organization. 16 November 1988. *Software Policy*. SDIO Management Directive No. 7.

[SEL82] *Annotated Bibliography of Software Engineering Laboratory (SEL) Literature*. Greenbelt, MD: NASA/GSFC. Report SEL-82-006. November 1982. **

[SERC87] Software Engineering Research Center. 1987. *The Mothra Testing Environment, User's Manual*. Purdue University. Technical Report SERC-TR-4-P.

[STAR85] *Guidebook for the STARS Measurement Program*, version 1. DoD STARS, September 1985. **

[STE86] Software Test and Evaluation Project. October 1986. *Software Test and Evaluation Manual, Vol. III, Good Examples of Software Testing in the Department of Defense*. Georgia Institute of Technology. Technical Report GIT-SERC-86/06. **

[SYSC83] *Avionics Software Support Cost Model*. SYSCON Corp. USAF Avionics Laboratory, Technical Report AFWAL-TR-1173, February 1983. **

[Sabn85] Sabnani, K.K., and A.T. Dahbura. 1985. "A New Technique for Generating Protocol Tests." In *Proceedings 9th Data Communications Symposium*. Published in *ACM: SIGCOM*, 15/4 (1985):36-43. **

[Sack68] Sackman, H., W.J. Erickson, and E.E. Grant. "Exploratory Experimental Studies Comparing On-Line and Off-Line Programming Performance." *ACM: Communications of the ACM*, 11/1 (Jan 1968):3-11.

[Sagl86] Saglietti, F., and W. Ehrenberger. 1986. "Software Diversity – Some Considerations about its Benefits and its Limitations." In *Proceedings Safety of Computer Control Systems (SAFECOM) '86*, October, Sarlat, France. **

[Sahn87] Sahner, R.A., and K.S. Trivedi. "Performance and Reliability Analysis Using Directed Acyclic Graphs." *IEEE: Transactions on Software Engineering*, 10/4 (Jul 1984):432-437.

[Salt82] Salt, N. "Defining Software Science Counting Strategies." *ACM: SIGPLAN Notices*, 17/3 (Mar 1982):58-67.

[Same76] Samet, H. 1977. "Compiler Testing via Symbolic Interpretation." In *Proceedings ACM 29th Annual National Computer Conference*, October 20-22, Houston, TX, 429-497. New York: Association for Computing Machinery.

[SanA83] San Antonio, R.C., and K.L. Jackson. 1983. "Application of Software Metrics During Early Program Phases." In *Proceedings National Security Industrial Conference of Software Test and Evaluation*, February, Washington, DC. **

[Sane83] Sanella, D., and M. Wirsing. 1983. "A Kernel Language for Algebraic Specification and Implementation." In *Proceedings International Conference on Foundations Computing Theory*, August, Bergholm, Sweden. **

[Sank85] Sankar, S., D.S. Rosenblum, and R. Neff. 1985. "An Implementation of Anna." In *Proceedings SIGAda International Conference*, May, Paris, France. Published in *ACM: Ada Letters*,V/2 (Sep-Oct 1985):285-296.

[Sank86] Sankar, S., and D.S. Rosenblum. 1986. *The Complete Transformation Methodology for Sequential Runtime Checking of an Anna Subset*. Stanford University. **

[Sari82] Sarikaya, B., and G.v. Bochmann. 1982. "Some Experience with Test Sequence Generation for Protocols." In *Proceedings 2nd Workshop on Protocols*, May, 555-567. **

[Sari84a] Sarikaya, B. March 1984. *Test Design for Computer Network Protocols*. Ph.D. thesis, McGill University. **

[Sari84b] Sarikaya, B., and G.v. Bochmann. "Synchronization and Specification Issues in Protocol Testing." *IEEE: Transactions on Communications*, COM-32/4 (Apr 1984):389-395.

[Sari87] Sarikaya, B., G.v. Bochmann, and E. Cerny. "A Test Design Methodology for Protocol Testing." *IEEE: Transactions on Software Engineering*, 13/5 (May 1987):518-531.

[Sari88a] Sarikaya, B. 1988. "Protocol Test Generation, Trace Analysis, and Verification Techniques." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 123-130. Washington, DC: IEEE Computer Society Press.

[Sari88b] Sarikaya, B., S. Eswara, V. Koukoulidis, and M. Barbeau. February 1988. *A Formal Specification Based Test Generation Tool*. Concordia University. Research Report. **

[Sari88c] Sarikaya, B., and G.v. Bochmann. March 1988. *Dynamic Analysis of Specifications in an Extended Finite-State Machine Model*. Available from author at Concordia University. **

[Sark89] Sarkar, D., and S.C. De Sarkar. "Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers." *IEEE: Transactions on Software Engineering*, 15/1 (Jan 1989).

[Satt72] Satterthwaite, E. "Debugging Tools for High-Level Languages." *Software Practice and Experience*, 2/3 (Jul 1972):197-217.

[Satt75] Satterthwaite, E.H. 1975. *Source Language Debugging Tools*. Ph.D. thesis, Stanford University. Technical Report STAN-CS-75-494. **

[Saud62] Sauder, R.L. 1962. "A General Test Data Generator for COBOL." In *Proceedings 1962 SJCC*, May, San Francisco, CA, 317-323. **

[Saxe77] Saxena, A.R. 1977. *Static Detection of Deadlock*. University of Colorado. Technical Report CU-CS-122-77. **

[Scha79] Schaffer, R.E., et al. June 1979. *Validation of Software Reliability Models*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-79-147.

[Sche85] Scheid, J., and S. Anderson. March 1985. *The Ina Jo[1] Specification Language Reference Manual* (draft). System Development Corp. Technical Report TM(L) 6021/001/01. **

[Schi73] Schick, G.J., and R.W. Wolverton. 1973. "Assessment of Software Reliability." In *Proceedings of Operations Research*, 395-422. Wurzburg-Wien: Physica-Verlag. **

[Schi78] Schick, G.J., and R.W. Wolverton. "An Analysis of Competing Software Reliability Models." *IEEE: Transactions on Software Engineering*, 4/2 (Feb 1978):104-120.

[Schi81] Schiffenbauer, R.D. September 1981. *Interactive Debugging in a Distributed Computational Environment*. Massachusetts Institute of Technology. Technical Report MIT/LCS/TR-264. **

[Schn75] Schneidewind, N.F. 1975. "Analysis of Error Processes in Computer Software." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 337-346. IEEE Cat. No. 75CH0940-7CSR.

[Schn76] Schneidewind, N.F., et al. November 1976. *Software Error Detection Models, Validation Tests and Program Complexity*. Naval Postgraduate School. Report NPS52SS76111. **

[Schn77a] Schneidewind, N.F., and H.M. Hoffman. 1977. "Software Structure and Error Properties: Models vs. Real Programs." In *Proceedings TIMS-ORSA Joint National Meeting*, May 9-11, San Francisco, CA. **

[Schn77b] Schneidewind, N.F. "The Use of Simulation in the Evaluation of Software." *IEEE: Computer*, 10/4 (Apr 1977):47-53.

[Schn77c] Schneidewind, N.F. "Modularity Considerations in Real Time Operating System Structures." In *Proceedings 1st International Computer Software and Applications Conference*, November 8-11, Chicago, IL, 397-403. Long Beach, CA: IEEE Computer Society Press.

[Schn78] Schneider, V. "Prediction of Software Effort and Project Duration: Four New Formulas." *ACM: SIGPLAN Notices*, 13 (Jun 1978):49-59. **

[Schn79a] Schneidewind, N.F., and H.-M. Hoffman. "An Experiment in Software Error Data Collection and Analysis." *IEEE: Transactions on Software Engineering*, 5/3 (May 1979):276-286.

[Schn79b] Schneidewind, N.F. 1979. "Software Metrics for Aiding Program Development and Debugging." In *Proceedings AFIPS National Computer Conference*, vol. 48, June 4-7, New York, NY, 989-994. Arlington, VA: AFIPS Press.

[Schr84] Schroeder, A. 1984. "Integrated Program Measurement and Documentation Tools." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 304-313. Washington, DC: IEEE Computer Society Press.

[Schu77] Schutts, D. 1977. "On a Hypergraph Oriented Measure for Applied Computer Science." In *Proceedings COMPCON Fall 1977*, Long beach, CA, 295-296. IEEE. **

1. Ina Jo is a trademark of the System Development Group of the Unisys Corp.

[Schu81] Schuman, S.A., E.M. Clarke, and C.N. Nikolaou. 1981. "Programming Distributed Applications in Ada: A First Approach." In *Proceedings 10th International Conference on Parallel Processing*, August, Ohio State University, OH, 38-49. Los Angeles, CA: IEEE Computer Society.

[Schw70a] Schwartz, J.T. 1970. "An Overview of Bugs." In *Debugging Techniques in Large Systems, 1st Courant Computer Science Symposium*,1-16. NYU Ed. Randell Rustin (ed.). Englewood Cliffs, NJ: Prentice-Hall.

[Scot83a] Scott, R.K. 1983. *Data Domain Modeling of Fault Tolerant Software Reliability*. Ph.D. diss., North Carolina State University. **

[Scot83b] Scott, R.K., et al. 1983. "Modeling Fault-Tolerant Software Reliability." In *Proceedings 3rd Symposium on Reliability Distrib. Software Database Systems*, Clearwater Beach, FL. **

[Scot84a] Scott, R.K., J.W. Gault, D.F. McAllister, and J.E. Wiggs. "Investigating Version Dependence in Fault-Tolerant Software." *AGARD Conference Proceedings 361*, 21.1-21.10.

[Scot84b] Scott, R.K., J.W. Gault, D.F. McAllister, and J.E. Wiggs. 1984. "Experimental Validation of Six Fault-Tolerant Software Reliability Models." In *Proceedings 14th Fault-Tolerant Computing Symposium*, 102-107.

[Scot87] Scott, R.K., J.W. Gault, and D.F. McAllister. "Fault-Tolerant Software Reliability Modeling." *IEEE: Transactions on Software Engineering*, 13/5 (May 1987):582-592.

[Sedl83] Sedlmeyer, R.L., W.B. Thompson, P.E. Johnson. 1983. "Knowledge-Based Fault Localization in Debugging." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):25-31. Baltimore, MD: ACM Order Department.

[Selb83] Selby, R.W. Jr. "An Empirical Study Comparing Software Testing Techniques." In *Proceedings 6th Minnowbrook Workshop on Software Performance Evaluation*, July 19-22, Blue Mountain Lake, NY. **

[Selb84] Selby, R.W. Jr. "Evaluating Software Testing Strategies." In *Proceedings 9th Annual Software Engineering Workshop*, November 28, Greenbelt, MD. NASA/GSFC. **

[Selb85] Selby, R.W. Jr. May 1985. *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*. Ph.D. diss, University of Maryland. Technical Report-1500.

[Selb86] Selby, R.W. Jr. 1986. "Combining Software Testing Strategies: An Empirical Evaluation." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 82-90. Washington, DC: IEEE Computer Society Press.

[Selb87a] Selby, R.W. Jr. 1987. "Incorporating Metrics into a Software Environment." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 326-331. Washington, DC: ACM Ada Technical Committee.

[Selb87b] Selby, R.W. Jr., V.R. Basili, and F.T. Baker. "CLEANROOM Software Development: An Empirical Evaluation." *IEEE: Transactions on Software Engineering*, 13/9 (Sep 1987):1027-1037.

[Selb87c] Selby, R.W. Jr. 1987. "Automatically Generating Software Metric Decision Trees for Identifying Error-Prone and Costly Modules." In *Proceedings 12th Annual Software Engineering Workshop*, Greenbelt, MD. NASA/GSFC. **

[Selb87d] Selby, R.W. Jr. 1987. "Analyzing Software Reuse at the Project and Module Design Levels." In *Proceedings 1st European Software Engineering Conference*. Strasbourg, France, September, 227-235. **

[Selb88a] Selby, R.W. Jr. 1988. "Generating Hierarchical System Descriptions for Software Error Localization." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 89-96. Washington, DC: IEEE Computer Society Press.

[Selb88b] Selby, R.W. Jr., and V.R. Basili. 1988. *Analyzing Error-Prone System Coupling and Cohesion*. University of California at Irvine. **

[Selb88c] Selby, R.W. Jr., and V.R. Basili. 1988. "Empirically Analyzing Software Reuse in a Production Environment." In *Software Reuse -- Emerging Technologies*. W. TRacz (ed.). New York: IEEE Computer Society. **

[Selb89] Selby, R.W. Jr., and A.A. Porter. "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis." To appear in *IEEE: Transactions on Software Engineering*. **

[Shan80] Shanthikumar J.G. 1980. "Software Performance Prediction Using a State-Dependent Error Occurrence-Rate Model." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 65-66.

[Shan81] Shanthikumar, J.G. 1981. "A State- and Time-Dependent Error Occurrence-Rate Software Reliability Model with Imperfect Debugging." In *Proceedings AFIPS National Computer Conference*, vol. 50, May 4-7, Chicago, IL, 311-315. Arlington, VA: AFIPS Press.

[Shan82] Shankar, K.S. "A Functional Approach to Module Verification." *IEEE: Transactions on Software Engineering*, 8/2 (Mar 1982):147-160.

[Shan87] Shankar, N. *Proof Checking Metamathematics: Volumes I and II.* Computational Logic Inc. Technical Report CLI-9. **

[Shap81] Shapiro, D.S. June 1981. *A System that Understands Bugs*. M.S. thesis, MIT Artificial Intelligence Laboratory. Memo 638. **

[Shat88] Shatz, S.M., and W. K. Cheng. "A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior." *Journal of Systems and Software*, (1988).

[Shaw78] Shaw, A.C. "Software Descriptions with Flow Expressions." *IEEE: Transactions on Software Engineering*, 4/3 (May 1978):242-254.

[Shaw80] Shaw, M. June 1980. "When is Good Enough." In *Software Metrics Panel Final Report*. ONR (AD A087 412). **

[Shaw89] Shaw, W.H., J.W. Howatt, R.S. Maness, D.M. Miller. "A Software Science Model of Compile Time." *IEEE: Transactions on Software Engineering*, 15/5 (May 1989):543-549.

[Shei81] Sheil, B.A. "The Psychological Study of Programming." *ACM: Computing Surveys*, 13/1 (Mar 1981):101-120.

[Shen80] Shen, V.Y., and H.E. Dunsmore. August 1980. *A Software Science Analysis of COBOL Programs*. Purdue University. Technical Report CSD-TR-348, Revised September 1981. **

[Shen83] Shen, V.Y., S.D. Conte, and H.E. Dunsmore. "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support." *IEEE: Transactions on Software Engineering*, 9/2 (Mar 1983):155-165.

[Shen85] Shen, V.Y., T.J. Yu, S.M. Thebaut, and L.R. Paulsen. "Identifying Error-Prone Software–An Empirical Study." *IEEE: Transactions on Software Engineering*, 11/4 (Apr 1985):317-324.

[Shep77] Sheppard, S.B., and L.T. Love. 1977. "A Preliminary Experiment to Test Influences on Human Understanding of Software." In *Proceedings 21st Annual Meeting of the Human Factors Society*, Vol. 21, 167-171. **

[Shep78] Sheppard, S.B., M.A. Borst, B. Curtis, and L.T. Love. 1978. *Factors Influencing the Understandability and Modifiability of Computer Programs*. Arlington, VA: General Electric Co.

[Shep79] Sheppard, S.B., B. Curtis, P. Milliman, and L.T. Love. "Modern Coding Practices and Programmer Performance." *IEEE: Computer* 12/12 (Dec 1979):41-49.

[Shep81] Sheppard, S.B., and E. Kruesi. 1981. *The Effects of the Symbology and Spatial Arrangement of Software Specifications on a Coding Task*. Arlington, VA: General Electric Co. Technical Report TR-81-388200-3. **

[Shim88] Shimeall, T.J., and N.G. Leveson. 1988. "An Empirical Comparison of Software Fault Tolerance and Fault Elimination." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 180-187. Washington, DC: IEEE Computer Society Press.

[Shne75] Shneiderman, B. 1975. "Experimental Testing in Programming Languages, Stylistic Considerations and Design Techniques." In *Proceedings AFIPS National Computer Conference*, vol. 44, May 19-22, Anaheim, CA, 653-656. Montvale, NJ: AFIPS Press.

[Shne77a] Shneiderman, B., R.E. Mayer, D. McKay, and P. Heller. "Experimental Investigations of the Utility of Detailed Flowcharts in Programming." *ACM: Communications of the ACM*, 20/6 (1977):373-381.

[Shne77b] Shneiderman, R. "Measuring Computer Program Quality and Comprehension." *International Journal of Man-Machine Studies*, 9 (1977):465-478. **

[Shne77c] Shneiderman, R. "Human Factors Experiments for Developing Quality Software." In *The Infotech State of the Art Report on Software Engineering*, Berkshire, England: Infotech International Ltd. **

[Shne80] Shneiderman, B. 1980. *Software Psychology: Human Factors in Computer and Information Systems*. Cambridge, MA: Winthrop Publishers. **

[Shol75] Sholl, H.A., and T.L. Booth. "Software Performance Modeling Using Computational Structures." *IEEE: Transactions on Software Engineering*, 1/4 (Dec 1975).

[Shoo72] Shooman, M.L. 1972. "Probabilistic Models for Software Reliability Prediction." In *Statistical Computer Performance Evaluation*, W. Freiderberg, ed., 485-502. New York: Academic Press.

[Shoo73] Shooman, M.L. 1973. "Operational Testing and Software Reliability Estimation During Program Development." In *Conference Record 1973 IEEE Symposium on Computer Software Reliability*, April 30 - May 2, New York, 51-57. **

[Shoo74] Shooman, M.L. January 1974. *Meaning of Exhaustive Software Testing*. Polytechnic Institute of New York. Report PINY EE/IP 74-006/EER/106. **

[Shoo75] Shooman, M.L., and M.I. Bolsky. 1975. "Types, Distribution, and Test and Correction Times for Programming Errors." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA, 347-357. IEEE Cat. No. 75CH0940-7CSR.

[Shoo76] Shooman, M.L. 1976. "Structural Models for Software Reliability Prediction." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA. Washington, DC: IEEE Computer Society Press.

[Shoo77a] Shooman, M.L. 1977. "The Spectre of Software Reliability and Its Exorcism." In *Proceedings Joint Automatic Control Conference*, 225-231. New York: IEEE Computer Society Press.

[Shoo77b] Shooman, M.L., and A. Laemmel. 1977. "Statistical Theory of Computer Programs in Information Content and Complexity." In *Proceedings COMPCON Fall 1977*, Long beach, CA, 341-347. IEEE. **

[Shoo77c] Shooman, M.L. 1977. "The Role of Reliability Analysis and Measurement." In *Proceedings Joint Automatic Control Conference*, June, 466-471. **

[Shoo79] Shooman, M.L., and H. Ruston. July 1979. *Summary of Technical Progress, Investigation of Software Models*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR79-188.

[Shoo83] Shooman, M.L. 1983. *Software Engineering: Design, Reliability, and Management*. New York: McGraw Hill. **

[Shoo86] Shooman, M.L. 1986. *Probabilistic Reliability: An Engineering Approach*. New York: McGraw-Hill, 1968. Updated and reprinted, Malabar, FL: Krieger, 1986. **

[Sidh89] Sidhu, D.P., and T.-K. Leung. "Formal Methods for Protocol Testing: A Detailed Study." *IEEE: Transactions on Software Engineering*, 15/4 (Apr 1989):413-426.

[Sief88] Siefert, P.T. and T.A. Babst. May 1988. *Automated Measurement System (AMS)*. Griffiss Air Force Base, NY: Rome Air Development Center. RADC Contract F30602-88-101.

[Sika88] Sikaczowski, R.O. January 1988. "Measuring Customer Satisfaction and Software Productivity through Quality Metrics." In *Quality Data Processing*, 37-44. **

[Silv79] Silverburg, B.A., L. Robinson, and K.N. Levitt. June 1979. *The Languages and Tools of HDM*. Stanford University, Research Project 4828. **

[Sing86] Singh, R., and N.F. Schneidewind. 1986. "Concept of a Software Quality Metrics Standard." In *Digest of Papers, Spring COMPCON 86*, A.G. Bell (ed.), March 3-6, 362-368. IEEE Computer Society. **

[Sist88] Sistla, A.P., and E.M. Clarke. "The Complexity of Propositional Linear Temporal Logic." *ACM: Journal of the ACM*, (1988). **

[Site74] Sites, S.L. May 1974. *Proving that Programs Terminate Cleanly*. Stanford University. Technical Report STAN-CS-74-418. **

[Siya80] Siyan, K.S. December 1980. *The Specification, Design and Implementation of an Input/Output Assertion Verifier*. M.S. Report, University of California at Berkeley. **

[Skil89] Skillicorn, D.B., and J.L. Glasgow. "Real-Time Specification Using Lucid." *IEEE: Transactions on Software Engineering*, 15/2 (Feb 1989).

[Slav75] Slavinski, R.T. November 1975. *Static FORTRAN Analyzer*. Griffiss Air Force Base, NY: Rome Air Development Center. **

[Sliv84] Slivinski, T., et al. 1984. *Study of Fault Tolerant Software Technology*. Mandex Inc., NASA Langley Research Center Report. **

[Smit79] Smith, C.P. June 1979. *Practical Applications of Software Science*. IBM Santa Teresa Laboratories. Technical Report 03.067. **

[Smit80a] Smith, C.P. 1980. *A Software Science Analysis of IBM Programming Products*. Santa Theresa, CA: IBM Corp. Technical Report TR 03.081. **

[Smit80b] Smith, C.P. 1980. "A Software Science Analysis of Programming Size." In *Proceedings ACM National Computer Conference*, October, 179-185. **

[Smit88] Smith, M.K., D. Craigen, and M. Saaltink. May 1988. *The nanoAVA Definition*. Computational Logic Inc. Technical Report CLI-21 (draft). **

[Snee78] Sneed, H., and K. Kirchoff. 1978. "Prufstand–A Testbed for Systematic Software Components." In *Proceedings INFOTECH State of the Art Conference on Software Testing*, London. Infotech. **

[Snee84] Sneed, H.M. "Software Renewal – A Case Study." *IEEE: Software*, 11/3 (Jul 1984):56-64.

[Snee85] Sneed, H., and A. Merey. "Automated Software Quality Assurance." *IEEE: Transactions on Software Engineering*, 11/9 (Sep 1985):909-916.

[Snee86] Sneed, H.M. 1986. "Data Coverage Measurement in Program Testing." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 34-40. Washington, DC: IEEE Computer Society Press.

[Sold89] Solderitsch, J.J., K.C. Wallnau, and J.A. Thalhamer. 1989. "Constructing Domain-Specific Ada Reuse Libraries." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 419-433. Washington, DC: ACM Ada Technical Committee. **

[Soli83] Solis, D.M., R.A. Kemmerer, and S. Eckmann. December 1983. *UNISEX Pascal Language Reference Manual*. University of California at Santa Barbara. Revised April 1985. **

[Soli85] Solis, D.M. 1985. "AutoParts–A Tool to Aid in Equivalence Partition Testing." In *Proceedings Soft-Fair II: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, December 2-5, San Francisco, CA, 122-125. **

[Solo83] Soloway, E. 1983. "You Can Observe a Lot by Just Watching How Designers Design." In *Proceedings 8th Annual Software Engineering Workshop*, November, Greenbelt, MD. NASA/GSFC. **

[Solo84] Soloway, E., and K. Ehrlich. "Empirical Studies of Programming Knowledge." *IEEE: Transactions on Software Engineering*, 10/5 (Sep 1984):595-609.

[Sone80] Soneriu, M.D., and D.S. Suk. 1980. "Markov Model for Estimating the Reliability of Duplicated and Repairable Computing Systems." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 87-96.

[Sone81] Soneriu, M.D. 1981. *A Methodology for the Design and Analysis of Fault-Tolerant Operating Systems*. Ph.D. diss., Illinois Intstitute of Technology.

[Soon77] Soong, N.L. "A Program Stability Measure." In *Proceedings 30th ACM Annual National Computer Conference*, October 16-19, Seattle, WA, 163-173. New York: Association for Computing Machinery.

[Sopp86] Soppe, M. October 1986. *A Tool for User Guided Test Suite Derivation from Formal Specifications*. M.S. thesis, Twente University. **

[Sork79] Sorkowitz, A.R. "Certification Testing: A Procedure to Improve the Quality of Software Testing." *IEEE: Computer*, 12/8 (Aug 1979):20-24.

[Srin85] Srinia, V.P. "A Fault-Tolerant Dataflow System." *IEEE: Computer*, 18/3 (Mar 1985):54-68.

[StJe85] St. Jean, L.D. 1985. *Testing Version Independence in Multi-Version Programming*. M.S. thesis, University of Virginia. **

[Stan77] Stanfield, J.R., and A.M. Skrukrud. November 1977. *Software Acquisition Management Guidebook -- Software Maintenance Volume*. System Development Corp. Technical Report TM-5772/004/02. **

[Stan80] Stankovic, J.A. 1980. "Debugging Commands for a Distributed Processing System." In *Proceedings COMPCON Fall 1980*, 701-705. **

[Stan83] Standish, T.A. "Interactive Ada in the Arcturus Environment. *ACM: Ada Letters*, III/1 (1983):23-35.

[Stan84a] Standish, T.A., and R.N. Taylor. "Arcturus: a Prototype Advanced Ada Programming Environment." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Published in *SIGPLAN Notices*, 19/5 (Aug 1984):57-64.

[Stan84b] Stanat, D.F., T.A. Thomas, and J.R. Dunham. 1984. *Proceedings of a Formal Verification/Design Proof Peer Review*. Research Triangle Institute. Technical Report RTI/2094/13-01F. **

[Stee86] Steenberger, C. August 1986. *Conformance Testing of OSI Systems*. M.S. thesis. Twente University. **

[Step74] Stepczyk, F.M. June 1974. "Requirements for Secure Operating Systems." In *TRW Software Series*. Report TRW-SS-74-05. **

[Stet86] Stetter, F. "Comments on "Number of Faults per Line of Code"." *IEEE: Transactions on Software Engineering*, 12/12 (Dec 1986):1145.

[Stev74] Stevens, W.P., G.J. Myers, and L.L. Constantine. 1974. "Structured Design." *IBM: Systems Journal*, 13/2 (1974):115-139.

[Stic78] Stickney, M.E. "An Application of Graph Theory to Software Test Data Selection." *ACM: Software Engineering Notes*, 3/5 (Nov 1978):72-78. **

[Stig74] Stigall, P.D., and O. Tasar. "Special Tutorial: A Review of Directed Graphs as Applied to Computers." *IEEE: Computer*, 7/10 (Oct 1974):39-47.

[Stol84] Stolzy, J.L. December 1984. *Software Fault Tree Analysis Tool*. University of California at Irvine. **

[Stoy77] Stoy, J. 1977. *Denotational Semantics: The Scott-Strachey Approach to Language Theory*. Boston, NJ: MIT Press. **

[Stuc72] Stucki, L.G. 1972. "A Prototype Automatic Program Testing Tool." In *Proceedings AFIPS Fall Joint Computer Conference*, vol. 41, December 5-7, Anaheim, CA, 829-836. Montvale, NJ: AFIPS Press.

[Stuc73] Stucki, L.G. 1973. "Automatic Generation of Self-Metric Software." In *Conference Record 1973 IEEE Symposium on Computer Software Reliability*, April 30 - May 2, New York, 94-100. **

[Stuc74] Stucki, L.G., and N.P. Svegel. January 1974. *Software Automated Verification System Study*. Huntington Beach, CA: McDonnell Dougals Astronautics Corp. Technical Report AD-784086. **

[Stuc75a] Stucki, L.G., and Foshee. 1975. "New Assertion Concepts for Self-Metric Software Validation." In *Proceedings International Conference on Reliable Software*, April 21-23, Los Angeles, CA. IEEE Cat. No. 75CH0940-7CSR. Published in *ACM: SIGPLAN Notices*, 10/6 (Jun 1975):59-71.

[Stuc75b] Stucki, L.G. 1975. "Testing Impact on the Future of Software Engineering." In *Proceedings 4th Annual Texas Conference on Computing Systems*, November, Austin, TX. **

[Stuc77] Stucki, L.G. 1977. "New Directions in Automated Tools for Improving Software Quality." In *Current Trends in Programming Methodology, Vol. II: Programming Validation*, 80-111. Englewood Cliffs, NJ: Prentice-Hall. Also in *Tutorial: Software Testing and Validation Techniques*, 2nd Edition, E. Miller and W.E. Howden (eds.). Los Alamitos, CA: IEEE Computer Society Press.

[Suke77a] Sukert, A.N. 1977. "An Investigation of Software Reliability Models." In *Proceedings Annual Reliability and Maintainability Symposium*.

[Suke77b] Sukert, A.N. 1977. "A Multi-Project Comparison of Software Reliability Models." In *Proceedings AIAA Conference on Computers in Aerospace: Exploration of the Outer Solar System*, vol. 50, November, Los Angeles, CA, 413-455. New York: American Institute of Aeronautics and Astronautics. **

[Suke79] Sukert, A.N. "Empirical Validation of Three Software Error Prediction Models." *IEEE: Transactions on Reliability*, R-28/3 (Aug 1979):199-205.

[Sull75] Sullivan, J.E. January 1975. *Measuring the Complexity of Computer Software*. Bedford, MA: Mitre Corp. Technical Report MTR-2648, Vol. V. **

[Suno82] Sunohara, T., A. Takano, K. Uehara, and T. Ohkawa. 1981. "Program Complexity Measure for Software Development Management." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 100-106. Washington, DC: IEEE Computer Society Press.

[Suns77] Sunshine, C.A., et al. "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models." *IEEE: Transactions on Software Engineering*, 3/3 (May 1977).

[Suns82] Sunshine, C.A., et al. "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models." *IEEE: Transactions on Software Engineering*, 8/5 (Sep 1982):460-489.

[Svob76] Svobodova, L. "Computer System Measurability." *IEEE: Computer*, 9/6 (Jun 1976):9-17. **

[Symo88] Symons, C.R. "Function Point Analysis: Difficulties and Improvements." *IEEE: Transactions on Software Engineering*, 14/1 (Jan 1988):2-11.

[Szul80] Szulewski, P.A., M.H. Whitworth, P. Buchan, and J.B. DeWolf. May 1980. *Quality Assurance Guidelines and Quality Metrics for Embedded Real-Time Software Designs*. Cambridge, MA: The Charles Stark Draper Laboratory, Inc. NBS Contract NB76SBCA0220. **

[Szul81] Szulewski, P.A., M.H. Whitworth, P. Buchan, and J.B. DeWolf. "The Measurement of Software Science Parameters in Software Design." *ACM: SIGMETRICS Performance Evaluation Review*, 10/1 (Spring 1981). **

[Szul83] Szulewski, P.A., N.M. Sodano, A.J. Rosner, and J.B. DeWolf. September 1983. *Automating Software Design Metrics*. Charles Stark Draper Laboratory. Technical Report CSDL-R-1662. Also published as Rome Air Development Center, Technical Report RADC-TR-84-27, February 1984. **

[Szul84] Szulewski, P.A., and N.M. Sodano. 1984. "Design Metrics and Ada." In *Washington Ada Symposium*, March 25-27, Laurel, MD, 105-114. Washington, DC: ACM Ada Technical Committee.

[Tai79] Tai, K.C. 1979. "On Program Testing Criteria." In *Proceedings 3rd International Computer Software and Applications Conference*, November 6-8, Chicago, IL, 695-701. Long Beach, CA: IEEE Computer Society Press. **

[Tai80] Tai, K.C. "Program Testing Complexity and Test Criteria." *IEEE: Transactions on Software Engineering*, 6/6 (Nov 1980):531-538.

[Tai85a] Tai, K.C., and C.Y. Din. 1985. "Validation of Concurrency in Software Specification and Design." In *Proceedings 3rd International Workshop on Software Specification and Design*, August, 223-227.

[Tai85b] Tai, K.C. 1985. "Reproducible Testing of Concurrent Ada Programs." In *Proceedings 2nd Conference on Software Development Tools, Techniques, and Alternatives*, December, San Francisco, CA, 114-121. **

[Tai85c] Tai, K.C. 1985. "On Testing Concurrent Programs." In *Proceedings 9th International Computer Software and Applications Conference*, October 9-11, 310-317. Los Angeles, CA: IEEE Computer Society.

[Tai86] Tai, K.C., and R.H. Carver. 1986. "Reproducible Testing of Concurrent Programs Based on Shared Variables." In *Proceedings IEEE 6th International Conference on Distributed Computing Systems*, May.

[Taka89] Takahashi, M., and Y. Kamayachi. 1989. "An Empirical Study of a Model for Program Error Prediction." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 330-336. Washington, DC: IEEE Computer Society Press. Also published in *IEEE: Transactions on Software Engineering*, 15/1 (Jan 1989):82-86.

[Tana81] Tanaka, A. 1981. *Equivalence Testing for the Fortran Mutation System Using Data Flow Analysis*. M.S. thesis, Georgia Institute of Technology. **

[Taus81] Tausworthe, R.C. 1981. *Deep Space Network Software Cost Estimation*. Pasadena, CA: Jet Propulsion Laboratory. **

[Taus82] Tausworthe, R.C. 1982. "Staffing Implications of Software Productivity Models." In *Proceedings 7th Annual Software Engineering Workshop*, Greenbelt, MD. NASA/GSFC. **

[Taus87a] Tauson-Conte, H.J., J.P. Salvador, C.A. Finnell, G. Baratta-Perez, and D.R. Clarson. March 1987. *Extending McCabe's Cyclomatic Complexity Metric for Analysis of Ada Software*. McCabe Associates. Technical Report MC87-McCabe II-0003. **

[Taus87b] Tauson-Conte, H.J., G. Baratta-Perez, C.A. Finnell, and D.R. Clarson. April 1987. *Modified A-Level Software Design Specification for the Ada Complexity Analysis Tool which Automates the ACE Metric*. McCabe Associates. Technical Report MC87-McCabe II-0005. **

[Taus88] Tauson-Conte, H.J. 1988. "Ada Complexity Extension (ACE) An Extension of McCabe's Cyclomatic Complexity Metric for Analysis of Ada Software." In *Proceedings 6th National*

Conference on Ada Technology, March 14-17, Arlington, VA, 7-12. Washington, DC: ACM Ada Technical Committee.

[Tayl77a] Taylor, D.J. 1977. *Robust Data Structure Implementation for Software Reliability*. Ph.D. thesis, University of Waterloo. **

[Tayl77b] Taylor, J.R., and S. Bologna. 1977. "Validation of Safety Related Software." In *Proceedings IAEA/NPPCI Specialists' Meeting on Software Reliability*, July, Pittsburgh, PA. **

[Tayl78a] Taylor, D.J., D.E. Morgan, and J.P. Black. 1978. *Theoretical Foundations for Robust Data Structure Implementations*. University of Waterloo. Technical Report CS-78-52. **

[Tayl78b] Taylor, R.N., and L.J. Osterweil. 1978. "A Facility for Verification, Testing, and Documentation of Concurrent Process Software." In *Proceedings 2nd International Computer Software and Applications Conference*, November 13-16, Chicago, IL, 36-41. Long Beach, CA: IEEE Computer Society Press.

[Tayl80a] Taylor, D.J., D.E. Morgan, and J.P. Black. "Redundancy in Data Structures: Some Theoretical Results." *IEEE: Transactions on Software Engineering*, 6/6 (Nov 1980):585-594.

[Tayl80b] Taylor, R.N., and L.J. Osterweil. "Anomaly Detection in Concurrent Software by Static Data Flow Analysis." *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):265-278.

[Tayl80c] Taylor, R.N., 1980. *Static Analysis of the Synchronization Structure of Concurrent Programs*. Ph.D. thesis, University of Colorado. **

[Tayl81] Taylor, R.N. May 1981. *An Algorithm for Analyzing Concurrent Programs*. University of Victoria. Technical Report DCS-10-IR. **

[Tayl82a] Taylor, T., and T.A. Standish. "Initial Thoughts on Rapid Prototyping Techniques." *ACM: Software Engineering Notes*, 7/5 (Dec 1982):160-166.

[Tayl82b] Taylor, R.N. November 1982. *Debugging Real-Time Software in a Host-Target Environment*. University of California at Irvine. Technical Report 212. Also published in *Technique et Science Informatiques*, 3/4 (1984):281-288, and in *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 194-201. Washington, DC: IEEE Computer Society Press.

[Tayl82c] Taylor, R.N. 1982. *An Integrated Verification and Testing Environment*. University of Victoria. Technical Report DCS-15-IR. **

[Tayl83a] Taylor, R.N. "A General-Purpose Algorithm for Analyzing Concurrent Programs." *ACM: Communications of the ACM*, 6/5 (May 1983):362-376.

[Tayl83b] Taylor, R.N. *Complexity of Analyzing the Synchronization Structure of Concurrent Programs*. University of Victoria. Technical Report DCS-9-IR. Also published in *Acta Informatica*, 19/1 (Apr 1983):57-84.

[Tayl83c] Taylor, R.N. 1983. "Analysis of Concurrent Software by Cooperative Application of Static and Dynamic Techniques." In *Proceedings Symposium on Software Validation*, H.-L. Hansen (ed.), September, Darmstadt. Amersterdam: North-Holland. Also published in *Software Validation*, H.-L. Hausen (ed.), 127-137. Amsterdam: North Holland.

[Tayl84] Taylor, R.N., and L.J. Osterweil. 1984. "Analysis and Testing Based on Sequential Specifications." In *Proceedings 4th Jerusalem Conference on Information Technology*, May. **

[Tayl85] Taylor, R.N., and T.A. Standish. "Steps to an Advanced Ada Programming Environment." *IEEE: Transactions on Software Engineering*, 11/3 (Mar 1985):302-310.

[Tayl86a] Taylor, R.N., and C.D. Kelly. 1986. "Structural Testing of Concurrent Programs." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 164-169. Washington, DC: IEEE Computer Society Press.

[Tayl86b] Taylor, R.N., L. Clarke, L.J. Osterweil, J.C. Wileden, and M. Young. 1986. "Arcadia: A Software Development Environment Research Project." In *Proceedings IEEE Conference on Ada Applications and Environments*, April 8-10, Miami Beach, FL.

[Tayl87] Taylor, R.N., D.A. Baker, F.C. Belz, B.W. Boehm, L.A. Clarke, D.A. Fisher, L.J. Osterweil, R.W. Selby Jr., J.C. Wileden, A.L. Wolf, and M. Young. July 1987. *Next Generation Software Environments: Principles, Problems, and Research Directions*. University of Massachusetts. Arcadia Document, Technical Report 87-63. **

[Tayl88] Taylor, R.N., F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby Jr. , J.C. Wileden, A.L. Wolf, and M. Young. 1988. "Foundations for the Arcadia Environment Architecture." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 28-30, Boston, MA, 1-13.

[Teic74] Teichroew, E., M.J. Bastarache, and E.A. Hershey III. March 1974. *An Introduction to PSL/PSA, ISDOS Working Paper No. 86.* University of Michigan. **

[Teic77] Teichroew, D., and E.A. Hershey III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE: Transactions on Software Engineering*, 3/1 (Jan 1977):41-48.

[Teit81] Teitelman, W., and L. Masinter. "The InterLisp Programming Environment." *IEEE: Computer*, 14/4 (Apr 1981):25-33.

[Teit84] Teitelman, W. "A Tour Through Cedar." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 181-195. Washington, DC: IEEE Computer Society Press.

[Thay75] Thayer, T.A. 1975. "Understanding Software Through Empirical Reliability Analysis." In *Proceedings AFIPS National Computer Conference*, vol. 44, May 19-22, Anaheim, CA, 335-341. Montvale, NJ: AFIPS Press.

[Thay76] Thayer, T.A. 1976. *Software Reliability Study*. Griffiss Air Force Base, NY: Rome Air Development Center. **

[Thay78] Thayer, T.A., M. Lipow, and E.C. Nelson. 1978. *Software Reliability*. In *TRW Series of Software Technology, Vol. 2*. New York: North Holland. **

[Thay80] Thayer, R.H., A.B. Pyster, and R.C. Wood. "The Challenge of Software Engineering Project Management." *IEEE: Computer*, 3/1 (Aug 1980):51-59.

[Theb83] Thebaut, S.M. 1983. *The Saturation Effect in Large-Scale Software Development: Its Impact and Control*. Ph.D. thesis, Purdue University. **

[Theb84] Thebaut, S.M., and V.Y. Shen. "An Analytic Resource Model for Large-Scale Software Development." *Information Processing and Management*, 20(1-2) (1984):293-315.

[Thib78] Thibodeau, R. January 31, 1978. *The State-of-the-Art in Software Error Data Collection and Analysis-- Final Report*. Huntsville, AL: General Research Corp. **

[Thib81] Thibodeau, R. April 1981. *An Evaluation of Software Cost Estimating Models*. General Research Corp. Technical Report T10-2670. **

[Thom80] Thompson, W.E., and P.O. Chelson. 1980. "On the Specification and Testing of Software Reliability." In *Proceedings Annual Reliability and Maintainability Symposium*, 379-383.

[Thom81] Thompson, D., and R. Erickson (eds.). February 1981. *AFFIRM Reference Manual*. University Southern California. **

[Thom83] Thomas, J., and N.G. Leveson. 1983. "Applying Safety Design Techniques to Software Safety." In *Proceedings AIAA Space Science Meeting*, January, Reno, NV. **

[Tich79] Tichy, W.F. 1979. "Software Development Control Based on Module Interconnection." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 29-41. Washington, DC: IEEE Computer Society Press. **

[Tich86] Tichy, W.F. "Smart Recompilation." *ACM: Transactions on Programming Languages and Systems*, 8/3 (Jul 1986):273-291.

[Tisc83] Tischler, R., R. Schaufler, and C. Payne. 1983. "Static Analysis of Programs as an Aid to Debugging." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):155-158. Baltimore, MD: ACM Order Department.

[Trio86] Triolet, R. 1986. "Interprocedural Analysis Based Restructuring of Programs." In *Proceedings International Workshop on Parallel Algorithms and Architectures*, April 14-18, Luminy, France, 203-217. **

[Triv80] Trivedi, K.S., J.W. Gault, and J.B. Clary. 1980. "A Validation Prototype of System Reliability in Life-Critical Applications. In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 79-86.

[Troy81] Troy, D.A., and S.H. Zweben. "Measuring the Quality of Structured Designs." *Journal of Systems and Software*, 2/2 (Jun 1981):113-120.

[Troy86] Troy, R. and Y. Romain. "A Statistical Methodology for the Study of the Software Failure Process and Its Application to the ARGOS Center." *IEEE: Transactions on Software Engineering* 12/9 (Sep 1986):968-979.

[Tsal86] Tsalalikhin, L. 1986. "Dialog with a Tester (Architecture and Function of One Unit Test Facility)." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 51-60. Washington, DC: IEEE Computer Society Press.

[Tuck65] Tucker, A.E. January 1965. *The Correlation of Computer Program Quality with Testing Effort*. System Development Corp. Report TM 2219/000/00. **

[Turn80] Turner, J. "The Structure of Modular Programs." *ACM: Communications of the ACM*, 25/5 (May 1980):272-277.

[Turn81a] Turner, C., and G. Caron. May 1981. *A Comparison of RADC and NASA/SEL Software Development Data*. Data and Analysis Center for Software. Special Publication. **

[Turn81b] Turner, C., G. Caron, and G. Brement. April 1981. *NASA/SEL Data Compendium*. Data and Analysis Center for Software. Special Publication. **

[Ullm73] Ullman, J.D. "Fast Algorithms for the Elimination of Common Subexpressions." *Acta Informatica*, 2/3 (Dec 1973):191-123.

[Unde63] Underhill, L.H. "The Growth of Complexity of a General-Purpose Program." *Computer Journal*, 6/1 (Jan 1963):37-38.

[Ural83] Ural, H., and R.L. Probert. 1983. "User Guided Test Sequence Generation." In *Protocol Specification, Testing, and Verification*, H. Rudin and C.H. West (eds.), 421-436. North Holland. **

[Ural84] Ural, H., and R. Probert. 1984. "Automated Testing of Protocol Specifications and Their Implementations." In *Proceedings ACM SIGCOMM 1984 Symposium*, June, Montreal, Canada. **

[Urba73] Urban, R.J. December 1973. *SELFMET: A Program Package for Full Self-Metric Instrumentation of FORTRAN Programs*. General Research Corp. Report RM-1851. **

[Uren87] Uren, E., E. Miller, and J. Irwin. 1987. "Automated Software Testing – Case Studies." In *Proceedings IEEE Conference on Software Maintenance*, September 21-24, Austin, TX. **

[Uyar86] Uyar, M.U., and A.T. Dahbura. 1986. "Optimal Test Sequence Generation for Protocols: The Chinese Postman Algorithm Applied to Q.931." In *Proceedings IEEE Global Telecommunications Conference*, December, 3.1.1-5. **

[Vald83] Valdes, P.M., and A.L. Goel. 1983. "An Error-Specific Approach to Testing." In *Proceedings 8th Annual Software Engineering Workshop*, November, Greenbelt, MD. NASA/GSFC. **

[Vale89] Valett, J.D. and F.E. McGarry. "A Summary of Software Measurement Experiences in the Software Engineering Laboratory." *Journal of Systems and Software*, no. 9 (1989):137-148.

[Vali84] Valiant, L.G. "A Theory of the Learnable." *ACM: Communications of the ACM*, 27/11 (Nov 1984):1134-1142.

[Vemu80] Vemuri, V., and J.V. Cornacchio. 1980. "Figures of Merit for Software Quality." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 744-750. Los Alamitos, CA: IEEE Computer Society Press.

[Vern89] Verner, J.M., G. Tate, B. Jackson, and R.G. Hayward. 1989. "Technology Dependence in Function Point Analysis: A Case Study and Critical Review." In *Proceedings 11th International Conference on Software Engineering*, May 15-18, Pitsburgh, PA, 375-382. Washington, DC: IEEE Computer Society Press.

[Vese81] Vesely, W.E., F.F. Goldberg, N.H. Roberts, and D.F. Haasl. January 1981. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission. Nureg-0492. **

[Vess83] Vessey, L., and R. Weber. "Some Factors Affecting Program Repair Maintenance: An Empirical Study." *ACM: Communications of the ACM*, 26/2 (Feb 1983):128-134.

[Voge80] Voges, U., L. Gmeiner, and A.A. von Mayrhauser. "SADAT – An Automated Testing Tool." *IEEE: Transactions on Software Engineering*, 6/5 (May 1980):286-290.

[Vosb84] Vosburgh, J., B. Curtis, R.W. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu. 1984. "Productivity Factors and Programming Environments." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 143-152. Washington, DC: IEEE Computer Society Press.

[Vose88] Vose M.D. August 1988. *Applications of Compositional Test Generation to Recursively Specify Combinational Logic*. Computational Logic Inc. Technical Report CLI-26. **

[Vouk85a] Vouk, M.A., D.F. McAllister, K.C. Tai, and L.E. Deimel. 1985. *Effectiveness of Random Testing in Detecting Dependent Failures of Fault-Tolerant Software*. North Carolina State University. Technical Report TR-85-02. **

[Vouk85b] Vouk, M.A., D.F. McAllister, and R.K. Scott. 1985. *Influence of Programmer Profile on Correlated Software Errors*. North Carolina State University. Technical Report TR-855-21. **

[Vouk85c] Vouk, M.A., D.F. McAllister, and K.C. Tai. 1985. "Identification of Correlated Failures of Fault-Tolerant Software Systems." In *Proceedings 9th International Computer Software and Applications Conference*, October 9-11, 437-444. Los Angeles, CA: IEEE Computer Society.

[Vouk86a] Vouk, M.A., D.F. McAllister, and K.C. Tai. 1986. "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-Tolerant Software." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 74-81. Washington, DC: IEEE Computer Society Press.

[Vouk86b] Vouk, M.A., M.L. Helsabeck, K-C. Tai, and D.F. McAllister. 1986. "On Testing of Functionally Equivalent Components of Fault-Tolerant Software." In *Proceedings 10th International Computer Software and Applications Conference*, 414-419. Washington, DC: IEEE Computer Society Press.

[Wahl86] Wahl, N.J., S.R. Schach, and R.I. Winner. 1986. "A Very High Level Debugging Simulator for Low Level Microprograms." In *Proceedings 19th Annual Microprogramming Workshop*, New York, 148-155. **

[Wahl88] Wahl, N.J., and S.R. Schach. 1988. "A Methodology and Distributed Tool for Debugging Dataflow Programs." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 98-105. Washington, DC: IEEE Computer Society Press.

[Waka89] Wakahara, Y., Kakuda Y., A. Ito, and E. Utsunomiya. "Escort: An Environment for Specifying Communication Requirements." *IEEE: Software*, 6/2 (Mar 1989):38-45.

[Wake88] Wake, S. and S.M. Henry. 1988. *A Model Based on Software Quality Factors which Predicts Maintainability*. Virginia Polytechnic Institute. TR-88-8.

[Walk81] Walker, M. 1981. *Managing Software Reliability*. Amsterdam: Elsevier North-Holland Publishing.

[Wall89] Wallace, D.R., and R.U. Fujii. "Verification and Validation: Techniques to Assure Reliability." *IEEE: Software*, 6/3 (May 1989):8-9.

[Wals77a] Walston, C.E., and C.P. Felix. "A Method of Programming Measurement and Estimation." *IBM Systems Journal*, 16/1 (Jan 1977):54-73.

[Wals77b] Walston, C.E., and C.P. Felix. Author's response in letters section. *IBM Systems Journal*, 4 (1977). **

[Wals77c] Walsh, D.A. "Structured Testing." *Datamation*, 23/7 (Jul 1977):111-118. **

[Wals79] Walsh, T.J. 1979. "A Software Reliability Study Using a Complexity Measure." In *Proceedings AFIPS National Computer Conference*, vol. 48, June 4-7, New York, NY, 761-768. Arlington, VA: AFIPS Press. **

[Wals85] Walsh, P.J. 1985. *A Measure of Test Case Completeness*. Ph.D. diss., State University of New York, Watson School of Engineering. **

[Walt78] Walters, G., and J.A. McCall. 1978. "The Development of Metrics for Software R&M." In *Proceedings Annual Reliability and Maintainability Symposium*. **

[Walt79] Walters, G.F. 1979. "Application of Metrics to a Software Quality Management (QM) Program." In *Concepts of Software Quality*, J.D. Cooper and M.J. Fisher (eds.), 143-157. Petrocelli Books.

[Wamp85] Wampler, G.K. 1985. *Static Concurrency Analysis of Ada Programs*. Masters thesis, University of California (Irvine).

[Wand79] Wand, M. "Final Algebra Semantics and Data Type Extensions." *Journal of Computer and System Science*, 19/1 (1979). **

[Wang83] Wang, A.S., and H.E. Dunsmore. 1983. "Back-To-Front Programming Effort Prediction." In *Proceedings Symposium Empirical Foundations of Information and Software Science*, November 3-5. Atlanta, GA. Published in *Information Processing and Management*, (1983). **

[Wang84] Wang, A.S. 1984. *The Estimation of Software Size and Effort: An Approach Based on the Evolution of Software Metrics*. Ph.D. thesis, Purdue University. **

[Warn72] Warner, D.C. 1972. "System Performance and Evaluation--Past, Present and Future." In *Proceedings AFIPS Fall Joint Computer Conference*, vol. 41, December 5-7, Anaheim, CA, 959-964. Montvale. NJ: AFIPS Press.

[Warr82] Warren, S. "MAP – A Tool for Understanding Software." In *Proceedings 6th International Conference on Software Engineering*, September 13-16, Tokyo, Japan, 28-37. Washington, DC: IEEE Computer Society Press.

[Wate79] Waters, R.C. "A Method for Analyzing Loop Programs." *IEEE: Transactions on Software Engineering*, 5/5 (May 1979):237-247.

[Webe83] Weber, J.C. 1983. "Interactive Debugging of Concurrent Programs." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):112-113. Baltimore, MD: ACM Order Department.

[Wegb74] Wegbreit, B. "The Synthesis of Loop Predicates." *ACM: Communications of the ACM*, 16/2 (Feb 1974):102-112.

[Wegb75] Wegbreit, B. "Mechanical Program Analysis." *ACM: Communcations of the ACM*, 18/9 (Sep 1975):528-539.

[Wegb76] Wegbreit, B., and J.M. Spitzen, "Proving Properties of Complex Data Structures." *ACM: Journal of the ACM*, 23/2 (Apr 1976):389-396.

[Wegb77] Wegbreit, B. "Constructive Methods in Program Verification." *IEEE: Transactions on Software Engineering*, 3/3 (May 1977):193-207.

[Wegn79] Wegner, P. (ed). *Research Directions in Software Technology*. Cambridge, MA: MIT Press.

[Weid86] Weiderman, N.H., A.N. Habermann, M.W. Borger, and M.H. Klein. 1986. "A Methodology for Evaluating Environments." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 9-11, Palo Alto, CA. Published in *SIGPLAN Notices*, 22/1 (Jan 1987):199-207.

[Wein71] Weinberg, G.M. 1971. *The Psychology of Computer Programming*. Princeton, NJ: Van Nostrand-Reinhold.

[Wein80] Weinberger, E. 1980. "The Applied Bayesian Decision Model in the Risk Assessment Process." In *Proceedings ACM/NBS 19th Annual Technical Symposium: Pathways to System Integrity*, June, Gaithersburg, MD, 55-56.

[Weis74] Weissman, L.M. 1974. *Psychological Complexity of Computer Programs*. Ph.D. diss., University of Toronto. Also published in *ACM: SIGPLAN Notices*, 9/6 (Jun 1974):25-36. **

[Weis78] Weiss, D.M. December 1978. *Evaluating Software Development by Error Analysis*. Naval Research Laboratory. Technical Report NRL-8268. Also published in *Journal of Systems and Software*, Vol. 1 (1979):57-70. **

[Weis81] Weiss, D.M. November 1981. *Evaluating Software Development by Analysis of Change Data*. University of Maryland. Technical Report TR-1120. **

[Weis82] Weiss, D.M. July 1982. *A Comparison of Errors in Different Software-Development Environments*. Naval Research Laboratory, Computer Science and Systems Branch. NRL Report 8598. Also published in *Proceedings 6th Annual Software Engineering Workshop*, December, Grenbelt, MD. NASA/GSFC.

[Weis84] Weiser, M.D. "Program Slicing." *IEEE: Transactions on Software Engineering*, 10/4 (Jul 1984):352-357.

[Weis85a] Weiser, M.D., J.D. Gannon, and P.R. McMullin. "Comparison of Structural Test Coverage Metrics." *IEEE: Software*, 2/2 (Mar 1985):80-85.

[Weis85b] Weiss, S.N., and E.J. Weyuker. February 1985. *A Time-Independent Definition of Software Reliability*. Courant Institute of Mathematical Sciences, New York University. Technical Report #146.

[Weis85c] Weiss, D.M., and V.R. Basili. "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory." *IEEE: Transactions on Software Engineering*, 11/2 (Feb 1985):157-168.

[Weis86] Weiss, S.N., and E.J. Weyuker. 1986. "A Generalized Domain-Based Definition of Software Reliability." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 98-107. Washington, DC: IEEE Computer Society Press. Previously published as Courant Institute of Mathematical Sciences, New York University, Technical Report #146, February 1985. Also published in *IEEE: Transactions on Software Engineering*, 14/10 (Oct 1988):1512-1524.

[Weis87] Weiss, S.N. 1987. *A Theory of Concurrent Programs and Test Data Adequacy*. Ph.D. diss., New York University. **

[Weis88a] Weiss, S.N. 1988. "A Formal Framework for the Study of Concurrent Program Testing." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 106-113. Washington, DC: IEEE Computer Society Press.

[Weis88b] Weiss, S.N., and E.J. Weyuker. "An Extended Domain-Based Model of Software Reliability." *IEEE: Transactions on Software Engineering*, 14/10 (Oct 1988):1512-1524.

[Weis88c] Weiss, S.N. 1988. *A Formal Theory of Concurrent Program Testing*. Hunter College of CUNY. Technical Report CS-TR 88-02. **

[Weis83] Welsh, H.O. 1983. "Distributed Recovery Block Performance in a Real-Time Control Loop." In *Proceedings Real-Time Systems Symposium*, Arlington, VA, 268-276. **

[Wexe87] Wexelblat, A. May 1987. *Report on Scenario Technology*. MCC. Technical Report STP-139-87. **

[Weyu79] Weyuker, E.J. "The Applicability of Program Schema Results to Programs." *International Journal Computer Information Sciences*, 8 (1979):387-403. **

[Weyu80a] Weyuker, E.J. 1980. "The Oracle Assumption of Program Testing." In *Proceedings IEEE 13th Hawaii International Conference on System Sciences*, January, Honolulu, HA, 44-49. **

[Weyu80b] Weyuker, E.J. May 1980. *Measuring the Adequacy of Test Data*. New York University, Courant Institute of Mathematical Sciences. Technical Report 022. **

[Weyu80c] Weyuker, E.J., and T.J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):236-246.

[Weyu81] Weyuker, E.J. January 1981. *An Error-Based Testing Strategy*. New York University, Courant Institute of Mathematical Sciences. **

[Weyu82] Weyuker, E.J. "On Testing Non-Testable Programs." *Computer Journal*, 15/4 (1982):465-470.

[Weyu83] Weyuker, E.J. "Assessing Test Data Adequacy through Program Inference." *ACM: Transactions on Programming, Languages and Systems*, 5/4 (Oct 1983):641-655.

[Weyu84a] Weyuker, E.J. "The Complexity of Data Flow Criteria for Test Data Selection." *Information Processing Letters*, 19/2 (Aug 1984):103-109.

[Weyu84b] Weyuker, E.J. January 1984. *Axiomatizing Software Test Data Adequacy*. New York University, Courant Institute of Mathematical Sciences. Technical Report 99 (Revised April 1986). Also published in *IEEE: Transactions on Software Engineering*, 12/12 (Dec 1986):1128-1138.

[Weyu88a] Weyuker, E.J. 1988. "An Empirical Study of the Complexity of Data Flow Testing." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 188-195. Washington, DC: IEEE Computer Society Press.

[Weyu89] Weyuker, E.J. Author's Reply to "On the Adequacy of Weyuker's Test Data Adequacy Axioms." *IEEE: Transactions on Software Engineering*, 15/4 (Apr 1989):500-501.

[Whit78a] White, L.J., F.C. Teng, H. Kuo, and D. Coleman. 1978. *An Error Analysis of the Domain Testing Strategy*. Ohio State University. Technical Report CISRC-TR-78-2. **

[Whit78b] White, L.J., E.I. Cohen, and B. Chandrasekaran. August 1978. *A Domain Strategy for Computer Program Testing*. Ohio State University. Technical Report 78-4. Also published in *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):247-257.

[Whit80] Whitworth, M.H., and P.A. Szulewski. 1980. "The Measurement of Control and Data Flow Complexity in Software Designs." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 735-743. Los Alamitos, CA: IEEE Computer Society Press.

[Whit81] White, L.J. 1981. "Basic Mathematical Definitions and Results in Testing." In *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (eds.), 3-24. New York: North Holland. **

[Whit85] White, L.J., and P.N. Sahay. "Experiments Determining Best Paths for Testing Computer Program Predicates." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 238-243. Washington, DC: IEEE Computer Society Press. **

[Whit86] White, L.J., and I.A. Perera. 1986. "An Alternative Measure for Error Analysis of the Domain Testing Strategy." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 122-131. Washington, DC: IEEE Computer Society Press.

[Whit88a] White, L.J., and B. Wiszniewski. 1988. "Complexity of Testing Iterated Borders for Structured Programs." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 231-237. Washington, DC: IEEE Computer Society Press.

[Whit88b] White, L. 1988. Position Statements from Panel On: Testing and Verification Problems in Industry: Technology Transfer. In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 239-245. Washington, DC: IEEE Computer Society Press.

[Wich79] Wichman, B.A., and A.H.J. Sale. July 1979. *Pascal Validation Suite*. University of Tasmania. Report R79-3(PVS/4). **

[Wien84] Wiener-Ehrlich, W.K., J.R. Hamrick, and V.F. Rupolo. "Modeling Software Behavior in Terms of a Formal Life Cycle Curve: Implications for Software Maintenance." *IEEE: Transactions on Software Engineering*, 10/4 (Jul 1984):376-383.

[Wigg84] Wiggs, J.E. 1984. *Experimental Validation of Fault-Tolerant Software Reliability Models*. M.S. thesis, North Carolina State University. **

[Wild87] Wild, C. "Automating Software Fault Tolerance." *Journal of Spacecraft and Rockets*, 24/1 (Jan-Feb 1987):86-89. **

[Wild88] Wild, C. 1988. "Generic Constraint Logic Programming and Incompleteness in the Analysis of Software." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 140. Washington, DC: IEEE Computer Society Press.

[Wile80] Wileden, W. "Techniques for Modelling Parallel Systems with Dynamic Structure." *Journal of Digital Systems*, (Summer 1980):177-197. **

[Wile83] Wileden, J.C., J. Sayler, W.E. Riddle, A. Segal, and A. Stavely. "Behavior Specification in a Software Design System." *Journal Computer Systems Software*, 3 (Jun 1983):123-135. **

[Wile84] Wileden, J.C., and L.A. Clarke. 1984. "Feedback-Directed Development of Complex Software Systems." In *Proceedings 1st Software Process Workshop*, February 6-8, Egham, England, 89-93. Los Angeles, CA: IEEE Computer Society. **

[Wile88] Wileden, J.C., A.L. Wolf, C. Fisher, and P. Tarr. 1988. "PGRAPHITE: An Experiment in Persistent Typed Object Management." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 28-30, Boston, MA, 130-142.

[Will79] Williams, G. 1979. "Program Checking." In *Proceedings SIGPLAN '79 Symposium on Compiler Construction*, August, Denver, CO. Published in *ACM: SIGPLAN Notices*, 14/8 (Aug 1979):13-25.

[Will89] Williams, R., and P. Brashear. 1989. "Ada Compiler Validation: Purpose and Practice." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 522-527. Washington, DC: ACM Ada Technical Committee. **

[Wing89] Wing, J.M., and M.R. Nixon. "Extending Ina Jo with Temporal Logic." *IEEE: Transactions on Software Engineering*, 15/2 (Feb 1989):181-197.

[Wint78] Winters, D., N. Ogden, and L.A. Clarke. December 1978. *A Definition of AID: The ATTEST Interface Description Language*. University of Massachusetts. Technical Report 78-15. **

[Wirs83] Wirsing, M., P. Pepper, H. Partsch, W. Dosch, and M. Broy. "On Hierarchies of Abstract Data Types." *Acta Informatica*, 20/1 (Oct 1983):1-34.

[Wisz87] Wiszniewski, B. Jaunary 1987. *Collected Ideas: Can Domain Testing Overcome Loop Analysis?* University of Alberta. Technical Report TR-87-1. **

[Wolf85a] Wolf, A.L., L.A. Clarke, and J.C. Wileden. "Ada-Based Support for Programming-in-the-Large." *IEEE: Software*, 2/2 (Mar 1985):58-71.

[Wolf85b] Wolf, A.L. September 1985. *Language and Tool Support for Precise Interface Control*. Ph.D. diss., University of Massachusetts. COINS Technical Report 85-23. **

[Wolf85c] Wolf, A.L., L.A. Clarke, and J.C. Wileden. 1985. "Interface Control and Incremental Development in the PIC Environment." *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England. Washington, DC: IEEE Computer Society Press. **

[Wolf86a] Wolf, A.L., L.A. Clarke, and J.C. Wileden. 1986. "A Formal Model for Describing and Evaluating Visibility Control Mechanisms." In *Proceedings IEEE Computer Society 1986 International Conference on Computer Languages*, October, Miami Beach, FL, 182-189. **

[Wolf86b] Wolf, A.L. 1986. "An Overview of Arcadia." In *Proceedings ACM SIGAda Future APSE '86 Workshop*, September, Saratoga Springs, NY. Published in *ACM: Ada Letters*. **

[Wolf86c] Wolf, A.L., L.A. Clarke, and J.C. Wileden. September 1986. *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process*. University of Massachusetts. COINS Technical Report 86-51. Also published in *IEEE: Transactions on Software Engineering*, 15/3 (Mar 1989):250-263.

[Wolv74] Wolverton, R.W. "The Cost of Developing Large Scale Software." *IEEE: Transactions on Computers*, C-23/6 (Jun 1974):615-636.

[Wood77] Woodward, M.R., M.A. Hennell, and D. Hedley. 1977. "The Analysis of Control Flow Structure in Computer Programs." In *Proceedings Liverpool University Conference on Combinatorial Programming (CP77)*, T.B. Boffey (ed.), September, 190-201. **

[Wood78] Woods, J.L. 1978. *Path Selection for Symbolic Execution Systems*. Ph.D. thesis, University of Massachusetts. **

[Wood79a] Woodward, M.R., M.A. Hennell, and D. Hedley. "A Measure of Control Flow Complexity in Program Text." *IEEE: Transactions on Software Engineering*, 5/1 (Jan 1979):45-50.

[Wood79b] Woodfield, S.N. "An Experiment on Unit Increase in Programming Complexity." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):76-79.

[Wood80a] Woodfield, S.N. 1980. *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*. Ph.D. thesis, Purdue University. **

[Wood80b] Woodward, M.R., D. Hedley, and M. Hennell. "Experience with Path Analysis and Testing of Programs." *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):278-286.

[Wood80c] Woods, J.L. May 1980. *Path Selection for Symbolic Execution Systems*. Ph.D. diss., University of Massachusetts. **

[Wood81a] Woodfield, S.N., V.Y. Shen, and H.E. Dunsmore. "A Study of Several Metrics for Programming Effort." *Journal of Systems and Software*, 2/2 (Jun 1981):97-103.

[Wood81b] Woodfield, S.N., H.E. Dunsmore, and V.Y. Shen. 1981. "The Effect of Modularization and Comments on Program Comprehension." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 215-223. Washington, DC: IEEE Computer Society Press.

[Wood81c] Woodfield, S.N., V.Y Shen, and H.E. Dunsmore. *A Module Interconnection Complexity Measure*. Authors from Computer Science Dept. at Arizona State University and Purdue University.

[Wood88] Woodward, M.R., and K. Halewood. 1988. "From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 152-158. Washington, DC: IEEE Computer Society Press.

[Wu87a] Wu, D. March 1987. *Syntax Directed and Semantics Aided Mutation*. University of Liverpool. **

[Wu87b] Wu, D., I.J. Riddell, M.A. Hennell, and D. Hedley. April 1987. *The Minimum Set of Test Data on Syntax Directed Mutation of Boolean Expression*. University of Liverpool. **

[Wu87c] Wu, L., V. R. Basili, and K. Reed. 1987. "A Structure Coverage Tool for Ada Software Systems." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 294-302. Washington, DC: ACM Ada Technical

Committee.

[Wu88] Wu, D., M.A. Hennell, D. Hedly, and I.J. Riddell. 1988. "A Practical Method for Software Quality Control via Program Mutation." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 159-170. Washington, DC: IEEE Computer Society Press.

[Wulf76] Wulf, W.A., R.L. London, and M. Shaw. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE: Transactions on Software Engineering*, 2/4 (Dec 1976):253-265.

[Yama83] Yamada, S., M. Ohba, and S. Osaki. "S-Shaped Reliability Growth Modeling for Software Error Detection." *IEEE: Transactions on Reliability*, R-35/5 (May 1983):475-478.

[Yau78] Yau, S.S., J.S. Collofello, and T. McGregor. 1978. "Ripple Effect Analysis of Software Maintenance." In *Proceedings 2nd International Computer Software and Applications Conference*, November 13-16, Chicago, IL, 60-65. Long Beach, CA: IEEE Computer Society Press.

[Yau79] Yau, S.S., and J.S. Collofello. 1979. "Some Stability Measures for Software Maintenance." In *Proceedings 3rd International Computer Software and Applications Conference*, November 6-8, Chicago, IL, 674-679. Long Beach, CA: IEEE Computer Society Press. Also published in *IEEE: Transactions on Software Engineering*, 6/11 (Nov 1980):545-552.

[Yau80] Yau, S.S., and F.-C. Chen. "An Approach to Concurrent Control Flow Checking." *IEEE: Transactions on Software Engineering*, 6/3 (Mar 1980):126-137.

[Yeh77] Yeh, R.T. (ed.) 1977. *Current Trends in Programming Methodology, Vol. 2 Program Validation*. Englewood Cliffs, NJ: Prentice-Hall.

[Yeh79] Yeh, R.T. "In Memory of Maurice H. Halstead." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):74-75.

[Yin78] Yin, B.H., and J.W. Winchester. 1978. "The Establishment and Use of Measures to Evaluate the Quality of Software Designs." In *Proceedings ACM Software Quality Assurance Workshop*, November 15-17, San Diego, CA:45-52. New York: Association for Computing Machinery.

[Yin79] Yin, B.H., and J.W. Winchester. 1979. "Software Design Quality Metrics System." In *Proceedings 2nd International Conference on Mathematical Modeling*, July. **

[Yin80] Yin, B.H. 1980. "Software Design Testability Analysis." In *Proceedings 4th International Computer Software and Applications Conference*, October 27-31, Chicago, IL, 729-734. Los Alamitos, CA: IEEE Computer Society Press.

[Youn71] Youngs, E.A. 1971. *Error-Proneness in Programming*. Ph.D. thesis, University of North Carolina. **

[Youn74] Youngs, E.A. "Human Errors in Programming." *International Journal of Man-Machine Studies*, Vol. 6 (1974):361-376. **

[Youn85] Yount, L.J., K.A. Lievel, and B.H. Hill. 1985. "Fault Effect Protection and Partitioning for Fly-By-Wire/Fly-By-Light Avionics Systems." In *Proceedings 5th AIAA Conference on Computers in Aerospace*, October, Long Beach, CA, 275-284. **

[Youn86a] Young, M., and R.N. Taylor. 1986. "Combining Static Concurrency Analysis with Symbolic Execution." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 170-178. Washington, DC: IEEE Computer Society Press.

[Youn86b] Young, B. April 1986. *The Low-Water-Mark Problem Using Non-Interference*. Honeywell Secure Computing Technology Center. Internal Note. **

[Youn86c] Young, W.D. 1986. *A Verified Compiler for Micro Gypsy*. Ph.D. diss., University of Texas (in progress). **

[Youn88a] Young, M. 1988. "How to Leave Out Details: Error Preserving Abstractions of State-Space Models." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 63-70. Washington, DC: IEEE Computer Society Press.

[Youn88b] Young, M., R.N. Taylor, D.B. Troup, and C.D. Kelly. 1988. "Design Principles Behind Chiron: A UIMS for Software Environments." In *Proceedings 10th International Conference on Software Engineering*, April 11-15, Singapore, 367-376. Washington, DC: IEEE Computer Society Press. **

[Youn89a] Youngblut, C., B. Brykczynski, K. Gordon, R.N. Meeson, and J. Salasin. February 1989. *SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation With Recommended R&D Tasks*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2132.

[Youn89b] Youngblut, C., and B. Brykczynski. *Bibliography of Testing and Evaluation Reference Material*. Alexandria, VA: Institute for Defense Analyses. Draft IDA Memorandum M-496.

[Youn89c] Young, M., and R. Taylor. May 1989. "Rethinking the Taxonomy of Fault Detection Techniques." In *Proceedings 11th International Conference on Software Engineering*, May 15-18, Pitsburgh, PA, 53-62. Washington, DC: IEEE Computer Society Press.

[Your76] Yourdon, E., and L.L. Constantine. 1975. *Structured Design*. New York: Yourdon Inc.

[Yu84] Yu, T.J., and B.A. Nejmeh. 1984. *Software Metrics Data Collection*. Purdue University. Technical Report CSD-TR-421. **

[Yu85] Yu, T.J. 1985. *The Static and Dynamic Models of Software Defects and Reliability*. Ph.D. thesis, Purdue University. **

[Yu88a] Yu, T.J., B.A. Nejmeh, H.E. Dunsmore, and V.Y. Shen. "SMDC: An Interactive Software Metrics Data Collection and Analysis System." *Journal of Systems and Software*, no. 8 (1988).

[Yu88b] Yu, T., V.Y Shen, and H.E. Dunsmore. "An Analysis of Several Software Defect Models," *IEEE: Transactions on Software Engineering*, 14/9 (Sep 1988):1261-1270.

[Zafi80] Zafiropulo, P., C.H. West, H. Rudin, D.D. Cowan, and D. Brand. "Towards Analyzing and Synthesizing Protocols." *IEEE: Transactions on Communications*, COM-28/4 (Apr 1980):651-661.

[Zeig89] Zeigler, J., J.M. Grasso, L.G. Burgermeister, and L.D. Mollod. 1989. "Developing a Universal Ada Test Language for Generating Software/System Integration and Fault Isolation Test Programs." In *Proceedings 7th Annual National Conference on Ada Technology*, March 13-16, Altantic City, NJ, 494-510. Washington, DC: ACM Ada Technical Committee. **

[Zeil81a] Zeil, S.J. 1981. *Selecting Sufficient Sets of Test Paths for Program Testing*. Ph.D. diss., Ohio State University. Technical Report OSU-CISRC-TR-81-10. **

[Zeil81b] Zeil, S.J., and L.J. White. 1981. "Sufficient Test Sets for Path Analysis Testing Strategies." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 184-191. Washington, DC: IEEE Computer Society Press.

[Zeil83a] Zeil, S.J. "Testing for Perturbations of Program Statements." *IEEE: Transactions on Software Engineering*, 9/3 (May 1983):335-346.

[Zeil83b] Zeil, S.J. December 1983. *Perturbation Testing for Domain Errors*. University of Massachusetts. Technical Report 83-38. **

[Zeil84] Zeil, S.J. 1984. "Perturbation Testing for Computation Errors." In *Proceedings 7th International Conference on Software Engineering*, March, 26-29, Orlando, FL, 257-265: . Washington, DC: IEEE Computer Society Press.

[Zeil86] Zeil, S.J. 1986. "The EQUATE Testing Strategy." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 142-151. Washington, DC: IEEE Computer Society Press.

[Zeil87] Zeil, S.J., and E.C. Epp. October 1987. *Interpretation in a Tool-Fragment Environment*. University of Massachusetts. COINS Technical Report 87-108. Also published in *Proceedings 10th International Conference on Software Engineering*, April 11-15, Singapore, 241-248. Washington, DC: IEEE Computer Society Press.

[Zeil88a] Zeil, S.J. 1988. "Selectivity of Data-Flow and Control-Flow Path Criteria." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 216-222. Washington, DC: IEEE Computer Society Press.

[Zeil88b] Zeil, S.J. "Complexity of the EQUATE Testing Strategy." *Journal of Systems and Software*, 8/2 (Mar 1988):91-104.

[Zeil88c] Zeil, S.J. 1988. "Testing Criteria versus Abstraction: Incidental and Inherent Limitations." In *Towards SDS Testing and Evaluation: A Collection of Relevant Topics*. IDA Memorandum Report M-513. Alexandria, VA: Institute for Defense Analyses. Draft.

[Zeil89] Zeil, S.J. "Perturbation Techniques for Detecting Domain Errors." *IEEE: Transactions on Software Engineering*, 15/6 (Jun 1980):737-746.

[Zelk77] Zelkowitz, M.V. 1977. "Operational Aspects of a Software Measurement Facility." In *Proceedings 2nd Life Cycle Management Workshop*, August. **

[Zelk78] Zelkowitz, M.V. "Perspectives on Software Engineering." *ACM: Computing Surveys*, 10/2 (Jun 1978): 197-216.

[Zelk79] Zelkowitz, M.V. 1979. "Resources Estimation for Medium Scale Software Projects." In *12th Conference on Computer Science and Statistics: Proceedings Annual Symposium on the Interface*. New York: IEEE Computer Society Press. **

[Zelk82] Zelkowitz, M.V. 1982. "Data Collection and Evaluation for Experimental Computer Science Research." In *Proceedings Empirical Foundations for Computer and Information Science*, November. **

[Zhao86] Zhao, J.-R., and G.v. Bochmann. 1986. "Reduced Reachability Analysis of Communication Protocols: A New Approach." In *Proceedings 6th IFIP Workshop on Protocols*, June, 243-254. North-Holland. **

[Zhou81] Zhou, C.C., and C.A.R. Hoare. 1981. *Partial Correctness of Communicating Processes and Protocols*. University of Oxford. Technical Monograph PRG-20. **

[Zill74] Zilles, S.N. "Algebraic Specification of Data Types." In *Project MAC Progress Report for 1973-1974*. MIT. CSG Memo 119. **

[Zoln76] Zolnowski, J.M.C. 1976. *A System for Measuring Program Complexity*. Ph.D. diss., Texas A&M University.

[Zoln77] Zolnowski, J.C., and D.B. Simmons. 1977. "Measuring Program Complexity." In *Proceedings COMPCON Fall 1977*, Long beach, CA, 336-340. IEEE. **

[Zoln81] Zolnowski, J.M.C., and D.B. Simmons. 1981. "Taking the Measure of Program Complexity." In *Proceedings AFIPS National Computer Conference*, vol. 50, May 4-7, Chicago, IL, 329-336. Arlington, VA: AFIPS Press.

[Zwan84] Zwanzig, K. (ed.) November 1984. *Handbook for Estimating Using Function Points*. GUIDE Int., GUIDE Project DP-1234. **

[Zweb79] Zweben, S.H., and M. H. Halstead. "The Frequency Distribution of Operators in PL/I Programs." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):91-94.

[Zweb89] Zweben, S.H., and J.S. Gourlay. "On the Adequacy of Weyuker's Test Data Adequacy Axioms." *IEEE: Transactions on Software Engineering*, 15/4 (Apr 1989):496-500.

[vanH68] van Horn, E.C. "Three Criteria for Designing Computing Systems to Facilitate Debugging." *ACM: Communications of the ACM*, 11/5 (May 1968):360-365.

[vanT74] van Tassel, D. 1974. *Program Style, Design, Efficiency, Debugging, and Testing*. Englewood Cliffs, NJ: Prentice-Hall. **

[vonH85] von Henke, F.W., D.C. Luckham, B. Krieg-Brueckner, and O. Owe. 1985. "Semantic Specification of Ada Packages." In *Proceedings SIGAda International Conference*, May, Paris, France. Published in *ACM: Ada Letters*,V/2 (Sep-Oct 1985):185-196.

3. AUTHOR INDEX

Abdel-Ghaly A.A. ..... [Abde86]
Abmayr D.W. ..... [Rums77]
Acree A.T. ..... [Acre79], [Acre80]
Adam A. ..... [Adam80]
Adamowicz M. ..... [Moha76a]
Adams E.N. ..... [Adam84]
Adler M.A. ..... [Kear85], [Kear86]
Adrion W.R. ..... [Adri80], [Adri82], [Bran80], [Cher80a]
[Cher80b]
Agarwal V.K. ..... [Jach84]
Agile C.R. ..... [Agil76]
Agresti W.W. ..... [Agre84a], [Agre84b], [Agre84c], [Agre86]
[Agre87], [Brop87], [Card86a], [Card87a], [Chur86]
Aho A.V. ..... [Aho86]
Air Force Operational Test and Evaluation Center ..... [AFOT82], [AFOT86]
[AFOT87], [AFOT88a], [AFOT88b]
Air Force Systems Command ..... [AFSC86a], [AFSC86b], [AFSC88a], [AFSC88b]
Al-Jarrah M.M.F. ..... [Al-J82]
Albert B. ..... [Vosb84]
Alberts D.S. ..... [Albe76]
Albrecht A.J. ..... [Albr79], [Albr81], [Albr83]
Alexander W.P. ..... [Howa73]
Allen F.E. ..... [Alle71], [Alle74], [Alle76]
Ambler A.L. ..... [Ambl76a], [Ambl76b]
Ammann P.E. ..... [Knig85a]
Amoroso E.G. ..... [Amor89]
Amory W. ..... [Amor75]
Amster S.J. ..... [Amst76]
Anderson E.R. ..... [Ande76b]
Anderson J.D. ..... [Ande88]
Anderson R.B. ..... [Ande79a]
Anderson S. ..... [Sche85]
Anderson T. ..... [Ande76a], [Ande79b], [Ande81], [Ande83]
[Ande85]
Andrews D.M. ..... [Andr81]
Angel M. ..... [Ange89]
Angluin D. ..... [Angl80], [Angl83], [Budd80d]
Angluin D.C. ..... [Angl76]
Angus J.E. ..... [Angu80], [Angu83]
Antle C. ..... [Card85c]
Apt K.R. ..... [Apt80], [Apt81], [Apt83a], [Apt83b]
Aragon R.W. ..... [Maye89]
Ardis M. ..... [Gann80], [Haml79]
Ardoin C.D. ..... [Ardo88], [Kapp88], [Linn88]
Arlat J. ..... [Grna80a], [Grna80b]
Arthur J.D. ..... [Arth88], [Henr85]
Auerheimer B. ..... [Auer85], [Auer86]
Avery S. ..... [Aver84]
Avizienis A. ..... [Aviz75], [Aviz77], [Aviz78], [Aviz84]

[Aviz85], [Aviz87], [Chen78b], [Grna80a], [Grna80b], [Kell83]
Avrunin G.S. ..... [Avru83], [Avru85], [Avru86], [Dill85]
[Dill86], [Dill88c]
Ayache J.M. ..... [Ayac79]
Azema P. ..... [Ayac79]
Azuma M. ..... [Okad82]
Babst T.A. ..... [Babs83], [Sief88]
Baggi D.L. ..... [Bagg78], [Bagg80]
Baiardi F. ..... [Baia84], [Baia85]
Bailey J. ..... [Romb88g], [Romb88h]
Bailey J.E. ..... [Bail83]
Bailey J.W. ..... [Bail80], [Bail81], [Basi82b]
Baker A.L. ..... [Bake79a], [Bake79b], [Bake80]
Baker C.T. ..... [Bake88]
Baker D.A. ..... [Tayl87]
Baker F.T. ..... [Bake72a], [Bake72b], [Bake81], [Selb87b]
Baker W.F. ..... [Bake77]
Balcer M.J. ..... [Ostr88]
Baldwin D. ..... [Bald79]
Balzer R.M. ..... [Balz69], [Balz81], [Balz82], [Cohe82]
Baratta-Perez G. ..... [Taus87a], [Taus87b]
Barbeau M. ..... [Barb88], [Sari88b]
Barkataki S. ..... [Bark89]
Barnes M. ..... [Bish86]
Barondi J.J. ..... [Ives83]
Barrett P.A. ..... [Ande85]
Barringer H. ..... [Barr82], [Barr84], [Barr85]
Barth J.M. ..... [Bart78]
Bartlett K.A. ..... [Bart80]
Bartussek W. ..... [Bart77]
Barzdin J.M. ..... [Barz75]
Basili V.R. ..... [Bail80], [Basi75], [Basi77a], [Basi77b]
[Basi78a], [Basi78b], [Basi78c], [Basi79a], [Basi79b], [Basi79c]
[Basi80a], [Basi80b], [Basi80c], [Basi81a], [Basi81b], [Basi81c]
[Basi81d], [Basi81e], [Basi81f], [Basi81g], [Basi82a], [Basi82b]
[Basi82c], [Basi82d], [Basi83a], [Basi83b], [Basi83c], [Basi83d]
[Basi84a], [Basi84b], [Basi84c], [Basi84d], [Basi85a], [Basi85b]
[Basi85c], [Basi85d], [Basi85e], [Basi85f], [Basi85g], [Basi85h]
[Basi86a], [Basi86b], [Basi86c], [Basi86d], [Basi86e], [Basi87a]
[Basi87b], [Basi87c], [Basi87d], [Basi88], [Brop87], [Card82]
[Doer85], [Freb79], [Gann83], [Gann85], [Gann86], [Hutc83]
[Katz86], [Katz87], [Romb85a], [Romb87b], [Romb88b], [Romb88g]
[Romb88h], [Selb87b], [Selb88b], [Selb88c], [Weis85c], [Wu87c]
Bastani F.B. ..... [Bast78], [Rama81], [Rama82]
Bastarache M.J. ..... [Teic74]
Bates P.C. ..... [Bate81], [Bate82], [Bate83a], [Bate83b]
[Less80]
Battaglia M. ..... [Batt87]
Bauer F.L. ..... [Baue79b], [Baue89]
Bauer J.A. ..... [Baue79a], [Baue79a]
Bayer R. ..... [Hals73b]

Bazzichi F. ..... [Bazz82]
Beane J. ..... [Basi81f]
Beckman L. ..... [Beck76]
Beeler J. ..... [Beel85]
Begeman M. ..... [Conk88]
Behrens C.A. ..... [Behr83]
Beizer B. ..... [Beiz83]
Belady L.A. ..... [Bela76], [Bela77], [Bela81]
Belford P.C. ..... [Belf79]
Belkhouche B. ..... [Belk86]
Bell D.E. ..... [Bell74]
Belz F.C. ..... [Ande76b], [Tayl87], [Tayl88]
Bendell A. ..... [Bend86]
Bender M.E. ..... [Bend89], [Lefk89]
Bendick M. ..... [Care77]
Benejean R. ..... [Bene85]
Bengston N.M. ..... [Beng87]
Bennett D.L. ..... [Rich78]
Benson J.P. ..... [Andr81], [Bens81], [Mill74a], [Page74]
Bentley J.L. ..... [Bent87]
Benyon-Tinker G. ..... [Beny79]
Berard E.V. ..... [Bera83]
Berg H.K. ..... [Berg82]
Berg R.C. ..... [Belf79]
Berlinger E. ..... [Berl80]
Berns G.M. ..... [Bern84]
Berry D.M. ..... [Berr87]
Berzins V. ..... [Lisk79]
Besharatian R.H. ..... [Besh85]
Besson M. ..... [Bess87]
Bevier W.R. ..... [Bevi87], [Bevi88]
Bicevskis J.J. ..... [Barz75]
Biche P.W. ..... [Rube75]
Biebow B. ..... [Bieb85]
Bilsel M.S. ..... [Bils83]
Bird D.L. ..... [Bird83]
Bishop P.G. ..... [Bish86]
Bjorner D. ..... [Bjor78], [Bjor82], [Bjor87]
Black J.P. ..... [Blac81], [Tayl78a], [Tayl80a]
Blaine J.D. ..... [Blai85a], [Blai85b]
Blair J. ..... [Blai71]
Bledose W.W. ..... [Good75c]
Bloom M. ..... [Besh85]
Bloomfield R.E. ..... [Bloo86]
Blum E.K. ..... [Ande76b]
Blum L. ..... [Blum75]
Blum M. ..... [Blum75]
Bochmann G.v. ..... [Boch78], [Boch80], [Boch87a], [Boch87b]
[Boch88], [Dsso85], [Jard83], [Sari82], [Sari84b], [Sari87]
[Sari88c], [Zhao86]
Boebert W.E. ..... [Berg82]

Boehm B.W. ..... [Boeh73], [Boeh75a], [Boeh75b], [Boeh78]
[Boeh81], [Boeh84a], [Boeh84b], [Boeh86], [Boeh87], [Tayl87]
Bohrer R. ..... [Bohr75]
Boies S.J. ..... [Boie72]
Bologna S. ..... [Tayl77b]
Bolsky M.I. ..... [Shoo75]
Bonnett B. ..... [Bonn84]
Booth T.L. ..... [Boot80], [GilkXX], [Shol75]
Borger M.W. ..... [Weid86]
Borodin A. ..... [Boro72]
Borst M.A. ..... [Curt79a], [Shep78]
Bouge L. ..... [Boug85a], [Boug85b], [Boug86]
Bowen C. ..... [McCa87a], [McCa87b]
Bowen J.B. ..... [Bowe78], [Bowe79], [Bowe80], [Bowe84]
Bowen T.P. ..... [Bowe83], [Bowe85], [Press83]
Bowman A.B. ..... [Love76]
Bowser J.H. ..... [Bows87], [BowsXX]
Boyer R.S. ..... [Boye75], [Boye79], [Boye80], [Boye81]
[Boye83], [Boye84a], [Boye84b], [Boye88], [Neum75]
Boysen J.P. ..... [Boys79]
Bozeman R.E. ..... [Jone89]
Bradley G.H. ..... [Brad75]
Brand D. ..... [Bran78], [Zafi80]
Branstad M.A. ..... [Adri82], [Bran80], [Cher80a], [Cher80b]
Brashear P. ..... [Will89]
Bravdica ..... [Jack71]
Brement G. ..... [Turn81b]
Briand J.P. ..... [Bria86]
Brilliant S.S. ..... [Bril84], [Bril87]
Brinch Hansen P. ..... [Brin73], [Brin78]
Brindle A.F. ..... [Brin85]
Brinksma E. ..... [Brin87]
Bristow G. ..... [Bris79]
Britcher R.N. ..... [Brit82], [Brit88]
Britton K.H. ..... [Clem84]
Brooks F.P. ..... [Broo75]
Brooks M. ..... [Broo80c]
Brooks M.F. ..... [Broo80d]
Brooks R. ..... [Less80]
Brooks R.E. ..... [Broo80a]
Brooks W.D. ..... [Broo80b], [Broo81], [Motl76]
Brophy C. ..... [Brop87], [Godf87]
Brown A.R. ..... [Brow73b]
Brown D.B. ..... [Brow89]
Brown J.R. ..... [Boeh78], [Brow72a], [Brow72b], [Brow75]
[Brow76]
Brown P.J. ..... [Brow80a]
Brown S.R. ..... [Brow73a]
Browne J.C. ..... [Brow78], [Brow80b], [Lync81]
Broy M. ..... [Krie86]
Broy M. ..... [Baue79b], [Wirs83]

Bruegge B. ..... [Brue83]
Bruen M.W. ..... [Post87]
Brunelle J.E. ..... [Brun85]
Bruns G. ..... [Brun86]
Bryan D.L. ..... [Luck87]
Bryan W.L. ..... [Brya80]
Brykczynski B. ..... [Bryk89], [Youn89a], [Youn89b]
Buchan P. ..... [Furt81], [Szul80], [Szul81]
Buck F.O. ..... [Buck81]
Buckley F. ..... [Buck79]
Budd T.A. ..... [Acre79], [Budd77], [Budd78a], [Budd78b]
[Budd78c], [Budd80a], [Budd80b], [Budd80c], [Budd80d], [Budd81]
[Budd83a], [Budd83b], [Budd85]
Budinger C.A. ..... [BowsXX], [Guin87]
Bulut N. ..... [Bulu74]
Bunce W.E. ..... [Bunc80]
Burger W.F. ..... [Ambl76b]
Burgermeister L.G. ..... [Zeig89]
Burns J. ..... [Burn78]
Burstall R.M. ..... [Burs74]
Buschbach T. ..... [Hout81]
Buzen J.P. ..... [Denn78]
Byrnes C. ..... [Byrn89]
Caglayan M.U. ..... [Cagl82]
Cailliau R. ..... [Cail79]
Camp J.W. ..... [Camp76]
Campbell I.G. ..... [Krie86]
Campbell R.H. ..... [Camp74], [Camp79]
Canning J.T. ..... [Cann85], [Kafu84], [Kafu85a], [Kafu88]
Cantone G. ..... [Cant89]
Card D.N. ..... [Agre84b], [Card81], [Card82], [Card84]
[Card85a], [Card85b], [Card85c], [Card85d], [Card86a], [Card86b]
[Card87a], [Card87b], [Chur82], [Chur86], [Koer84], [McGa84]
[Page82], [Page84], [Page85]
Carey R. ..... [Care77]
Caron G. ..... [Turn81a], [Turn81b]
Carpenter L.C. ..... [Carp75]
Carpenter N. ..... [Cohe77]
Carre B.A. ..... [Carr80], [Carr82]
Carson S.D. ..... [Cars84]
Cartwright R. ..... [Cart81]
Carver R.H. ..... [Carv88], [Tai86]
Caswell D. ..... [Grad87b]
Cavano J. ..... [Cava78]
Celentano A. ..... [Cele80], [Cele81]
Ceri S. ..... [DiMa85]
Ceriani M. ..... [Ceri81]
Cerny E. ..... [Sari87]
Cha S.D. ..... [Cha87], [Cha88]
Chan M. ..... [Chan84]
Chan P.Y. ..... [Abde86]

Chandersekaran C.S. ..... [Glig87]
Chandrasekaran B. ..... [Whit78b]
Chandrasekharan M. ..... [Chan85]
Chandy K.M. ..... [Chan79], [Chan88], [Misr81], [Misr82]
Chang C.K. ..... [Chan89]
Chang P. ..... [Rich76]
Chang S. ..... [Basi85h]
Chang Y.-F. ..... [Chan89]
Chanon R.N. ..... [Chan73]
Chapin N. ..... [Chap79]
Chapman D. ..... [Chap82]
Charles Stark Draper Laboratory ..... [CSDL80]
Cheatham T.E. ..... [Chea78], [Chea79]
Cheheyl M. ..... [Cheh81]
Chelson P.O. ..... [Thom80]
Chen B. ..... [Chen83]
Chen E.T. ..... [Chen78a], [Chen81]
Chen F.-C. ..... [Yau80]
Chen J.-J. ..... [Chan89]
Chen L. ..... [Aviz77], [Chen78b]
Chen W.T. ..... [Chen75], [Chen76], [Rama75b], [Rama76]
Cheng W.K. ..... [Shat88]
Cherniavsky J.C. ..... [Adri82], [Bran80], [Cher79], [Cher80a]
[Cher80b], [Cher86], [Cher87a], [Cher87b], [Cher88]
Chester D.L. ..... [Ches77]
Cheung R. ..... [Rama74b]
Cheung W.K. ..... [Li87]
Chin G.H. ..... [Rama81]
Choquet N. ..... [Boug85a], [Boug86], [Choq85], [Choq86]
Chou C.-R. ..... [Chan89]
Chou X. ..... [Ohba89]
Chow T.S. ..... [Chow78]
Christensen K. ..... [Chri81]
Chrysler E. ..... [Chry78]
Church V.E. ..... [Agre84c], [Card86a], [Chur82], [Chur86]
Chusho T. ..... [Chus87]
Cicu A. ..... [Ceri81]
Cimitile A. ..... [Cant89]
Cinlar E. ..... [Cinl75]
Clapp J.A. ..... [Amor75]
Clark V.A. ..... [Dunn74]
Clarke E.M. ..... [Schu81], [Sist88]
Clarke L.A. ..... [Clar76a], [Clar76b], [Clar78a], [Clar78b]
[Clar81a], [Clar81b], [Clar81c], [Clar82], [Clar83a], [Clar83b]
[Clar84], [Clar85a], [Clar85b], [Clar86a], [Clar86b], [Clar86c]
[Clar86d], [Clar88a], [Clar88b], [Hass80], [Long88], [Rich78]
[Rich81a], [Rich82], [Rich85a], [Rich85b], [Tayl86b], [Tayl87]
[Tayl88], [Wile84], [Wint78], [Wolf85a], [Wolf85c], [Wolf86a]
[Wolf86c]
Clarson D.R. ..... [Taus87a], [Taus87b]
Clary J.B. ..... [Triv80]

Clements P.C. ..... [Clem84]
Cochran W.G. ..... [Coch50], [Coch53]
Cocke J. ..... [Alle76]
Coffman M.L. ..... [Coff87]
Cohen D. ..... [Cohe82]
Cohen E.A. ..... [Cohe78]
Cohen E.I. ..... [Whit78b]
Cohen J. ..... [Cohe77]
Cohen R.M. ..... [Good79b]
Coleman D. ..... [Gerr85], [Whit78a]
Coles R. ..... [Cole85]
Collofello J.S. ..... [Yau78], [Yau79]
Comer D. ..... [Come79]
Computer Sciences Corporation ..... [CSC78]
Conklin E.J. ..... [Conk86], [Conk88]
Conn R. ..... [Conn87]
Conradi R. ..... [Conr85]
Constantine L.L. ..... [Stev74], [Your76]
Conte S.D. ..... [Cont81], [Shen83]
Conti R.A. ..... [Cont85]
Cook C.R. ..... [Harr85]
Cook J.F. ..... [Cont86], [Cook80]
Cook M.L. ..... [Cook82]
Corkill D.D. ..... [Less80], [Less81]
Cornacchio J.V. ..... [Vemu80]
Cornell L. ..... [Cook81], [Corn76]
Cosloy E. ..... [Holt76]
Coulter N.S. ..... [Coul83]
Cowan D.D. ..... [Zafi80]
Cox G.M. ..... [Coch50]
Cox P.R. ..... [Cox81]
Craigen D. ..... [Crai86], [Crai87a], [Crai87b], [Crai87c]
[Crai87d], [Crai88a], [Crai88b], [Smit88]
Crow J. ..... [Crow85a], [Crow85b]
Cruickshank R.D. ..... [Crui80]
Curnow R.P. ..... [Blac77]
Currit P.A. ..... [Curr83], [Curr86]
Curry R.W. ..... [Curr76]
Curtis B. ..... [Curt79a], [Curt79b], [Curt80], [Curt81]
[Curt83], [Shep78], [Shep79], [Vosb84]
DACS ..... [DACS79a], [DACS79b], [DACS85], [Turn81a]
[Turn81b]
Dahbura A.T. ..... [Sabn85], [Uyar86]
Dahl G. ..... [Bish86]
Dahl O.J. ..... [Dahl72], [Dahl78], [Dahl79a]
Dahll G. ..... [Dahl79b]
Dale C. ..... [Dale86]
Daly E.B. ..... [Daly77]
Dana J.A. ..... [Rube75]
Darringer J.A. ..... [Darr78]
Dasarathy B. ..... [Chan85]

Davis C.F. ..... [Gaff88]
Davis C.L. ..... [Davi85]
Davis E.J. ..... [Amst76]
Davis G.B. ..... [Lite76]
Davis J.S. ..... [Davi88]
Davis M.D. ..... [Davi81], [Davi82a], [Davi83a], [Davi83b]
Davis P.J. ..... [Davi77], [Rowl81a], [Rowl81b]
Davis R.E. ..... [Davi82b]
De Carlini U. ..... [Cant89]
De Francesco N. ..... [Baia85], [DeFr85]
DeKock A. ..... [Harr82]
DeMillo R.A. ..... [Acre79], [Budd78a], [Budd80a], [DeMi77]
[DeMi78], [DeMi79a], [DeMi79b], [DeMi81], [DeMi86a], [DeMi86b]
[DeMi87a], [DeMi87b], [DeMi87c], [DeMi87d], [DeMi88a], [DeMi88b]
DeRemer F. ..... [DeRe76]
DeViot L. ..... [Good84b]
DeWolf J.B. ..... [Furt81], [Szul80], [Szul81], [Szul83]
Deason W.H. ..... [Brow89]
Decker W.J. ..... [Deck82a], [Deck82b], [Gree81]
Deimel L.E. ..... [Vouk85a]
Dekert J.L.F. ..... [Deke81]
Delaney R.P. ..... [Dela88]
Delis A. ..... [Romb88g], [Romb88h]
Demers A. ..... [Nguy86]
Demshki M. ..... [Dems82]
Denning D. ..... [Crow85a], [Crow85b]
Denning P.J. ..... [Denn78]
Deransart P. ..... [Dera85]
Deutsch L.P. ..... [Deut73]
Di Maio A. ..... [DiMa85]
DiVito B.L. ..... [DiVi82]
Diaz M. ..... [Ayac79]
Dickman B.N. ..... [Amst76]
Dijkstra E.W. ..... [Dahl72], [Dijk68], [Dijk76a], [Dijk76b]
Dillon L.K. ..... [Avru86], [Dill81], [Dill84], [Dill85]
[Dill86], [Dill87], [Dill88a], [Dill88b], [Dill88c]
Dillon T.S. ..... [Lew88]
Din C.Y. ..... [Tai85a]
Dingee W.L. ..... [Bail81]
Dircks H.F. ..... [Dirc81]
DoD ..... [Army87], [DOD88], [DODD86a], [DODD86b]
[DODD87], [DODS86], [MIL85]
Doerflinger C.W. ..... [Basi83d], [Doer85]
Doi D.K. ..... [Lee89b]
Donahoo J. ..... [Duva80]
Dosch W. ..... [Wirs83]
Doubleday D.L. ..... [Doub87]
Downs T. ..... [Down85a], [Down85b], [Down86]
Drake D.L. ..... [Mill81b]
Draper N.R. ..... [Drap66]
Draper S. ..... [JohnXX]

Drey C. ..... [Bris79]
Drissi O. ..... [Jard87]
Drongowski P. ..... [Goul72], [Goul74]
Dssouli R. ..... [Boch87a], [Dsso85], [Dsso86]
Duclos L.C. ..... [Ducl82]
Duisberg R.A. ..... [Lond85]
Duke E.L. ..... [Duke89]
Dumas R.L. ..... [Duma83]
Duncan A.G. ..... [Dunc78], [Dunc81]
Dunham J.R. ..... [Dunh81], [Dunh83], [Dunh85], [Dunh86]
[Stan84b]
Dunn O.J. ..... [Dunn74]
Dunn R.H. ..... [Dunn82], [Dunn84]
Dunsmore H.E. ..... [Cont81], [Duns77], [Duns78a], [Duns78b]
[Duns80], [Duns83], [Shen80], [Shen83], [Wang83], [Wood81a]
[Wood81b], [Yu88a], [Yu88b]
Duran E.H. ..... [Form79]
Duran J.W. ..... [Dura78], [Dura80], [Dura81a], [Dura81b]
Duvall L. ..... [Duva80]
Dyer M. ..... [Curr86], [Dyer80], [Dyer81a], [Dyer81b]
[Dyer81c], [Dyer82a], [Dyer82b], [Dyer82c], [Dyer83], [Dyer85a]
[Dyer85b], [Mill87a]
Easterling R.G. ..... [East72]
Eckhardt D.E. ..... [Brun85], [Eckh85]
Eckmann S. ..... [Eckm83a], [Eckm83b], [Eckm84], [Eckm85]
[Soli83]
Eckmann S.T. ..... [Kemm85b]
Edwards B. ..... [Bris79]
Edwards E. ..... [Card85c]
Edwards S.H. ..... [Ardo88], [Linn88]
Eggert P. ..... [Loca80]
Ehrenberger W. ..... [Ehre73], [Ehre76], [Ehre78], [Sagl86]
Ehrig H. ..... [Ehri85]
Ehrlich K. ..... [Solo84]
Ehrlich W.A. ..... [Ehrl87]
Eichhorst H.P. ..... [Howd78d]
Ejiogu L.O. ..... [Ejio87]
Elmendorf W.R. ..... [Elme69], [Elme71], [Elme73]
Elshoff J.L. ..... [Elsh76a], [Elsh76b], [Elsh78a], [Elsh78b]
[Elsh78c], [Elsh84]
Elspas B. ..... [Boye75], [Elsp72a], [Elsp72b], [Elsp73]
[Elsp74], [ElspXX]
Emden M.H. ..... [Emde81]
Emerson E.A. ..... [Clar81b], [Clar86d], [Emer83], [Emer85]
Emerson T.J. ..... [Ehrl87], [Emer84]
Endres A. ..... [Endr75]
Engels ..... [Godo77]
Epp E.C. ..... [Epp86], [Zeil87]
Erickson R. ..... [Thom81]
Erickson R.L. ..... [Eric85]
Erickson W.J. ..... [Sack68]

Eslinger S. ..... [Card82]
Esp D.G. ..... [Bish86]
Eswara S. ..... [Sari88b]
Evangelist C.J. ..... [Bela81]
Evangelist W.M. ..... [Evan83a], [Evan83b], [Evan84a], [Evan84b]
[Evan84c], [Evan87]
Facemire J.L. ..... [Lind85]
Fagan M.E. ..... [Faga74], [Faga76], [Faga86]
Fainter R.G. ..... [Fain85], [Fain86], [Gors80]
Fairfield P. ..... [HennXX]
Fairley R.E. ..... [Fair75], [Fair79]
Fantechi A. ..... [Baia84]
Farr L. ..... [Farr65]
Farr W.H. ..... [Farr83], [Farr88]
Favaro J.M. ..... [Fava79]
Feather M.S. ..... [Feat89]
Federal Information Processing Standards Publication ..... [FIPS77]
Federal Software Testing Center ..... [FSTC83]
Fehri M.C. ..... [Bria86]
Feiertag R.J. ..... [Feie80]
Feiler P.H. ..... [Medi81]
Feldman M.B. ..... [Feld89]
Felix C.P. ..... [Wals77a], [Wals77b]
Fenwick S. ..... [McCa87a], [McCa87b]
Fernandez J.C. ..... [Fern85]
Ferrentino A.B. ..... [Ferr77]
Fetzer J.H. ..... [Fetz88]
Feuer A.R. ..... [Feue79a], [Feue79b]
Finger A.B. ..... [Baue79a], [Baue79a]
Finkel R.A. ..... [Fink83], [Gord86], [Gord88]
Finnell C.A. ..... [Taus87a], [Taus87b]
Fischer K.F. ..... [Fisc77]
Fisher C. ..... [Wile88]
Fisher D.A. ..... [Tayl87]
Fitsos G.P. ..... [Chri81], [Fits79], [Fits80]
Fitzsimmons A.B. ..... [Fitz78a], [Fitz78b]
Flon L. ..... [Flon77], [Flon78a], [Flon78b], [Flon81]
Floyd R.W. ..... [Floy67]
Ford R. ..... [Gill88]
Forghani B. ..... [Forg87]
Formann E.H. ..... [Form77]
Forward K.E. ..... [Lew88]
Fosdick L.D. ..... [Fosd74], [Fosd76a], [Fosd76b], [Oste75a]
[Oste75b], [Oste76a], [Oste76b]
Foshee G.L. ..... [Stuc75a]
Foster K.A. ..... [Fost80], [Fost83], [Fost84], [Fost85]
Fowlkes E.B. ..... [Feue79a], [Feue79b]
Francez N. ..... [Apt80], [Fran79], [Fran80]
Frankl P.G. ..... [Fran85a], [Fran85b], [Fran86], [Fran88]
Franta W.R. ..... [Berg82]
Freburger K. ..... [Basi81a], [Freb79]

Freedman S.B. ..... [Mill87b]
Freiman F.R. ..... [Frei79]
Freudenberger S.M. ..... [Freu84]
Frewin G.D. ..... [Hame82]
Fribourg L. ..... [Boug85a], [Boug86], [Choq85]
Froome P.K. ..... [Bloo86]
Fryer S. ..... [Frye81]
Fujii M.S. ..... [Fuji77]
Fujii R.U. ..... [Musa89], [Wall89]
Funami Y. ..... [Funa75]
Fung C.K-C. ..... [Fung85]
Furtek F.C. ..... [Furt81]
GRC ..... [GRC79]
Gabow H.N. ..... [Gabo76]
Gaffney J.E. ..... [Albr83], [Brit82], [Crui80], [Gaff79]
[Gaff80], [Gaff81a], [Gaff81b], [Gaff88]
Galiano E. ..... [Gali87a], [Gali87b], [Math87a], [Math87b]
Gallier J.H. ..... [Gall81]
Gallimore R. ..... [Gerr85]
Gannon C. ..... [Gann79]
Gannon J.D. ..... [Basi82b], [Duns77], [Duns80], [Gann75]
[Gann76], [Gann77], [Gann80], [Gann81], [Gann83], [Gann85]
[Gann86], [Haml79], [McMu80], [McMu83], [Weis85a]
Ganzinger H. ..... [Krie86]
Garcia-Molina H. ..... [Garc83], [Garc84]
Garey M. ..... [Gare78]
Garman J.R. ..... [Garm81]
Gaudel M.C. ..... [Boug85a], [Boug86], [GaudXX]
Gault J.W. ..... [Scot84a], [Scot84b], [Scot87], [Triv80]
Geber G.W. ..... [Boch87b]
Geiger W. ..... [Geig79]
Geller M. ..... [Gell78]
Gelperin G. ..... [Gelp88]
Gelpert D. ..... [Gelp79]
General Electric Co. ..... [GE77a], [GE77b]
Georgia Institute of Technology ..... [STE86]
Gerhart S.L. ..... [Brun86], [Gerh76a], [Gerh76b], [Gerh78]
[Gerh79], [Gerh80], [Gerh84], [Gerh85], [Gerh88a], [Gerh88b]
[Good75a], [Good75b], [Land86]
German S.M. ..... [Germ82a], [Germ82b], [Germ84], [Luck79b]
Germano F. ..... [Garc83], [Garc84]
Gerrard C.P. ..... [Gerr85]
Gerth R.B. ..... [Gert84]
Getz S.L. ..... [Getz83]
Ghezzi C. ..... [Cele81], [Mand85]
Ghosh S. ..... [Luck86b], [Luck86c]
Giammo T. ..... [Giam86]
Gibson V.R. ..... [Gibs89]
Gideadi A.N. ..... [Gide74]
Gilb T. ..... [Gilb76], [Gilb79]
Gilkey T.J. ..... [GilkXX]

Gilles J. ..... [Gill88]
Ginzberg M.G. ..... [Ginz65]
Girard E. ..... [Gira73]
Girgis M.R. ..... [Girg85], [Girg86a], [Girg86b]
Glasgow J.L. ..... [Skil89]
Glass R.L. ..... [Glas79], [Glas80], [Glas81]
Gligor V.D. ..... [Glig87]
Gmeiner L. ..... [Geig79], [Gmei79], [Voge80]
Godfrey S. ..... [Godf87]
Godoy ..... [Godo77]
Goel A. ..... [Goel89]
Goel A.L. ..... [Goel78], [Goel79], [Goel80a], [Goel80b]
[Goel80c], [Goel81], [Goel82], [Goel83], [Goel85], [Goel88]
[Vald83]
Goff R. ..... [HenrXX]
Goguen J.A. ..... [Gogu78], [Gogu79a], [Gogu79b], [Gogu80]
Goldberg F.F. ..... [Vese81]
Goldberg J. ..... [Gold80]
Goldman N. ..... [Balz82]
Gollis M.L. ..... [Mend79]
Good D.I. ..... [Ambl76b], [Good70], [Good75c], [Good75e]
[Good79b], [Good82a], [Good82b], [Good84a], [Good84b], [Good86a]
[Good88], [Land86]
Goodenough J.B. ..... [Good75a], [Good75b], [Good75d], [Good79a]
[Good86b]
Goodwin M.A. ..... [Krau73]
Gopal A.S. ..... [Budd85]
Gordon A.J. ..... [Gord85b], [Gord86], [Gord88]
Gordon K. ..... [Gord85a], [Youn89a]
Gordon R.D. ..... [Gord76], [Gord77], [Gord79a], [Gord79b]
Gorlick M.M. ..... [Gorl87]
Gorsline G.W. ..... [Gors80]
Gorzela R.A. ..... [Perk87]
Gould J.D. ..... [Boie72], [Goul72], [Goul74], [Goul75]
Gourlay J.S. ..... [Gour81], [Gour83], [Zweb89]
Grady R.B. ..... [Grad87a], [Grad87b]
Grant E.E. ..... [Sack68]
Grant E.L. ..... [Gran72]
Grasso J.M. ..... [Zeig89]
Gratz R. ..... [Baue79b]
Gray M.A. ..... [Kear85], [Kear86]
Gray M.D. ..... [Blac77]
Gray T.E. ..... [Boeh84a]
Green A.L. ..... [Gree81]
Green M.W. ..... [Elsp72b], [ElspXX]
Green T.F. ..... [Brad75], [Gree76]
Greene H. ..... [Lefk89]
Greenlaw T. ..... [Gree87], [Math87a]
Gremillion L.L. ..... [Grem84]
Gries D. ..... [Grie76], [Grie77], [Grie79], [Grie81]
[Levi81], [Nguy86], [Owic76]

Griest T.E. ..... [Bend89]
Griffth P.F. ..... [Grif72]
Grine V.S. ..... [Knig85b]
Grnarov A. ..... [Grna80a], [Grna80b]
Groves L.J. ..... [Grov80]
Guindi D.S. ..... [DeMi87d], [DeMi88a], [Guin87], [Guin89]
Guttag J.V. ..... [Gutt75], [Gutt77], [Gutt78a], [Gutt78b]
[Gutt80], [Gutt85]
Haasl D.F. ..... [Vese81]
Haberler M.A. ..... [Luck87], [Meld88]
Habermann A.N. ..... [Camp74], [Habe75], [Weid86]
Hagelstein J. ..... [Bieb85]
Hakimi S.L. ..... [Ntaf79], [Ntaf81b]
Halewood K. ..... [Wood88]
Haley A. ..... [Hale82]
Hall D. ..... [McGa85b]
Hall M.L. ..... [Hall80]
Hall W.E. ..... [Hall86]
Haller A.P. ..... [Hall73], [Hall74]
Halliwell D.N. ..... [Ande85]
Halpern J.D. ..... [Halp87]
Halpern J.Y. ..... [Emer83]
Halstead M.H. ..... [Bulu74], [Come79], [Cook81], [Corn76]
[Funa75], [Gord76], [Hals72a], [Hals73a], [Hals73b], [Hals75a]
[Hals75b], [Hals76], [Hals77a], [Hals77b], [Hals77c], [Hals77d]
[Hals78], [Zweb79]
Hamer P.G. ..... [Hame82]
Hamlet D. ..... [Haml86], [Haml87], [Haml88]
Hamlet P.A. ..... [Haml78c]
Hamlet R.G. ..... [Gann80], [Gann81], [Haml77a], [Haml77b]
[Haml78a], [Haml78b], [Haml79], [More81]
Hamrick J.R. ..... [Wien84]
Han Y.W. ..... [Han76]
Hanata S. ..... [Naka89]
Haney H.M. ..... [Hane72]
Hanford K.V. ..... [Hanf70]
Hanks J.M. ..... [Hank80]
Hannan T.L. ..... [Belf79]
Hansen H.L. ..... [Hans84]
Hansen P.B. ..... [Hans73]
Hansen W.J. ..... [Hans78]
Hantler S. ..... [Hant76]
Haraldson A. ..... [Beck76]
Harandi M.T. ..... [Hara83]
Harel E. ..... [Hare82]
Harris K. ..... [Henr81a]
Harrison L.J. ..... [Dill88a], [Harr88a], [Harr88c]
Harrison W. ..... [Harr81a], [Harr81b], [Harr82], [Harr85]
[Harr88b]
Hart J.J. ..... [Hart79]
Harter P.K. ..... [Hart84]

Hartmanis J. ..... [Hart71]
Harver P. ..... [Harv82]
Harvey P.R. ..... [Leve83b]
Hassell J. ..... [Hass80]
Hatch M.J. ..... [Holt78]
Hayward R.G. ..... [Vern89]
He C.S. ..... [Boch88]
He X. ..... [Lee88]
Hecht H. ..... [Hech76], [Hech77b], [Hech77c], [Hech79]
[Hech80], [McCa87a], [McCa87b]
Hecht M. ..... [McCa87a], [McCa87b]
Hecht M.S. ..... [Hech72], [Hech75], [Hech77a]
Hedley D. ..... [Henn76a], [Henn76b], [Henn79], [Henn84]
[Ridd80], [Wood77], [Wood79a], [Wood80b], [Wu87b], [Wu88]
Heidler W. ..... [Heid82]
Heller G.L. ..... [Gaff80], [Hell87]
Heller P. ..... [Shne77a]
Hellerman L. ..... [Hell72]
Helmbold D.P. ..... [Germ82a], [Helm83], [Helm84a], [Helm84b]
[Helm85], [Luck87]
Helsabeck M.L. ..... [Vouk86b]
Henderson P. ..... [Hend75]
Hengeveld W. ..... [Heng87]
Hennell M.A. ..... [Henn76a], [Henn76b], [Henn78], [Henn79]
[Henn81], [Henn84], [HennXX], [Ridd80], [Wood77], [Wood79a]
[Wood80b], [Wu87b], [Wu88]
Henry R.M. ..... [Grif72]
Henry S.M. ..... [Henr79], [Henr81a], [Henr81b], [Henr84]
[Henr85], [Henr88a], [Henr88b], [HenrXX], [Kafu81], [Wake88]
Herd J.R. ..... [Herd79]
Herman P.M. ..... [Herm76]
Herndon M.A. ..... [McCa84]
Hershey E.A. III ..... [Teic74], [Teic77]
Hess J.A. ..... [Hess88]
Hess R. ..... [Budd80c]
Hesse W. ..... [Baue79b]
Hetzel W.C. ..... [Gelp88]
Hetzel W.C. ..... [Hetz73], [Hetz76], [Hetz84]
Hewitt C. ..... [Hewi76], [Lieb80]
Hewlett-Packard Company ..... [HCP82]
Hibbard P.G. ..... [Brue83], [Hibb82]
Hill B.H. ..... [Youn85]
Hill C.R. ..... [Hill83]
Ho P. ..... [Ho79]
Ho S.-B.F. ..... [Ho78], [Long77], [Ra 4a], [Rama75a]
[R 6], [Rama80]
Hoare C.A.R. ..... [Dahl72], [Fran79], [Hoar69], [Hoar71a]
[Hoar71b], [Hoar72], [Hoar74], [Hoar75], [Hoar78], [Hoar81]
[Hoar85], [Hoar87], [Kenn80], [Zhou81]
Hoben S. ..... [Vosb84]
Hocking D.E. ..... [DeMi81]

Hodges B.C. ..... [Hodg76]
Hoermann H.A. ..... [Hoer74]
Hoffman H.-M. ..... [Hoff77], [Schn77a], [Schn79a]
Hoffman R.H. ..... [Brow72a], [Hoff73], [Hoff75], [Hoff76]
Holloway G.H. ..... [Chea79]
Holthouse M.A. ..... [Holt76], [Holt78]
Holzmann G. ..... [Holz82]
Honeywell Inc. ..... [HONE80]
Hopcroft J.E. ..... [Hart71]
Horning J.J. ..... [Gann75], [Gutt78b], [Gutt80], [Gutt85]
[Horn74]
Horowitz E. ..... [Gutt78a]
Houssais B. ..... [Hous77]
Houtz C. ..... [Hout81]
Howard G.T. ..... [Brad75], [Gree76]
Howard J.H. ..... [Howa73]
Howatt J.W. ..... [Howa85], [Shaw89]
Howden W.E. ..... [Howd74a], [Howd74b], [Howd74c], [Howd75a]
[Howd75b], [Howd76a], [Howd76b], [Howd76c], [Howd76d], [Howd76e]
[Howd77a], [Howd77b], [Howd77c], [Howd78a], [Howd78b], [Howd78c]
[Howd78d], [Howd78e], [Howd78f], [Howd79], [Howd80a], [Howd80b]
[Howd80c], [Howd80d], [Howd81a], [Howd81b], [Howd81c], [Howd81d]
[Howd82a], [Howd82b], [Howd83], [Howd85], [Howd86], [Howd87]
[Howd88], [Howd89a], [Howd89b], [Mill81a]
Howes N.R. ..... [Howe84]
Hsieh C-C. ..... [Hsie82]
Hsieh C.S. ..... [Hsie89]
Huang J.C. ..... [Huan75], [Huan78], [Huan79]
Huet G. ..... [Huet80]
Huh Y. ..... [Luck86b], [Luck86c]
Hullot J.M. ..... [Huet80]
Humphrey W.S. ..... [Hump88]
Humphreys P. ..... [Bish86]
Hunt W.A. Jr. ..... [Hunt85], [Hunt87]
Hutchens D.H. ..... [Basi80b], [Basi86a], [Hutc83]
Hutchinson J.S. ..... [Dunc81]
Hwang S.-S.V. ..... [Hwan81]
IEEE ..... [IEEE83a], [IEEE83b], [IEEE83c], [IEEE84]
[IEEE87], [IEEE88]
ISO ..... [ISO87a], [ISO87b], [ISO87c]
Iannino A. ..... [Iann84], [Musa87]
Ibarra O.H. ..... [Ibar82]
Igarashi S. ..... [Igar73]
Ignalls D.H. ..... [Igna71]
Infotech International Ltd. ..... [INFO76], [INFO79]
Inglis J. ..... [Ingl86]
Ireland E.A. ..... [Davi85]
Irwin J. ..... [Uren87]
Isoda S. ..... [Isod87]
Itakura M. ..... [Itak82]
Ito A. ..... [Waka89]

Ives B. ..... [Ives83]
Jachner J. ..... [Jach84]
Jackson ..... [Jack71]
Jackson B. ..... [Vern89]
Jackson K.L. ..... [SanA83]
Jahanian F. ..... [Jaha86]
Jalote P. ..... [Jalo89]
James T. ..... [Jame77]
Jard C. ..... [Jard83], [Jard87]
Jarratt R.M.A. ..... [Jarr84]
Jeffery D.R. ..... [Jeff85]
Jelinski J. ..... [Jeli72], [Jeli73], [Mora72]
Jenkins J.R. ..... [Jenk86], [Lind88a]
Jensen E.P. ..... [Camp76]
Jensen R.W. ..... [Jens83a], [Jens83b]
Jiang W. ..... [Glig87]
Johnson C. ..... [Brun86]
Johnson D. ..... [Gare78]
Johnson D.B. ..... [Brow78], [John77]
Johnson J.D. ..... [John83]
Johnson J.P. ..... [John75]
Johnson M.S. ..... [John78], [John79], [John82a], [John82b]
Johnson P.E. ..... [Sedl83]
Johnson W.L. ..... [John84], [JohnXX]
Johnston D.E. ..... [John81]
Johri A. ..... [Glig87]
Joint Logistics Commanders ..... [JLC84]
Jones A.M. ..... [Jone89]
Jones C.B. ..... [Bjor78], [Bjor82], [Jone80]
Jones T.C. ..... [Jone76], [Jone78], [Jone79], [Jone81]
Jordan Q.L. ..... [Chur86]
Joyce E. ..... [Joyc87a]
Joyce J. ..... [Joyc87b]
Joyner W.H. ..... [Bran78]
Juozitis P. ..... [Ange89]
Kafura D.G. ..... [Henr81a], [Henr81b], [Henr84], [Henr88b]
[Hite88], [Kafu81], [Kafu84], [Kafu85a], [Kafu85b], [Kafu88]
Kahn G. ..... [Kahn77]
Kakuda Y. ..... [Waka89]
Kalligiannis G. ..... [Getz83]
Kalninsh A.A. ..... [Barz75]
Kamayachi Y. ..... [Taka89]
Kamin S. ..... [Kami80]
Kant K. ..... [Kant80]
Kappel M.R. ..... [Ardo88], [Kapp88], [Linn88]
Karp R.A. ..... [Luck79b]
Kaspar H. ..... [Boeh78]
Kato T. ..... [Kato86]
Katz E.E. ..... [Basi83a], [Basi84c], [Basi84d], [Basi85h]
[Basi86c], [Gann83], [Gann85], [Gann86], [Katz86], [Katz87]
Katz R. ..... [Blac77]

Katz S.M. ..... [Katz73]
Kaufmann M. ..... [Kauf87a], [Kauf87b]
Kearney J.K. ..... [Kear85], [Kear86]
Keeton-Williams J. ..... [Good79b]
Keiler P.A. ..... [Keil87]
Keller R.M. ..... [Kell76]
Keller S.E. ..... [Kell85a], [Kell85b], [Payt82], [Perk86]
Kelly C.D. ..... [Tayl86a], [Youn88b]
Kelly J. ..... [Bark89]
Kelly J.P.J. ..... [Aviz84], [Kell82], [Kell83]
Kemmerer R.A. ..... [Auer85], [Auer86], [Blai85a], [Dill88a]
[Eckm83b], [Eckm84], [Eckm85], [Harr88c], [Kemm80], [Kemm81]
[Kemm85a], [Kemm85b], [Kemm86], [Kemm87], [Soli83]
Kennaway J.R. ..... [Kenn80]
Kennedy K.W. ..... [Kenn75]
Kenney G.W. ..... [John83]
Kernighan B.W. ..... [Bent87], [Kern74a], [Kern74b], [Kern81]
Kerr R. ..... [Ande76a]
Kesselman C.F. ..... [Gorl87]
Khoshgoftaar T.M. ..... [Muns89]
Kieburtz R.B. ..... [Kieb83]
Kim K.H. ..... [Rama74b], [Rama75b]
King J.C. ..... [Darr78], [Hant76], [King69], [King70]
[King75a], [King75b], [King76]
King K.N. ..... [DeMi87d], [DeMi88a], [Offu87]
Kirchoff K. ..... [Snee78]
Kishimoto Z. ..... [Chan85]
Kitchenham B.A. ..... [Kitc81]
Klein M.H. ..... [Weid86]
Kluczny R. ..... [Harr82]
Knight J.C. ..... [Ande83], [Cha87], [Dunh81], [Knig84]
[Knig85a], [Knig85b], [Knig86a], [Knig86b]
Knijff D.J.J. van der ..... [Knij78], [Lass79], [Lass81]
Knuth D.E. ..... [Knut71], [Knut73]
Koerner K. ..... [Koer84]
Kohler W. ..... [Garc83], [Garc84]
Kolstad R.B. ..... [Camp79]
Koppang R.G. ..... [Kopp76]
Korel B. ..... [Kore85], [Kore86a], [Kore86b], [Kore87]
[Kore88], [Lask83]
Korelsky T. ..... [Kore84]
Korsak A. ..... [ElspXX]
Kosaraju R. ..... [Kosa72]
Koss W.E. ..... [Koss88]
Kosy D.W. ..... [Kosy73]
Koukoulidis V. ..... [Sari88b]
Kracik P.J. ..... [Krac78]
Krause K.W. ..... [Krau73]
Krauser E.W. ..... [Krau86], [Krau88], [Math86], [Math88a]
Krieg-Brueckner B. ..... [Baue79b], [Krie80], [Krie83], [Krie86]
[Luck84b], [vonH85]

Kroger F. ..... [Krog87]
Kromodimoeljo S. ..... [Crai88b], [Pase87a], [Pase87b]
Kron H. ..... [DeRe76]
Kroon J. ..... [Heng87]
Kruesi E. ..... [Basi82b], [Dunh83], [Shep81]
Kruger G.A. ..... [Krug88]
Kruszewski G. ..... [Krus78]
Kuhn W.W. ..... [Kuhn82]
Kuiper R. ..... [Barr84]
Kulkarni V.G. ..... [Nico87]
Kuo H. ..... [Whit78a]
Kuoni J.P. ..... [Amst76]
Kwan S.P. ..... [Parn88]
Kwon I.S. ..... [Lind88b]
Ladkin P. ..... [Crow85a], [Crow85b]
Laemmel A. ..... [Laem78], [Shoo77b]
Lahti J. ..... [Bish86], [Dahl79b]
Lam S.S. ..... [Lam84]
Lamb D.A. ..... [Lamb83]
Lamb S.S. ..... [Lamb78]
Lamport L. ..... [Lamp77], [Lamp78], [Lamp79a], [Lamp79b]
[Lamp80], [Lamp82], [Lamp83], [Lamp84], [Owic82]
Landrault C. ..... [Land77]
Landry S.P. ..... [Land79]
Landwehr C.E. ..... [Land86]
Laprie J.-C. ..... [Land77], [Lapr84]
Larsen H.L. ..... [Luck81]
Laski J.W. ..... [Kore85], [Kore88], [Lask79], [Lask82]
[Lask83], [Lask86], [Lask87], [Lask88a], [Lask88b]
Lasseter G. ..... [Hall73]
Lassez J.-L. ..... [Lass79], [Lass81]
Latella D. ..... [DeFr85], [Late84]
Latour L. ..... [Lato89]
Laub J. ..... [Howd75b]
Lauesen S. ..... [Laue79]
Laurent J. ..... [Adam80]
Lavener R.G. ..... [Lave88]
Lawlis P.K. ..... [Lind87]
Lawrence M.J. ..... [Jeff85], [Lawr81]
Lawrynuik D. ..... [Lawr87]
Lawson D.J. ..... [Laws83]
LeBlanc R.J. ..... [Davi88]
LeDoux C.H. ..... [LeDo85]
Leach R.J. ..... [Leac87]
Lease D.M. ..... [Perk86]
Leavenworth R.S. ..... [Gran72]
Leck V.G. ..... [Lamb78]
Ledgard H. ..... [Gide74]
Lee A.J. ..... [Lee89b]
Lee J.A.N. ..... [Lee88]
Lee L.D. ..... [Eckh85]

Lee P.-N. ..... [Lee89a]
Lee P.A. ..... [Ande81]
Lefkowitz L. ..... [Less80]
Lefkowitz S. ..... [Lefk89]
Lehman M.M. ..... [Bela76], [Lehm80]
Lehmann D.J. ..... [Fran79]
Lei C.-L. ..... [Emer85]
Leininger B.S. ..... [Ibar82]
Lesser V.R. ..... [Less80], [Less81]
Leung H.K. ..... [Leun88]
Leung T.-K. ..... [Sidh89]
Levendel Y. ..... [Leve86a]
Leveson N.G. ..... [Cha87], [Cha88], [Knig86a], [Knig86b]
[Land86], [Leve83a], [Leve83b], [Leve83c], [Leve83d], [Leve86b]
[Leve87], [Leve89a], [Shim88], [Thom83]
Levin G.M. ..... [Levi80], [Levi81]
Levine D.P. ..... [Lind87]
Levitt K.N. ..... [Boye75], [Elsp72a], [Elsp72b], [Elsp73]
[Levi78], [Neum75], [Robi79], [Silv79]
Levy M.R. ..... [Levy84]
Lew K.S. ..... [Lew88]
Li H.F. ..... [Li87]
Lieberman H. ..... [Lieb80]
Lievel K.A. ..... [Youn85]
Ligett D. ..... [Dems82]
Light W. ..... [Ligh76]
Ligon W.E. ..... [Ligo87], [Math87a]
Lin H. ..... [Lin85]
Lind R.K. ..... [Lind89]
Linden T.A. ..... [Lind76]
Lindquist T.E. ..... [Fain86], [Lind85], [Lind87], [Lind88a]
[Lind88b], [Lind88c]
Lindsay P.A. ..... [Lind88d]
Linger R.C. ..... [Dyer80], [Ling79], [Mill87a]
Linn B. ..... [Dems82]
Linn C.J. ..... [Ardo88], [Kapp88], [Linn88]
Linn J.L. ..... [Ardo88], [Kapp88], [Linn88]
Lipow M. ..... [Boeh78], [Brow75], [Brow76], [Lipo73]
[Lipo77], [Lipo79], [Thay78]
Lipton R.J. ..... [Acre79], [Budd78a], [Budd78b], [Budd78c]
[Budd80a], [DeMi77], [DeMi78], [DeMi79a], [DeMi79b], [Lipt78]
Liskov B.H. ..... [Lisk75], [Lisk79]
Lister A.M. ..... [John81], [List82]
Litecky C.R. ..... [Lite76]
Littlewood B. ..... [Abde86], [Iann84], [Litt73], [Litt75]
[Litt76], [Litt78], [Litt79], [Litt80a], [Litt80b], [Litt80c]
[Litt81a], [Litt81b]
Liu Y. ..... [Vosb84]
Lo P. ..... [Lo83]
Locasso R. ..... [Loca80]
Loggia-Ramsey C. ..... [Basi85g]

Logrippo L. ..... [Bria86]
Lohse J.B. ..... [Lohs84]
Lomow G. ..... [Joyc87b]
London R.L. ..... [Good70], [Good75c], [Igar73], [Lond70]
[Lond71], [Lond75], [Lond85], [Wulf76]
Long A.B. ..... [Long77]
Long D.L. ..... [Long88]
Love L.T. ..... [Curt79a], [Fitz78a], [Love76], [Love77a]
[Love77b], [Shep77], [Shep78], [Shep79]
Lucas S. ..... [Jens83b]
Luckenbaugh G.L. ..... [Glig87]
Luckham D.C. ..... [Germ82a], [Helm83], [Helm84a], [Helm85]
[Igar73], [Krie80], [Luck77], [Luck79a], [Luck79b], [Luck80a]
[Luck80b], [Luck80c], [Luck81], [Luck84a], [Luck84b], [Luck85]
[Luck86a], [Luck86b], [Luck86c], [Luck87], [Meld88], [vonH85]
Lukey F.J. ..... [Luke80]
Lynch W.C. ..... [Lync81]
MacQueen D. ..... [Kahn77]
Magel K. ..... [Harr81a], [Harr81b], [Harr82]
Maghsoodloo S. ..... [Brow89]
Maheshward S.N. ..... [Gabo76]
Mahr B. ..... [Ehri85]
Maibaum T.S.E. ..... [Emde81]
Maiocchi M. ..... [Cele81], [Ceri81]
Maione A. ..... [Koer84]
Maitland R. ..... [Mait80]
Majoros M. ..... [Majo83]
Malec H. ..... [Vosb84]
Maluszynski J. ..... [Dera85]
Mancarella P. ..... [Manc83]
Mandriolo D. ..... [Mand85]
Maness R.S. ..... [Shaw89]
Manna Z. ..... [Katz73], [Mann70], [Mann74], [Mann78]
Mardinly S. ..... [Payt82]
Mark L. ..... [Romb87c], [Romb88a], [Romb88b], [Romb88c]
[Romb89b]
Marotta D.A. ..... [Gorl87]
Marrcotty M. ..... [Elsh78c]
Marre B. ..... [GaudXX]
Martens J. ..... [Duva80]
Martin D.F. ..... [Brin85]
Martin D.J. ..... [Mart83]
Martin R.J. ..... [DeMi87a], [DeMi88b]
Martyn J. ..... [Mart70]
Masinter L. ..... [Teit81]
Mathur A.P. ..... [Krau86], [Krau88], [Math86], [Math87a]
[Math87b], [Math88a], [Math88b], [RegoXX]
Matsumoto M.T. ..... [McCa80a], [McCa80b]
Matteoli E. ..... [Baia85]
Mauboussin A. ..... [Choq85]
Mauger C. ..... [Maug85]

Mayer R.E. ..... [Shne77a]
Mayes L. ..... [Maye89]
Mayfield W.T. ..... [Mayf85], [Mayf86]
Mayo K. ..... [Henr88b]
McAllister D.F. ..... [Scot84a], [Scot84b], [Scot87], [Vouk85a]
[Vouk85b], [Vouk85c], [Vouk86a], [Vouk86b]
McCabe T.J. ..... [McCa76], [McCa82a], [McCa82b], [McCa82c]
McCall J.A. ..... [Cava78], [McCa77a], [McCa77b], [McCa78]
[McCa79], [McCa80a], [McCa80b], [McCa84], [McCa87a], [McCa87b]
[Walt78]
McClure C.L. ..... [McCl76], [McCl78a], [McCl78b]
McCluskey G. ..... [Dems82]
McCracken W.M. ..... [DeMi87a], [DeMi87d], [DeMi88a], [Guin89]
McDaniel G. ..... [McDa77]
McGarry F.E. ..... [Agre84b], [Agre84c], [Babs83], [Basi77a]
[Card82], [Card84], [Card85a], [Card85d], [Card87b], [Chur82]
[Cont86], [Cook80], [Gree81], [McGa82], [McGa84], [McGa85a]
[McGa85b], [Page82], [Page84], [Page85], [Vale89]
McGettrick A.D. ..... [Krie86], [McGe82]
McGibbon T.L. ..... [McGi77]
McGregor T. ..... [Yau78]
McIntree J.W. Jr. ..... [McIn83]
McIver W. ..... [Jone89]
McKay D. ..... [Shne77a]
McKelvey N. ..... [McCa87a], [McCa87b]
McLean E.R. ..... [Hare82]
McLean J. ..... [Land86], [MacL82]
McMullin P.R. ..... [Gann80], [Gann81], [Haml79], [McMu80]
[McMu82], [McMu83], [Weis85a]
McTap J.L. ..... [McTaXX]
McWethy S. ..... [McWe84]
Mearns I. ..... [Barr82], [Mear81], [Mear83]
Medina R. ..... [Medi81]
Meeker R.E. ..... [Hall73], [Rama73]
Meeson R.N. ..... [DeMi88b], [Youn89a]
Meisels I. ..... [Crai88b]
Meldal S. ..... [Luck87], [Meld88]
Melliar-Smith P.M. ..... [Crow85a], [Crow85b], [Mell82]
Mellor P. ..... [Bend86]
Melton R.A. ..... [Mill75c]
Mendis K.S. ..... [Mend79]
Menon P.R. ..... [Leve86a]
Merey A. ..... [Snee85]
Merrit M.J. ..... [DeMi81]
Meyer A.R. ..... [Meye67]
Miara R.J. ..... [Miar83]
Michon J.P. ..... [Bene85]
Migneault G.E. ..... [Mign82]
Mili A. ..... [Mili84]
Millen J.K. ..... [Mill81b], [Mill87b]
Miller A.M. ..... [Mill80c]

Miller B.P. ..... [Mill84]
Miller D.M. ..... [Shaw89]
Miller D.P. ..... [Hite88]
Miller D.R. ..... [Mill85], [Mill86]
Miller E.F. ..... [Mill72c], [Mill74a], [Mill74b], [Mill74c]
[Mill74d], [Mill75b], [Mill75c], [Mill75e], [Mill75f], [Mill77a]
[Mill77b], [Mill79a], [Mill79b], [Mill79c], [Mill80a], [Mill81a]
[Paig72], [Uren87]
Miller R. ..... [Dems82]
Miller R.E. ..... [Mill80d]
Milliman P. ..... [Curt79a], [Curt79b], [Shep79]
Mills H.D. ..... [Curr86], [Dyer80], [Dyer81b], [Dyer81c]
[Dyer82b], [Ferr77], [Ling79], [Mill71], [Mill72a], [Mill72b]
[Mill72d], [Mill75a], [Mill75d], [Mill80b], [Mill83], [Mill87a]
Milne P.W. ..... [Luck79b]
Minsky N.H. ..... [Mins83]
Misra J. ..... [Chan79], [Chan88], [Misr81], [Misr82]
Misra P.N. ..... [Misr83]
Mital R. ..... [Koer84]
Mittermeir R.T. ..... [Mitt82]
Miyamoto I. ..... [MiyaXX]
Miyazaki Y. ..... [Miya85], [Miya87]
Mizumo Y. ..... [Mizu83]
Mohanty S.N. ..... [Moha76a], [Moha76b], [Moha79]
Moher T. ..... [Berg82], [Mohe82]
Mok A.K. ..... [Jaha86]
Mok Y.R. ..... [Rama81]
Moller B. ..... [Baue89]
Mollod L.D. ..... [Zeig89]
Monche U. ..... [Krie86]
Moore J.S. ..... [Boye79], [Boye80], [Boye81], [Boye83]
[Boye84a], [Boye84b], [Boye88], [Moor88]
Mora R. ..... [Medi81]
Moran M.L. ..... [Feld89], [Mora85]
Moranda P.B. ..... [Jeli72], [Jeli73], [Mora72], [Mora75]
[Mora78a], [Mora78b], [Mora78c], [Mora80]
Morell L.J. ..... [More81], [More84], [More87], [More88]
Morgan D.E. ..... [Blac81], [Tayl78a], [Tayl80a]
Morgan E.T. ..... [Morg84], [Morg86], [Morg87]
Mori K. ..... [Miya85]
Moriconi M. ..... [Mori83]
Morris J. ..... [McCa87a], [McCa87b]
Morris J.H. ..... [Morr71], [Morr77]
Motley R.W. ..... [Broo80b], [Motl76]
Moulding M.R. ..... [Ande85]
Mukunda R. ..... [Less80]
Munoz C.U. ..... [Bird83], [Muno88]
Munson J.C. ..... [Muns89]
Murakami N. ..... [Miya87]
Murata T. ..... [Mura89]
Musa J.D. ..... [Haml78c], [Iann84], [Musa75], [Musa76]

[Musa77], [Musa79a], [Musa79b], [Musa80a], [Musa80b], [Musa84]
[Musa87], [Musa89]
Musselman J.A. ..... [Miar83]
Musser D.R. ..... [Gutt78a], [Muss79]
Myers B.A. ..... [Myer83]
Myers G.J. ..... [Myer76], [Myer77], [Myer78a], [Myer78b]
[Myer79], [Stev74]
Myers J.P. Jr. ..... [Prat87]
Myhre J.M. ..... [Myhr68]
NBS ..... [FIPS77]
NSWC ..... [Farr83]
Nagel P.M. ..... [Nage82], [Nage84]
Najm E. ..... [Najm87]
Nakagawa Y. ..... [Naka89]
Nance R.E. ..... [Arth88], [Henr85]
National Bureau of Standards ... [Mait80], [NBS74], [NBS82a], [NBS82b]
[Rose85b]
Naur P. ..... [Naur69]
Navarro J.A. ..... [Miar83]
Neff R. ..... [Luck86a], [Sank85]
Neilson A. ..... [Crai88b]
Nejmeh B.A. ..... [Yu84], [Yu88a]
Nelson E.A. ..... [Nels66]
Nelson E.C. ..... [Nels78], [Thay78]
Neumann P.G. ..... [Neum75]
Ng P.H. ..... [Ng78]
Nguyen T.D. ..... [Amor89]
Nguyen V. ..... [Nguy86]
Nicola V.F. ..... [Nico87]
Nikolaou C.N. ..... [Schu81]
Nixon M.R. ..... [Wing89]
Noonan R.E. ..... [Noon75]
Norris M.T. ..... [Pate89]
Ntafos S.C. ..... [Dura81a], [Ntaf79], [Ntaf81a], [Ntaf81b]
[Ntaf82], [Ntaf84], [Ntaf85]
O'Neill D. ..... [Dyer80]
Obaid A. ..... [Bria86]
Offutt A.J. ..... [DeMi87c], [DeMi88a], [Offu87]
Ogden N. ..... [Wint78]
Ohba M. ..... [Ohba84], [Ohba89], [Yama83]
Ohkawa T. ..... [Suno82]
Okada M. ..... [Okad82]
Okroy K. ..... [Ehre76]
Okumoto K. ..... [Goel78], [Goel79], [Goel80a], [Goel81]
[Iann84], [Musa87]
Oldehoeft R.R. ..... [Olde77], [Olde83]
Olender K.M. ..... [Olen86]
Olson M.H. ..... [Ives83]
Ono Y. ..... [Isod87]
Oppen D.C. ..... [Luck79b]
Orr R.A. ..... [Pate89]

Osaki S. ..... [Yama83]
Oskarsson O. ..... [Beck76]
Osterweil L.J. ..... [Fosd76a], [Fosd76b], [Gabo76], [Olen86]
[Oste75a], [Oste75b], [Oste76a], [Oste76b], [Oste77], [Oste80]
[Oste81a], [Oste81b], [Oste83], [Oste84], [Oste86a], [Oste86b]
[Oste87], [Tayl78b], [Tayl80b], [Tayl84], [Tayl86b], [Tayl87]
[Tayl88]
Ostrand T.J. ..... [Ostr78], [Ostr79], [Ostr80], [Ostr84]
[Ostr86], [Ostr88], [Weyu80c]
Ottenstein K.J. ..... [Otte76]
Ottenstein L ..... [Otte81]
Ottenstein L.M. ..... [Otte78], [Otte79]
Ouimet D. ..... [Boch88]
Owe O. ..... [Luck84b], [vonH85]
Owicki S.S. ..... [Nguy86], [Owic75], [Owic76], [Owic82]
Owre S. ..... [Halp87]
Oxman S.W. ..... [Oxma78]
Page G.T. ..... [Card85d], [Card87b], [McGa84], [Page82]
[Page84], [Page85]
Page J. ..... [Card82], [Card84]
Page M.P. ..... [Page74]
Paige M. ..... [Paig81]
Paige M.R. ..... [Mill74a], [Paig72], [Paig75], [Paig77a]
[Paig77b], [Paig78a], [Paig78b]
Pammett K. ..... [Maug85]
Panililo-Yap N.M. ..... [Basi84d], [Basi85e], [Basi85h]
Panzl D.J. ..... [Panz76], [Panz78a], [Panz78b], [Panz78c]
[Panz81a], [Panz81b]
Pariseau R.J. ..... [Gree76], [Pari76]
Park R.D. ..... [Frei79]
Parker D.S. ..... [Gorl87], [LeDo85]
Parker R.A. ..... [Clem84]
Parnas D.L. ..... [Bart77], [Clem84], [Parn72a], [Parn72b]
[Parn72c], [Parn74], [Parn77], [Parn78], [Parn79], [Parn85]
[Parn88]
Parr F.N. ..... [Parr80]
Partsch H. ..... [Baue79b], [Baue89], [Wirs83]
Pase B. ..... [Crai88b], [Pase87a], [Pase87b]
Passafiume J.F. ..... [DeMi87a]
Pate S. ..... [Pate89]
Patnaik D. ..... [Basi86e]
Paulsen L.R. ..... [Shen85]
Pavlin J. ..... [Less80]
Payne C. ..... [Tisc83]
Payton T. ..... [Payt82]
Pearl J. ..... [Pear84]
Pearson S.W. ..... [Bail83]
Pease M. ..... [Lamp82]
Penedo M.H. ..... [Boeh84b]
Pepper P. ..... [Baue79b], [Baue89], [Wirs83]
Perera I.A. ..... [Pere85], [Whit86]

Perkins J.A. ..... [Ande88], [Kell85a], [Kell85b], [Payt82]
[Perk86], [Perk87]
Perlis A.J. ..... [DeMi79a], [Perl81]
Perricone B.T. ..... [Basi82d]
Perry S. ..... [Perr87]
Perry W.E. ..... [Perr83], [Perr86], [Perr88]
Pesch H. ..... [Pesc85]
Peters L.J. ..... [Lamb78]
Peterson J. ..... [Pete77], [Pete81]
Peterson R.J. ..... [Pete76]
Petschenik N.H. ..... [Pets85]
Phelps C.V. ..... [Razo85]
Phillips T.Y. ..... [Basi81g], [Basi83b]
Piatkowski T.F. ..... [Piat80]
Picasso G.O. ..... [Pica81]
Pierce J.L. ..... [Dunh85]
Pikul R.A. ..... [Piku76]
Pimont S. ..... [Pimo75]
Pippenger N. ..... [Pipp78]
Piwowarski P. ..... [Piwo82]
Plauger P.J. ..... [Kern74a], [Kern74b], [Kern81]
Ploedereder E. ..... [Ploe79]
Plogert K. ..... [Ehre78]
Pnueli A. ..... [Mann70]
Pnuenuli A. ..... [Pnue77]
Podgurski A. ..... [Clar85a], [Clar86a]
Polak W. ..... [Luck79b], [Luck80a], [Luck80b], [Luck80c]
[Pola81]
Pooch P.C. ..... [Pooc74]
Poole P.C. ..... [Pool73]
Popkin G.S. ..... [Popk78]
Porter A.A. ..... [Selb89]
Post J.V. ..... [Bowe83], [Press83]
Postak J.N. ..... [Herd79]
Poston R.M. ..... [Post87]
Poutanen O. ..... [Pout87]
Prather R.C. ..... [Prat80]
Prather R.E. ..... [Prat83], [Prat87]
Presson P.E. ..... [Bowe83], [Press83]
Principato R.N. ..... [Prin78]
Probert R.L. ..... [Prob80], [Prob82a], [Prob82b], [Prob82c]
[Prob83], [Prob84], [Ural83], [Ural84]
Proctor N. ..... [Halp87]
Protzel P.W. ..... [Prot88]
Prycker M de ..... [Pryc82]
Purdom P. ..... [Purd72]
Purdue University ..... [SERC87]
Putnam L.H. ..... [Putn77], [Putn78], [Putn79], [Putn82]
Pyster A.B. ..... [Thay80]
Queyras B. ..... [Bess87]
Quinnan R.R. ..... [Dyer80]

Quirk W.J. ..... [Quir85]
RADC ..... [Amor75], [Angu83], [Bagg80], [Bake77]
[Balz81], [Blac77], [Bowe83], [Bowe85], [Broo80b], [Clar86b]
[Goel80a], [Goel82], [Goel83], [Goel88], [Heid82], [Herd79]
[Laem78], [McCa77a], [McCa80a], [McCa80b], [McCa87a], [McCa87b]
[Popk78], [Press83], [RADC76a], [RADC76b], [RADC86], [Scha79]
[Shoo79], [Sief88], [Szul83]
Rabin M.O. ..... [Rabi77]
Radatz J. ..... [McWe84]
Ramamoorthy C.V. ..... [Chen75], [Long77], [Rama73], [Rama74a]
[Rama74b], [Rama75a], [Rama75b], [Rama76], [Rama80], [Rama81]
[Rama82]
Ramsey C.L. ..... [Basi84d], [Basi85h]
Ramsey J. ..... [Basi84a]
Randall W. ..... [McCa87a], [McCa87b]
Randell B. ..... [Ande79b], [Rand75]
Rapps S. ..... [Rapp80]
Rauch G. ..... [Ehre76]
Rault J.-C. ..... [Gira73], [Pimo75], [Raul73]
Rayner D. ..... [Bart80]
Razouk R.R. ..... [Morg86], [Morg87], [Razo85]
Reddy G.R. ..... [Kafu84], [Kafu85b], [Redd84a], [Redd84b]
Redwine S.T. Jr. ..... [Redw83]
Reed K. ..... [Romb88b], [Wu87c]
Reed S. ..... [Less80]
Reeves H.L. ..... [Long77]
Reghizzi S.C. ..... [DiMa85]
Rego V. ..... [Krau88], [RegoXX]
Reich L.E. ..... [Glig87]
Reif J.H. ..... [Reif79c]
Reifer D.J. ..... [Reif75], [Reif78], [Reif79a], [Reif79b]
Reiter R.W. Jr. ..... [Basi77a], [Basi78b], [Basi79b], [Basi79c]
[Basi80a], [Basi81c], [Reit79]
Reynolds R.G. ..... [Reyn86], [Reyn87], [Reyn89]
Rich C. ..... [Rich81e]
Richards P.K. ..... [McCa77a], [McCa77b], [Rich76]
Richardson D.J. ..... [Clar81a], [Clar81c], [Clar82], [Clar83a]
[Clar83b], [Clar84], [Clar85a], [Clar85b], [Clar86a], [Clar86b]
[Clar88a], [Hass80], [Rich78], [Rich81a], [Rich81b], [Rich81c]
[Rich81d], [Rich82], [Rich85a], [Rich85b], [Rich86a], [Rich86b]
[Rich87a], [Rich88]
Richier J.L. ..... [Fern85], [Rich87b]
Riddell I.J. ..... [Henn81], [Henn84], [Ridd80], [Wu87b]
[Wu88]
Riddle W.E. ..... [Avru86], [Bris79], [Ridd78], [Ridd79]
[Wile83]
Ritchies D.M. ..... [Meye67]
Roach M.G. ..... [Roac80]
Roberts D. ..... [Reyn86]
Roberts N.H. ..... [Vese81]
Robinson L. ..... [Neum75], [Robi77], [Robi79], [Roub77]

[Silv79]
Roby C.G. ..... [Roby85]
Rodeheffer T.L. ..... [Hibb82]
Roe R.P. ..... [Roe87]
Roever W.P. de ..... [Apt80], [Fran79], [Gert84]
Rogers W.J. ..... [Grov80]
Roggio R.F. ..... [Rogg80]
Rohleder M.G. ..... [Babs83]
Rolandelli C. ..... [Rola86]
Romain Y. ..... [Troy86]
Rombach H.D. ..... [Basi86b], [Basi87a], [Basi87b], [Basi88]
[Katz86], [Romb84], [Romb85a], [Romb85b], [Romb87a], [Romb87b]
[Romb87c], [Romb88a], [Romb88b], [Romb88c], [Romb88d], [Romb88e]
[Romb88f], [Romb88g], [Romb88h], [Romb89a], [Romb89b]
Rome Air Development Center .... [McGi77], [Motl76], [Slav75], [Thay76]
Roquet J.C. ..... [Roqu86]
Rosen B. ..... [Rose75]
Rosenblum D.S. ..... [Luck86a], [Rose84], [Rose85a], [Sank85]
[Sank86]
Rosenthal L.S. ..... [Rose85b]
Rosner A.J. ..... [Szul83]
Rosson C.V. ..... [Ross88]
Roubine O. ..... [Robi77], [Roub77]
Rowan S. ..... [Payt82]
Rowland J.H. ..... [Roe87], [Rowl81a], [Rowl81b], [Rowl88]
Rubey R.J. ..... [Rube75]
Rubin F. ..... [Cail79]
Rubin J. ..... [Rubi82]
Rudin H. ..... [Zafi80]
Rugaber S. ..... [Guin89]
Rumsey J.R. ..... [Rums77]
Rupolo V.F. ..... [Wien84]
Rushby J. ..... [Crow85a], [Crow85b], [Rush84]
Russell W.E. ..... [Herd79]
Russinoff D.M. ..... [Russ83]
Rustin R. ..... [Rust71]
Ruston H. ..... [Shoo79]
Ryan J.P. ..... [Hodg76]
SAMSO ..... [SAMS77]
SEL ..... [Agre84c], [Agre86], [Agre87], [Babs83]
[Basi77a], [Card82], [Card84], [Card85a], [Card85c], [Card86b]
[Chur82], [Cook80], [Cook81], [Deck82a], [Deck82b], [Godf87]
[Gree81], [Hell87], [Lo83], [McGa84], [McGa85a], [Mill80c]
[NASA81], [Page82], [Page85], [Perr87], [Pica81], [SEL82]
STARS Program ..... [STAR85]
SYSCON Corporation ..... [SYSC83]
Saaltink M. ..... [Crai88b], [Pase87a], [Smit88]
Sabnani K.K. ..... [Sabn85]
Sackman H. ..... [Sack68]
Saglietti F. ..... [Sagl86]
Sahay P.N. ..... [Whit85]

Sahner R.A. ..... [Sahn87]
Salasin J. ..... [Ardo88], [Besh85], [Bryk89], [Kapp88]
[Linn88], [Youn89a]
Sale A.H.J. ..... [Wich79]
Salt N. ..... [Salt82]
Salvador J.P. ..... [Taus87a]
Samet H. ..... [Same76]
Sampson W.A. ..... [Brow73b]
San Antonio R.C. ..... [SanA83]
Sandewall E. ..... [Beck76]
Sanella D. ..... [Sane83]
Sankar S. ..... [Sank85], [Sank86]
Sarikaya B. ..... [Barb88], [Forg87], [Sari82], [Sari84a]
[Sari84b], [Sari87], [Sari88a], [Sari88b], [Sari88c]
Sarkar D. ..... [Sark89]
Sarkar S.C. de ..... [Sark89]
Satterthwaite E.H. ..... [Satt72], [Satt75]
Sauder R.L. ..... [Saud62]
Saunders S.C. ..... [Myhr68]
Saxema A.R. ..... [Neum75]
Saxena A.R. ..... [Saxe77]
Sayler J. ..... [Ridd78], [Wile83]
Sayward F.G. ..... [Acre79], [Bald79], [Budd77], [Budd78a]
[Budd80a], [Budd80c], [DeMi78], [DeMi79b], [Lipt78], [Perl81]
Schach S.R. ..... [Getz83], [Wahl86], [Wahl88]
Schaffer R.E. ..... [Angu80], [Scha79]
Schaller H. ..... [Pesc85]
Schaufler R. ..... [Tisc83]
Scheid J. ..... [Loca80], [Sche85]
Scherlis W.L. ..... [Luck79b]
Schick G.J. ..... [Schi73], [Schi78]
Schiffenbauer R.D. ..... [Schi81]
Schimmelpfenneg C.L. ..... [Farr88]
Schmidt R.L. ..... [Bowe83], [Press83]
Schneider F.B. ..... [Lamp84]
Schneider G.M. ..... [Mohe82]
Schneider V. ..... [Schn78]
Schneidewind N.F. ..... [Brad75], [Gree76], [Schn75], [Schn76]
[Schn77a], [Schn77b], [Schn77c], [Schn79a], [Schn79b], [Sing86]
Schnupp P. ..... [Pesc85]
Scholz F.W. ..... [Nage84]
Schouwen A.J. van ..... [Parn88]
Schroeder A. ..... [Schr84]
Schuller H. ..... [Ehre73]
Schuman S.A. ..... [Schu81]
Schutts D. ..... [Schu77]
Schwartz J.T. ..... [Schw70a]
Schwartz R.L. ..... [Crow85a], [Crow85b], [Mell82]
Scott R.K. ..... [Scot83a], [Scot83b], [Scot84a], [Scot84b]
[Scot87], [Vouk85b]
Sedlmeyer R.L. ..... [Kear85], [Kear86], [Sedl83]

Seewaldt T. ..... [Boeh84a]
Segal A. ..... [Ridd78]
Selby R.W. Jr. ..... [Basi83b], [Basi84b], [Basi85a], [Basi85b]
[Basi85f], [Basi86a], [Basi86b], [Card85a], [Romb85b], [Selb83]
[Selb84], [Selb85], [Selb86], [Selb87a], [Selb87b], [Selb87c]
[Selb87d], [Selb88a], [Selb88b], [Selb88c], [Selb89], [Tayl87]
[Tayl88]
Senn J.A. ..... [Gibs89]
Senn R. ..... [McCa87a], [McCa87b]
Serre J.-M. ..... [Boch87b]
Sethi R. ..... [Aho86]
Shaikh M.U. ..... [HennXX]
Shankar A.U. ..... [Lam84]
Shankar K.S. ..... [Shan82]
Shankar N. ..... [Shan87]
Shanthikumar J.G. ..... [Shan80], [Shan81]
Shapiro D.S. ..... [Shap81]
Shar M. ..... [Perl81]
Shatz S.M. ..... [Mura89], [Shat88]
Shaw A.C. ..... [Shaw78]
Shaw M. ..... [Brow80b], [Shaw80], [Wulf76]
Shaw W.H. ..... [Shaw89]
Sheil B.A. ..... [Shei81]
Shen V.Y. ..... [Cont81], [Shen80], [Shen83], [Shen85]
[Theb84], [Wood81a], [Wood81b], [Wood81c], [Yu88a], [Yu88b]
Shenker B. ..... [Mura89]
Sheppard J. ..... [Lass81]
Sheppard S.B. ..... [Basi82b], [Curt79a], [Curt79b], [Shep77]
[Shep78], [Shep79], [Shep81]
Shih C. ..... [Basi84d]
Shimeall T.J. ..... [Cha87], [Cha88], [Leve83a], [Rola86]
[Shim88]
Shimomura T. ..... [Isod87]
Shneiderman B. ..... [Miar83], [Shne75], [Shne77a], [Shne77b]
[Shne77c], [Shne80]
Sholl H.A. ..... [Shol75]
Shooman M.L. ..... [Bagg78], [Bagg80], [Laem78], [Popk78]
[Shoo72], [Shoo73], [Shoo74], [Shoo75], [Shoo76], [Shoo77a]
[Shoo77b], [Shoo77c], [Shoo79], [Shoo83], [Shoo86]
Shorre V. ..... [Loca80]
Shostak R. ..... [Crow85a], [Crow85b], [Lamp82]
Shriver B.D. ..... [Land79]
Sidhu D.P. ..... [Sidh89]
Siebert A.E. ..... [Good82b]
Siefert P.T. ..... [Sief88]
Sigal R. ..... [Ostr86]
Signoret J.P. ..... [Bene85]
Sikaczowski R.O. ..... [Sika88]
Silberschatz A. ..... [Kieb83]
Silverburg B.A. ..... [Robi79], [Silv79]
Simmons D.B. ..... [Zoln77], [Zoln81]

Singh R. ..... [Sing86]
Singpurwalla N.D. ..... [Form77], [Form79]
Sistla A.P. ..... [Clar86d], [Sist88]
Sites S.L. ..... [Site74]
Siyan K.S. ..... [Siya80]
Skillicorn D.B. ..... [Skil89]
Skrivan J.A. ..... [Nage82], [Nage84]
Skrukrud A.M. ..... [Stan77]
Skuce D.R. ..... [Prob83]
Slavinski R.T. ..... [Slav75]
Slind K. ..... [Joyc87b]
Slivinski T. ..... [Sliv84]
Smith C.H. ..... [Angl83], [Cher86], [Cher87b]
Smith C.P. ..... [Chri81], [Smit79], [Smit80a], [Smit80b]
Smith E.J. ..... [Deck82b]
Smith G.L. ..... [Lamb78]
Smith H. ..... [Drap66]
Smith L.M. ..... [Good82b]
Smith M.K. ..... [Good84b], [Smit88]
Smith O.D. ..... [Farr88]
Smith R.W. ..... [Krau73]
Smith T. ..... [Misr82]
Sneed H.M. ..... [Majo83], [Snee78], [Snee84], [Snee85]
[Snee86]
So H.H. ..... [Long77]
Sodano N.M. ..... [Szul83], [Szul84]
Sofer A. ..... [Mill85], [Mill86]
Software Engineering Research Center ..... [SERC87]
Solderitsch J.J. ..... [Sold89]
Solis D.M. ..... [Soli83], [Soli85]
Sollo G. ..... [Brin87]
Solomon M.H. ..... [Fink83]
Soloway E. ..... [John84], [JohnXX], [Solo83], [Solo84]
Soneriu M.D. ..... [Sone80], [Sone81]
Soong N.L. ..... [Soon77]
Soppe M. ..... [Sopp86]
Sorkowitz A.R. ..... [Sork79]
Spadafora I. ..... [Bazz82]
Spafford E.H. ..... [DeMi86b], [DeMi87b]
Spirk A.P. ..... [Pesc85]
Spitzen J.M. ..... [Wegb76]
Srinia V.P. ..... [Srin85]
St. Jean L.D. ..... [StJe85]
Stanat D.F. ..... [Stan84b]
Stanculescu A. ..... [Luck86b], [Luck86c]
Standish T.A. ..... [Stan83], [Stan84a], [Tayl82a], [Tayl85]
Stanfield J.R. ..... [Stan77]
Stankovic J.A. ..... [Stan80]
Statman R. ..... [Cher88]
Stavely A. ..... [Ridd78]
Steenberger C. ..... [Brin87], [Stee86]

Stefanini S. ..... [Baia85]
Stepczyk F.M. ..... [Step74]
Stetter F. ..... [Stet86]
Stevens K.T. ..... [Arth88]
Stevens W.P. ..... [Stev74]
Steveson D.R. ..... [Luck81]
Stewart K.R. ..... [Herd79]
Stickney M.E. ..... [Stic78]
Stigall P.D. ..... [Stig74]
Stolzy J.L. ..... [Leve83a], [Leve83c], [Leve87], [Stol84]
Stotts D. ..... [Romb88b]
Stoy J. ..... [Stoy77]
Straker E.A. ..... [Long77]
Strategic Defense Initiative Organization ... [SDI087], [SDI088a], [SDI088b]
Stucki L.G. ..... [Howd74a], [Howd76b], [Stuc72], [Stuc73]
[Stuc74], [Stuc75a], [Stuc75b], [Stuc77]
Stuckle E.D. ..... [Boeh84b]
Sturm W.A. ..... [Hech77c]
Suk D.S. ..... [Sone80]
Sukert A.N. ..... [Angu80], [Suke77a], [Suke77b], [Suke79]
Sullivan J.E. ..... [Bell74], [Sull75]
Summerill L.F. ..... [Dela88]
Sunohara T. ..... [Suno82]
Sunshine C.A. ..... [Boch80], [Suns77], [Suns82]
Sutherland D. ..... [Kore84]
Suzuki K.S. ..... [Kato86], [Rama81]
Suzuki N. ..... [Flon78a], [Flon78b], [Flon81], [Luck79a]
Svanaes D. ..... [Conr85]
Svegel N.P. ..... [Stuc74]
Svobodova L. ..... [Svob76]
Swartout W. ..... [Cohe82]
Swearingen D. ..... [Duva80]
Symons C.R. ..... [Symo88]
Szulewski P.A. ..... [Szul80], [Szul81], [Szul83], [Szul84]
[Whit80]
TRW ..... [Nels73]
Tai K.C. ..... [Carv88], [Tai79], [Tai80], [Tai85a]
[Tai85b], [Tai85c], [Tai86], [Vouk85a], [Vouk85c], [Vouk86a]
[Vouk86b]
Takahashi M. ..... [Taka89]
Takano A. ..... [Suno82]
Takayanagi A. ..... [Itak82]
Tamboli A. ..... [Lee89a]
Tanaka A. ..... [Tana81]
Tardo J.J. ..... [Gogu79a], [Gogu79b]
Tarr P. ..... [Wile38]
Tasar O. ..... [Stig74]
Tate G. ..... [Vern89]
Tauson-Conte H.J. ..... [Taus87a], [Taus87b], [Taus88]
Tausworthe R.C. ..... [Taus81], [Taus82]
Taylor D.J. ..... [Blac81], [Tayl77a], [Tayl78a], [Tayl80a]

Taylor J.R. ..... [Tayl77b]
Taylor R. ..... [Haml88]
Taylor R.N. ..... [Brin85], [Stan84a], [Tayl78b], [Tayl80b]
[Tayl80c], [Tayl81], [Tayl82b], [Tayl82c], [Tayl83a], [Tayl83b]
[Tayl83c], [Tayl84], [Tayl85], [Tayl86a], [Tayl86b], [Tayl87]
[Tayl88], [Youn86a], [Youn88b], [Youn89c]
Taylor T. ..... [Tayl82a]
Taylor W.A. ..... [Deck82a], [Deck82b]
Teichroew D. ..... [Teic74], [Teic77]
Teitelman W. ..... [Teit81], [Teit84]
Teng F.C. ..... [Whit78a]
Terrien D. ..... [Maye89]
Thalhamer J.A. ..... [Sold89]
Thatcher J.W. ..... [Gogu78]
Thayer R.H. ..... [Thay80]
Thayer T.A. ..... [Lipo77], [Thay75], [Thay76], [Thay78]
Thebaut S.M. ..... [Shen85], [Theb83], [Theb84]
Thibodeau R. ..... [Thib78], [Thib81]
Thoman T.A. ..... [Stan84b]
Thomas J. ..... [Leve83a], [Thom83]
Thompson D. ..... [Thom81]
Thompson M.C. ..... [Rich86a], [Rich86b], [Rich87a], [Rich88]
Thompson W.B. ..... [Kear85], [Kear86], [Sedl83]
Thompson W.E. ..... [Thom80]
Tichy W.F. ..... [Tich79], [Tich86]
Tisato F. ..... [Mand85]
Tischler R. ..... [Tisc83]
Townley J.A. ..... [Chea79]
Trattner S. ..... [Hech77c], [Reif79b]
Trauboth H. ..... [Geig79]
Traverse P.J. ..... [Roqu86]
Triolet R. ..... [Trio86]
Tripp L.L. ..... [Carp75]
Trivedi K.S. ..... [Nico87], [Sahn87], [Triv80]
Trost J. ..... [Maye89]
Troup D.B. ..... [Youn88b]
Troy D.A. ..... [Troy81]
Troy R. ..... [Troy86]
Truszkowski W.F. ..... [Basi77a]
Tsai J.T. ..... [Bowe83], [Bowe85], [Press83]
Tsalalikhin L. ..... [Tsal86]
Tucker A.E. ..... [Tuck65]
Turano Y. ..... [Kato86]
Turinin F. ..... [Manc83]
Turner A.J. ..... [Basi75]
Turner C. ..... [Turn81a], [Turn81b]
Turner J. ..... [Hall73], [Rama73], [Turn80]
US Army Computer Systems Command ..... [Army84]
USAF Avionics Laboratory ..... [SYSC83]
Uehara K. ..... [Suno82]
Ulery T. ..... [Romb89a]

Ullman J.D. ..... [Aho86], [Hech72], [Hech75], [Ullm73]
Ullman R. ..... [Dunn82]
Underhill L.H. ..... [Unde63]
Ungar B. ..... [Joyc87b]
University of Texas ..... [EPI82]
Ural H. ..... [Prob82b], [Prob83], [Ural83], [Ural84]
Urban J.E. ..... [Belk86]
Urban R.J. ..... [Urba73]
Uren E. ..... [Uren87]
Utsunomiya E. ..... [Waka89]
Uyar M.U. ..... [Uyar86]
Vaglini G. ..... [Baia85], [DeFr85]
Vairavan K. ..... [Lind89]
Valdes P.M. ..... [Vald83]
Valett J.D. ..... [McGa85b], [Vale89]
Valiant L.G. ..... [Vali84]
Vaneschi M. ..... [Baia84]
Vemuri V. ..... [Vemu80]
Verner J.M. ..... [Vern89]
Verrall J.L. ..... [Litt73]
Vesely W.E. ..... [Vese81]
Vessey L. ..... [Vess83]
Vickery B.C. ..... [Mart70]
Vienneau R. ..... [McCa87a], [McCa87b]
Vlack R.K. ..... [Blac77]
Voges U. ..... [Geig79], [Gmei79], [Voge80]
Voiron J. ..... [Fern85]
Vosburgh J. ..... [Vosb84]
Vose M.D. ..... [Vose88]
Vouk M.A. ..... [Vouk85a], [Vouk85b], [Vouk85c], [Vouk86a]
[Vouk86b]
Wagner E.G. ..... [Gogu78]
Wahl N.J. ..... [Wahl86], [Wahl88]
Wakahara Y. ..... [Waka89]
Wake S. ..... [Henr88a], [Wake88]
Waksman A. ..... [Elsp72a]
Waldinger R.J. ..... [Elsp72a], [Elsp72b], [Elsp73], [Mann78]
Walker M. ..... [Walk81]
Wallace D.R. ..... [Wall89]
Wallnau K.C. ..... [Sold89]
Walsh D.A. ..... [Wals77c]
Walsh P.J. ..... [Wals85]
Walsh T.J. ..... [Wals79]
Walston C.E. ..... [Wals77a], [Wals77b]
Walters G.F. ..... [McCa77a], [McCa77b], [Walt78], [Walt79]
Wampler G.K. ..... [Wamp85]
Wand M. ..... [Wand79]
Wang A.S. ..... [Wang83], [Wang84]
Warner D.C. ..... [Warn72]
Warren S. ..... [Warr82]
Washington D.A. ..... [Chea78]

Waters R.C. ..... [Rich81e], [Wate79]
Weber J.C. ..... [Webe83]
Weber R. ..... [Vess83]
Wegbreit B. ..... [Morr77], [Wegb74], [Wegb75], [Wegb76]
[Wegb77]
Wegner P. ..... [Wegn79]
Weiderman N.H. ..... [Weid86]
Weinberg G.M. ..... [Wein71]
Weinberger E. ..... [Wein80]
Weiser M.D. ..... [Weis84], [Weis85a]
Weisgerber B. ..... [Krie86]
Weiss D.M. ..... [Basi77a], [Basi81b], [Basi81e], [Basi82a]
[Basi82c], [Frye81], [Weis78], [Weis81], [Weis82], [Weis85c]
Weiss S.N. ..... [Fran85a], [Weis85b], [Weis86], [Weis87]
[Weis88a], [Weis88b], [Weis88c]
Weissman L.M. ..... [Weis74]
Welke S.R. ..... [Mayf85]
Welsh H.O. ..... [Wels83]
West C.H. ..... [Rubi82], [Zafi80]
Wexelblat A. ..... [Wexe87]
Weyuker E.J. ..... [Davi81], [Davi83a], [Davi83b], [Fran85a]
[Fran85b], [Fran86], [Fran88], [Ostr78], [Ostr79], [Ostr80]
[Ostr84], [Ostr86], [Rapp80], [Weis85b], [Weis86], [Weis88b]
[Weyu79], [Weyu80a], [Weyu80b], [Weyu80c], [Weyu81], [Weyu82]
[Weyu83], [Weyu84a], [Weyu84b], [Weyu88], [Weyu89]
White J.R. ..... [GilkXX]
White L.J. ..... [Leun88], [Pere85], [Whit78a], [Whit78b]
[Whit81], [Whit85], [Whit86], [Whit88a], [Whit88b], [Zeil81b]
Whitworth M.H. ..... [Szul80], [Szul81], [Whit80]
Wichman B.A. ..... [Wich79]
Wiecek C.A. ..... [Boot80]
Wiener-Ehrlich W.K. ..... [Wien84]
Wiggs J.E. ..... [Scot84a], [Scot84b], [Wigg84]
Wigle G.B. ..... [Bowe85]
Wild C. ..... [Wild87], [Wild88]
Wile D. ..... [Balz82]
Wileden J.C. ..... [Avru83], [Avru85], [Avru86], [Bate81]
[Bate82], [Bate83a], [Bate83b], [Clar86c], [Dill85], [Dill86]
[Dill88c], [Less80], [Ridd78], [Tayl86b], [Tayl87], [Tayl88]
[Wile83], [Wile84], [Wile88], [Wolf85a], [Wolf85c], [Wolf86a]
[Wolf86c]
Wileden W. ..... [Wile80]
Wilhelm R. ..... [Krie86]
Williams G. ..... [Will79]
Williams R. ..... [Will89]
Williamson N. ..... [Gogu79b]
Wilson W.F. ..... [Halp87]
Winchester J.W. ..... [Yin78], [Yin79]
Wing J.M. ..... [Gutt85], [Wing89]
Winner R.I. ..... [Wahl86]
Winters D. ..... [Wint78]

Winterstein G. ..... [Krie86]
Wiorkowski J.J. ..... [Dura78], [Dura80], [Dura81b]
Wirsing M. ..... [Sane83], [Wirs83]
Wisehart W.R. ..... [Mill74a]
Wiszniewski B. ..... [Whit88a], [Wisz87]
Witt B.I. ..... [Ling79]
Wojcik R.T. ..... [Piku76]
Wolf A.L. ..... [Clar86c], [Tayl87], [Tayl88], [Wile88]
[Wolf85a], [Wolf85b], [Wolf85c], [Wolf86a], [Wolf86b], [Wolf86c]
Wolverton R.W. ..... [Putn77], [Schi73], [Schi78], [Vosb84]
[Wolv74]
Wong P. ..... [ElspXX]
Wood R.C. ..... [Thay80]
Wood V.L. ..... [Lind88b]
Woodfield S.N. ..... [Wood79b], [Wood80a], [Wood81a], [Wood81b]
[Wood81c]
Woods J.L. ..... [Wood78], [Wood80c]
Woodward M.R. ..... [Girg85], [Girg86a], [Henn76a], [Henn76b]
[Henn79], [Henn81], [Ridd80], [Wood77], [Wood79a], [Wood80b]
[Wood88]
Wossner H. ..... [Baue79b]
Wu D. ..... [Wu87a], [Wu87b], [Wu88]
Wu L. ..... [Basi86d], [Wu87c]
Wulf W.A. ..... [Wulf76]
Wyckoff D. ..... [Lo83]
Yam S. ..... [Chan84]
Yamada S. ..... [Yama83]
Yang L. ..... [Chan89]
Yap C.K. ..... [Mill80d]
Yates P. ..... [McCa87a], [McCa87b]
Yau S.S. ..... [Yau78], [Yau79], [Yau80]
Yaung A. ..... [Brun86]
Yeh R.T. ..... [Chen83], [Ches77], [Yeh77], [Yeh79]
Yelowitz L. ..... [Gerh76a], [Gerh76b]
Yerneni A. ..... [Henr88b]
Yin B.H. ..... [Yin78], [Yin79], [Yin80]
Yonezawa A. ..... [Hewi76]
Young B. ..... [Youn86b]
Young G. ..... [Ng78]
Young M. ..... [Tayl86b], [Tayl87], [Youn86a], [Youn88a]
[Youn88b], [Youn89c]
Young S. ..... [Tayl88]
Young W.D. ..... [Kauf87a], [Kauf87b], [Youn86c]
Youngblut C. ..... [Bryk89], [Youn89a], [Youn89b]
Youngs E.A. ..... [Youn71], [Youn74]
Yount L.J. ..... [Youn85]
Yourdon E. ..... [Your76]
Yu T.J. ..... [Slen85], [Yu84], [Yu85], [Yu88a]
[Yu88b]
Zafiropulo P. ..... [Zafi80]
Zagorski H.J. ..... [Farr65]

Zamfir M. ..... [Gogu79b]
Zeigler J. ..... [Zeig89]
Zeil S.J. ..... [Clar85a], [Clar86a], [Clar86b], [Clar88a]
[Clar88b], [Epp86], [Zeil81a], [Zeil81b], [Zeil83a], [Zeil83b]
[Zeil84], [Zeil86], [Zeil87], [Zeil88a], [Zeil88b], [Zeil88c]
[Zeil89]
Zelkowitz M.V. ..... [Basi77a], [Basi77b], [Basi78a], [Basi78c]
[Basi79a], [Basi82b], [Chen81], [Zelk77], [Zelk78], [Zelk79]
[Zelk82]
Zhao J.-R. ..... [Boch87a], [Boch88], [Zhao86]
Zhou C.C. ..... [Zhou81]
Zicari R. ..... [Mand85]
Zilles S.N. ..... [Lisk75], [Zill74]
Zolnowski J.M.C. ..... [Zoln76], [Zoln77], [Zoln81]
Zwanzig K. ..... [Zwan84]
Zweben S.H. ..... [Bake79b], [Bake80], [Hale82], [Hall86]
[Lohs84], [Troy81], [Zweb79], [Zweb89]
van Horn E.C. ..... [vanH68]
van Tassel D. ..... [vanT74]
von Henke F.W. ..... [Crow85a], [Crow85b], [Luck79b], [Luck81]
[Luck84a], [Luck84b], [vonH85]

4. ABSTRACTS

[AFOT86] Abbreviated Introduction: This pamphlet is a guide for the Air Force Operational Test and Evaluation Center (AFOTEC) Software Evaluation Manager (SEM) and Deputy for Software Evaluation (DSE). It describes the numerous activities associated with planning, conducting, analyzing, and reporting software operational test and evaluation (OT&E) assessments.

[AFOT87] Abbreviated Introduction: The purpose of this document is to provide the software evaluator the information needed to conduct the Air Force Operational Test and Evaluation Center's (AFOTEC's) software maintainability evaluation process. In this document software maintainability is limited in scope to software design and documentation assessments.

[AFOT88a] Abbreviated Introduction: This document describes the method and procedures used by the Air Force Operational Test and Evaluation Center (AFOTEC) for evaluating the software support resources (SSR) for mission critical computer resources (MCCR) supportability.

[AFOT88b] Abbreviated Introduction: The purpose of this pamphlet is to provide the software evaluation manager and the deputy for software evaluation information needed to evaluate mission critical computer software life cycle processes as they influence software supportability. In this pamphlet are the means to track the processes affecting mission critical computer software supportability, beginning as early as necessary to provide insight into the quality of the evolving software products, software support resources, and operational support life cycle procedures themselves.

[AFSC86a] Abbreviated Introduction: This pamphlet describes management indicators that will provide visibility into the acquisition of mission-critical computer resources. It is intended to help program managers by presenting software management indicators that reflect the status of software development in an acquisition program. It also provides information that reflects experience on previous acquisition projects. Indicators are just that: indicators. They do not, nor are they intended to, replace sound management practices and communications. Indicators, properly applied, thoroughly understood, and meticulously followed-up, will lead the contractor and program office to those areas requiring management attention.

[AFSC86b] Abbreviated Introduction: This pamphlet describes indicators that will provide insight into the quality of mission-critical computer resources. It is intended to help program managers by presenting indicators that reflect the quality of the software products developed in an acquisition program. It also provides information that reflects experience gained on previous acquisition programs. Indicators are just that: indicators. They do not, nor are intended to, replace sound quality practices. These indicators, properly applied and meticulously followed-up, will lead the contractor and program office to those areas requiring additional quality attention.

[AFSC88a] Overview: The purpose of this pamphlet is to help Program Directors (PD) develop an IV&V program that meets their system's specific requirements. The pamphlet describes a six step procedure for determining the need for a software IV&V effort, establishing its scope, identifying tasks and subtasks associated with each IV&V requirement, selecting a qualified contractor, and estimating software IV&V costs. In addition, this pamphlet integrates the software engineering tasks of D0D-STD-2167A with the software IV&V tasks to ensure value is added to the software development process and product. The methods used in this pamphlet are based on a MIL-STD-882 (System Safety Program Requirements) approach as well as a composite of similar initiatives from Space Division (SD), Aeronautical Systems Division (ASD), and Electronic Systems Division (ESD).

[AFSC88b] Abbreviated Overview: This pamphlet describes software risk abatement processes composed of risk identification, analysis, and handling techniques that can significantly contribute to improving the acquisition of mission-critical computer resources. It is intended to help program directors by integrating software risk

abatement with system-level risk handling techniques. Risk abatement techniques can help the contractor and program office to improve the performance and support of the software in weapon systems.

[Abde86] Abstract: Different software reliability models can produce very different answers when called upon to predict future reliability in a reliability growth context. Users need to know which, if any, of the competing predictions are trustworthy. Some techniques are presented which form the basis of a partial solution to this problem. Rather than attempting to decide which model is generally best, the approach adopted here allows a user to decide upon the most appropriate model for each application.

[Acre80] Abbreviated Introduction: Program testing has been practiced as long as has programming itself, in spite of the general confession that testing can never prove in any absolute sense that a program is correct. Two facts are responsible for the popularity of testing. The first is that testing has a tendency to uncover program errors, and that the more systematic the testing, the stronger this tendency. The second is that a program that is not completely correct is not necessarily unreliable in a given operating environment, and that even a program that is not completely reliable will usually not be completely worthless to its users. Those responsible for software system development are charged with deciding how much they are willing to pay for a given increase in reliability. The challenge for research is therefore to produce a testing method that is (1) more effective at uncovering errors and (2) less expensive to apply. Mutation analysis has been put forward as such a method. Working mutation systems have demonstrated that mutation analysis can be performed at an attractive cost on realistic programs. In this work, the effectiveness of the method is studied by experiments with:

1. System requirements definition
2. System functional specifications
3. Software requirements definition
4. Software functional specifications
5. Software implementation

The mutation analysis methodology examined in this work has as its goal validation of the last stage, software implementation. As such it overlaps some proposed validation methods, and complements others. The following sections outline some of these techniques.

[Adam80] Abstract: An effort to automated the debugging of real programs is presented. We discuss possible choices in conceiving a debugging system. In order to detect all the semantic errors, it must have knowledge of what the program is intended to achieve. Strategies and results are very dependent on the way of giving this knowledge. In the LAURA system that we have designed, the program's task is given by means of a "program model." Automatic debugging is then viewed as a comparison of programs. The main characteristics of LAURA are the representation of programs by graphs, which gets rid of many syntactical variations, the use of program transformations, realized on the graphs, and its heuristic strategy to identify step by step elements of the graphs. It has been tested with about a hundred programs written by students to solve eight different problems in various fields. It is able to recognize correct programs even if their structures are very different from the structure of the program model. It is also able to express exact diagnostics of errors, or at least to localize them. It could be an effective tool for student programmers.

[Adri82] Abstract: Software quality is achieved through the application of development techniques and the use of verification procedures throughout the development process. Careful consideration of specific quality attributes and validation requirements leads to the selection of a balanced collection of review, analysis, and testing techniques for use throughout the life cycle. This paper surveys current validation, verification, and testing approaches and discusses their strengths, weaknesses, and life cycle usage. In conjunction with these, the paper describes automated tools used to implement validation, verification, and testing. In the discussion of new research thrusts, emphasis is given to the continued need to develop a stronger theoretical basis for testing and the need to employ combinations of tools and techniques that may vary over each application.

[Albe76] Abstract: This paper presents an examination into the economics of software quality assurance. An

analysis of the software life cycle is performed to determine where in the cycle the application of quality assurance techniques would be most beneficial. The number and types of errors occurring at various phases of the software life-cycle are estimated. A variety of approaches in increasing software quality (including Structured Programming, Top Down Design, Programmer Management Techniques and Automated Tools) are reviewed and their potential impact on quality and costs are examined.

[Albr83] Abstract: One of the most important problems faced by software developers and users is the prediction of the size of a programming system and its development effort. As an alternative to "size," one might deal with a measure of the "function" that the software is to perform. Albrecht has developed a methodology to estimate the amount of the "function" the software is to perform, in terms of the data it is to use (absorb) and to generate (produce). The "function" is quantified as "function points," essentially, a weighted sum of the number of "inputs," "outputs," "master files," and "inquiries" provided to, or generated by, the software. This paper demonstrates the equivalence between Albrecht's external input/output data flow representative of a program (the "function points" metric) and Halstead's "software science" or "software linguistics" model of a program as well as the "soft content" variation of Halstead's model suggested by Gaffney.

Further, the degrees of correlation between "function points" and the eventual "SLOC" (source lines of code) of the program, and between "function points" and the work-effort required to develop the code, is demonstrated. The "function point" measure is thought to be more useful than "SLOC" as a prediction of work effort because "function points" are relatively easily estimated from a statement of basic requirements for a program early in the development cycle.

The strong degree of equivalency between "function points" and "SLOC" shown in the paper suggests a two-step work-effort validation procedure, first using "function points" to estimate "SLOC," and then using "SLOC" to estimate the work-effort. This approach would provide validation of application development work plans and work-effort estimates early in the development cycle. The approach would also more effectively use the existing base of knowledge on producing "SLOC" until a similar base is developed for "function points."

The paper assumes that the reader is familiar with the fundamental theory of "software science" measurements and the practice of validating estimates of work-effort to design and implement software applications (programs). If not, a review of [cited references] is suggested.

[Alle74] Abstract: The data relationships which exist between the procedures in a program are of interest in program analysis and optimization. In this paper an analysis algorithm is given which determines the interprocedural data flow relationships which exist in a collection of procedures. The context of the analysis is a static (compile time) analysis of procedures within a high level language. It assumes that the collection obeys certain constraints, the most serious of which is that the procedures cannot be recursive. While many practical considerations are not addressed in this paper, the basic practical constraint that each procedure in the collection be analyzed only once is satisfied. Existing results in intraprocedural data flow analysis form the basis for the algorithm.

[Alle76] Abstract: The global data relationships in a program can be exposed and codified by the static analysis methods described in this paper. A procedure is given which determines all the definitions which can possibly "reach" each node of the control flow graph of the program and all the definitions that are "live" on each edge of the graph. The procedure uses an "interval" ordered edge listing data structure and handles reducible and irreducible graphs indistinguishably.

[Ambl76a] Abstract: An introduction to the Gypsy programming and specification language is given. Gypsy is a high-level programming language with facilities for general programming and also for systems programming that is oriented toward communications processing. This includes facilities for concurrent processes and process synchronization. Gypsy also contains facilities for detecting and processing errors that are due to the actual running of the program in an imperfect environment. The specification facilities give a precise way of expressing the desired properties of the Gypsy programs. All of the features of Gypsy are fully verifiable, either by formal proof or by validation at run time. An overview of the language design and a detailed example program are given.

[Amor75] Abstract: This report presents preliminary results of a study in the area of error classification. A general method of error classification is described which is designed to serve as a guideline for experiment-specific applications. A survey of error classification and analysis work, both in the general literature and at MITRE, as well as a study of error experiment design considerations, are reflected in the discussion and conclusions.

[Amst76] Abstract: The purpose of the experiment was to see if it was possible to provide a useful tool to aid in the improvement and automatic measurement of the quality of computer programs. Such a tool was wanted because of the large number (over 1000) of programs involved, and because of a desire to obtain quantitative measures of quality. There were five subjective quality definitions considered. The first two dealt with the extent to which reduction in object code could be made via simple transformations or a complete restructuring of the program. The next two consisted of the extent to which reductions in the number of source statements could be made via simple transformations or a complete restructuring of the program. The last was the ranked clarity of the program source. The principal method used was the manual grading of a sample, extraction of quantifiable independent variables, and the use of regression analysis to derive prediction formulas. Results included some tentative quality prediction formulas, correlations among the independent and dependent variables (e.g., GOTO's and clarity), observations about programming, and a host of newly-generated questions. There are indications that for two large program populations, we have derived a useful tool for automatically differentiating between good and bad programs.

[Ande76a] Abstract: The need for reliable complex systems motivates the development of techniques by which acceptable service can be maintained, even in the presence of residual errors. Recovery blocks allow a software designer to include tests on the acceptability of the various phases of a system's operation, and to specify alternative actions should the acceptance tests fail. This approach relies on certain architectural features, ideally implemented in hardware, by which control and data structures can be retrieved after errors.

A brief account is presented of the recovery block scheme, together with a description of a new implementation of the underlying cache mechanism. The salient features of a proposed computer architecture are described, which incorporates this implementation and also provides a high level detection for errors such as the corruption of code and data. A prototype system has been constructed to test the viability of these techniques by executing programs containing recovery blocks on an emulator for the proposed architecture. Experiences in running this system are recounted with respect to the execution of program based on erroneous algorithms and also with respect to errors introduced by deliberate attempts to corrupt the system.

[Ande76b] Summary: SEMANOL is a practical programming system for writing readable formal specifications of the syntax and semantics of programming languages. SEMANOL is based on a theory of semantics which embraces algorithmic (operational) and extensional (input/output) semantics. Specifications for large contemporary languages have been constructed in the formal language, SEMANOL (73), which is a readable high-level notation. A SEMANOL (73) specification can be executed (by an existing interpreter program); when given a program from the specified language, and its input, the execution of the SEMANOL (73) provides important practical advantages. This paper includes discussions of the theory of semantics underlying SEMANOL, the syntax and semantics of the SEMANOL (73) language, the use of the SEMANOL (73) language in the SEMANOL method for describing programming languages, and the contrast between the Vienna definition method (VDL) and SEMANOL.

[Ande79a] Table of Contents: Mathematical Induction. Proving the correctness of flowchart programs, basic principles of proving flowchart programs correct, the inductive assertion method, and formalizing inductive assertion proofs. Proving the correctness of programs written in a standard programming language, examples for Fortran and PL/I. Proving the correctness of recursive programs by using structural induction. Current research related to proving program correctness. References.

[Ande83] Abstract: Real-time systems often have very high reliability requirements and are therefore prime candidates for the inclusion of fault tolerance techniques. In order to provide tolerance to software faults, some

form of state restoration is usually advocated as a means of recovery. State restoration can be expensive and the cost is exacerbated for systems which utilize concurrent processes. The concurrency present in most real-time systems and the further difficulties introduced by timing constraints suggest that providing tolerance for software faults may be inordinately expensive or complex. We believe that this need not be the case, and propose a straightforward pragmatic approach to software fault tolerance which is believed to be applicable to many real-time systems. The approach takes advantage of the structure of real-time systems to simplify error recovery, and a classification scheme for errors is introduced. Responses to each type of error are proposed which allow service to be maintained.

[Ande85] Abstract: In order to assess the effectiveness of software fault-tolerance techniques for enhancing the reliability of practical systems, a major experimental project has been conducted at the University of Newcastle upon Tyne. Techniques were developed for, and applied to, a realistic implementation of a real-time system (a naval command and control system). Reliability data were collected by operating this system in a simulated tactical environment for a variety of action scenarios. This paper provides an overview of the project and presents the results of three phases of experimentation. An analysis of these results shows that use of the software fault tolerance approach yielded a substantial improvement in the reliability of the command and control system.

[Ande88] Abstract: Analysis of the WIS Ada source code involved applying an automated, hierarchical, Ada-specific software metrics framework to approximately 200,000 lines of Air Force-supplied Ada source. The purpose of the analysis was to aid the Air Force in identification of the characteristics of the code that detract unnecessarily from reliability, maintainability, and portability. The software was analyzed during the initial phase of code development to insure that sufficient time would be allotted for the elimination of undesired characteristics.

DRC's Ada metrics framework measures three software factors, six software criteria, and 150 software metric elements, where each metric element relates a software quality principle to the use of specific features of the Ada language.

The analysis of the Air Force-supplied Ada source involved:

1. automated calculation of metric scores for the supplied source,
2. human analysis of the metric scores to determine those characteristics that augment or attenuate quality and to formulate recommendations on how to enhance quality,
3. modification of two modules of the supplied source to illustrate the impact of [the authors] recommendations, and
4. reporting of the findings to the Air Force.

[Andr81] Abstract: This paper describes an automated testing methodology and an experiment performed to determine its effectiveness. The method is to insert in the program to be tested a number of "executable assertions," statements about the program that trigger error signals whenever they are evaluated to be false (violated). A testcase is then developed for the program using actual values of the input variables. When the program is run, a plot is generated of the number assertions violated versus the input variable values used. The resulting function is called the "error function." Heuristic search algorithms can then be used to maximize this function and thereby automatically locate input values which cause the most errors to occur. The experiment included developing assertions for the program to be tested, choosing and inserting representative errors into the program, and implementing search and data collection algorithms for testing. The results indicate that combining executable assertions with heuristic search algorithms is an effective method for automating the testing of computer programs.

[Angl83] Abstract: There has been a great deal of theoretical and experimental work in computer science on inductive inference systems, that is systems that try to infer general rules from examples. However, a complete and applicable theory of such systems is still a distant goal. This survey highlights and explains the main ideas that have been developed in the study of inductive inference, with special emphasis on the relations between the general theory and the specific algorithms and implementation.

[Angu80] Abstract: The purpose of this study was the validation of existing software (S/W) reliability models. This validation was accomplished by investigating the properties of model parameter estimates, by investigating the validity of model internal assumptions, and by analyzing the goodness-to-fit of the models. These investigations were all made in terms of actual S/W error data for sixteen (16) electronic system computer programs which represented a wide variety of system types.

The types of S/W reliability models studied were basically two: Poisson and Binomial models. The methods of parameter estimation investigated were also two: the maximum likelihood method and the least squares method.

[Angu83] Abstract: The objective of this study was to demonstrate the use and applicability to Air Force software acquisition managers of six quantitative software reliability models to a major command, control, communications, and intelligence (C3I) system. The scope of the effort involved the collection of software error data from an ongoing C3I project, fitting the six models to the data thus collected, analysis of the predictions provided by the models, and the development of conclusions, recommendations, and guidelines for software acquisition managers pertaining to the use and applicability of the six software reliability models.

[Appe88] Abstract: Mutation analysis is a method for software testing in which many slightly differing versions of a program are executed on the same test data; the end result is a measure of the data's quality. Over the last ten years, several mutation-based systems have demonstrated the usefulness of mutation analysis for software testing. This paper shows how mutation analysis can be a useful tool for testing large scale Ada software systems. We first sketch the general theory of mutation analysis, then show how to apply it to Ada. We then discuss some of the significant new problems that Ada poses for mutation analysis. Finally, we describe a prototype multilanguage mutation system that allows the testing of both Fortran 77 and Ada programs.

[Apt80] Abstract: An axiomatic proof system is presented for proving partial correctness and absence of deadlock (and failure) of communicating sequential processes. The key (meta) rule introduces cooperation between proofs, a new concept needed to deal with proofs about synchronization by message passing. CSP's new convention for distributed termination of loops is dealt with. Applications of the method involve correctness proofs for two algorithms, one for distributed partitioning of sets, the other for distributed computation of the greatest common divisor of n numbers.

[Apt81] Abstract: A survey of various results concerning Hoare's approach to proving partial and total correctness of programs is presented. Emphasis is placed on the soundness and completeness issues. Various proof systems for **while** programs, recursive procedures, local variable declarations, and procedures with parameters, together with the corresponding soundness, completeness, and incompleteness results, are discussed.

[Apt83a] Abstract: In a previous paper a proof system dealing with partial correctness of communicating sequential processes was introduced. Soundness and relative completeness of this system are proved here. It is also indicated in what way the semantics and the proof system can be extended to deal with the total correctness of the programs.

[Ardo88] Abstract: IDA Paper P-2036 presents a simple architecture specification in the SDI Architecture Dataflow Modeling Technique (SADMT). The example code is given in the SADMT Generator (SAGEN) Language. This simple architecture includes (1) an informal description of the architecture, (2) the main program that creates the components of the simulation, (3) the specification of the BM/C3 logical processes of the architecture, (4) the specification of the Technology Modules (TMs) of the architecture, and (5) the specification of the BM/C3 and the TMs of the threat.

[Army84] Preface: This Software Quality Engineering Handbook was developed by the USA Army Computer Systems Command, Quality Assurance Directorate. It describes techniques for establishing quality goals for a software project, applying those goals during software development, and evaluating fulfillment of those goals.

[Army87] Overview: The Software Test and Evaluation Manual is a three volume reference set that provides checklists and guidance to Department of Defense components in the area of software test and evaluation for major systems through improved acquisition management and risk reduction procedures. This manual addresses the structuring, planning, conduct, and evaluation of software tests throughout the acquisition process. Volume II is intended for use by the Service Headquarters, Development Commands, Program Offices and Contractors, Development Test Agencies, and Operational Test Agencies.

[Arth88] Abbreviated Introduction: Software maintenance is a complex and costly activity. Such activities significantly outweigh developmental costs and are estimated to consume more than half of the total life cycle cost of a system. Factors contributing to this substantial burden include:

- the demand for and shortage of maintenance personnel who possess the necessary skills demanded by maintenance activities,
- the lack of complementary methods of techniques for performing maintenance activities, and
- the scarcity of tools for supporting activities intrinsic to the maintenance of complex software system.

In an effort to control the complexity and costs associated with maintenance activities, a group of software engineers at the Naval Surface Warfare Center (NSWC) in Dhalgren, Virginia has developed the Automated Design Description System (ADDS) that supports maintenance activities through the use of reverse engineering techniques. This report examines ADDS relative to current technologies, and discusses the strengths and weaknesses of the system as a tool for supporting software maintenance.

The Automated Design Description System employs reverse engineering concepts to produce specialized documents tailored for the maintenance activity. Many studies tout the need for and benefits of documentation during software maintenance. Other authors have enunciated the principle of *concurrent documentation* that reflects *current* system design and development status. Although recognized as a necessity, the maintenance of a consistently high level of documentation throughout the software life cycle is rarely achieved. Consequently, software engineers have sought methods and tools to compensate for human inadequacies in documentation. Slowly, reverse engineering concepts have emerged, as have tools such as ADDS.

[Auer86] Abstract: RT-ASLAN is a formal language for specifying real-time systems. It is an extension of the ASLAN specification language for sequential systems. Some of the features of the ASLAN language, such as constructs for writing procedural semantics in a nonprocedural logical language, are highlighted. The RT-ASLAN language supports specification of parallel real-time processes through arbitrary levels of abstraction; processes do not have to be specified to the same level of detail. Communicating processes use an interface process as an abstract data type representing shared information. From RT-ASLAN specifications, performance correctness conjectures are generated. These conjectures are logic statements whose proof guarantees the specification meets critical time bounds. A detailed example as well as a discussion of the advantages and disadvantages of formal specification and verification are included.

[Aviz75] Abstract: Two complementary methods which are employed in order to assure reliable computing are fault-intolerance and fault-tolerance. Fault-intolerance depends on the elimination of the causes of unreliability prior to the start of the computing process while fault-tolerance employs protective redundancy during the computing process in order to detect and to correct unreliable functioning. A balanced allocation of reliability resources between the two methods appears to offer the best practical solution. The paper reviews current fault-tolerance practices in system architecture and discusses their relevance to software systems.

[Aviz77] Abstract: N-version programming results in N independently generated, but functionally equivalent programs which are intended to provide fault-tolerance for software faults during program execution. A pilot experiment in N-version programming is described and an evolving methodology for this form of programming is outlined.

[Aviz84] Abbreviated Introduction: Fault tolerance is the survival attribute of computer architectures; when a system is able to recover automatically from fault-caused errors, and eliminate faults without suffering an

externally perceivable failure, the system is said to be fault tolerant. Originally, fault-tolerant *architectures* were developed to tolerate *physical* faults that occur because of random failure phenomena in the hardware of a system. More recently, the tolerance of *design* faults, especially in software, has been added to the objectives of fault tolerance.

Design diversity is the approach in which the hardware and software elements that are to be used for multiple computations are not copies, but are independently designed to meet a system's requirements. Different designers and design tools are employed in each effort, and commonalities are systematically avoided. The obvious advantage of design diversity is that reliable computing does not require the complete absence of design faults, but only that those faults not produce similar errors in a majority of the designs. This and other advantages, as well as some limitations of design diversity, are discussed in this article.

[Aviz85] Abstract: Evolution of the N-version software approach to the tolerance of design faults is reviewed. Principal requirements for the implementation of N-version software are summarized and the DEDIX distributed supervisor and testbed for the execution of N-version software is described. Goals of current research are presented and some potential benefits of the N-version approach are identified.

[Aviz87] Overview: The Advanced Automation System, or AAS, will provide automation services to both en-route and terminal air traffic controllers throughout the United States. Although controllers are able to maintain separation between aircraft during periods of interruption of the automatic services, the transition to backup modes of operation is potentially hazardous. The increased controller workload resulting from interruption of the services provided to controllers limits the traffic handling capability of the Air Traffic Control system, which can result in major delays during periods of heavy traffic. As the level of automation services provided to controllers increases, interruption of computer services to the controllers will become even more critical. Accordingly, extremely high reliability and availability of the services provided by the AAS will be required on a continuous basis, 24 hours a day, seven days a week.

In this article, we will discuss only the main element of the AAS, which is the area control computer coupler, or ACCC. The approach used to define the ACCC requirements illustrates the approach used for the other computer complexes as well.

[Avru85] Abstract: In this paper we outline an approach to describing and analyzing designs for distributed software systems. A descriptive notation is introduced, and analysis techniques applicable to designs expressed in that notation are presented. The usefulness of the approach is illustrated by applying it to a realistic distributed software-system design problem involving mutual exclusion in a computer network.

[Avru86] Abstract: We describe an approach to the design of concurrent software systems based on the *constrained expression* formalism. This formalism provides a rigorous conceptual method for the semantics of concurrent compilations, thereby supporting analysis of important system properties as part of the design process. At the same time, [the authors] approach allows designers to use standard specification and design languages, rather than forcing them to deal with the formal model explicitly or directly. As a result, [the authors] approach attains the benefits of formal rigor without the associated pain of unnatural concepts or notations for its users.

The conceptual model of concurrency underlying the constrained expression formalism treats the collection of possible behaviors of a concurrent system as a set of sequences of events. The constrained expression formalism provides a useful closed-form description of these sequences. We have developed algorithms for translating designs expressed in a wide variety of notations into these constrained expressions. We have also developed a number of powerful analysis tools that can be applied to these descriptions.

In this paper, we describe the constrained expression formalism and these analysis techniques. We describe the way this approach would be used in design, giving an example illustrating its use in conjunction with an Ada-like design language, and discuss present and future prospects for its automation and use.

[Bagg80] Abstract: In the past, most software tests were constructed by heuristics and by drawing upon experience with similar software. Recently, enough preliminary work has been done to propose an analytical

construction of test cases.

This report begins by defining five broad classes of software tests: Type O, Type 1, Type 2, Type 3 and Type 4. In a Type O test, all instructions are exercised at least once. In a Type 1 and 2 test, all flowchart paths are exercised at least once. Type 1 is performed by forced traversal and Type 2 by natural execution. Types 3 and 4 are unfeasible and only a strategy lying between Type 1 and 2 can effectively be implemented.

Since enumeration of all the paths in a given program is required for Type 1 and 2 tests, this report establishes the lower and upper bounds on the number of paths as a function of the number of deciders, describes a manual decomposition procedure to cut a graph into smaller subgraphs, and proposes an algorithm to machine-identify all paths. A complete Type 1.5 driver system for forced path traversal, implemented in PL/1, is then thoroughly described, together with suggestions on how to extend these techniques to other languages.

A typical program is analyzed manually, tested with data and run through the system. Some evaluation of the usefulness of the system is eventually given in the light of the accumulated experience.

[Baia84] Abstract: The structure of a complier for the ECSP language is described. ECSP is a concurrent language extending Hoare's CSP: it supports dynamic communication channels and nested processes. The compilation of ECPS programs is obtained by the composition of several tools of minimal functionalities.

A set of static checks on interactions between concurrent processes is described, The checks verify the mutual consistency of the interfaces of processes: an interface is given by a set of input/output channels connecting a process to its partners. It is shown that the amount of the coverage of checks depend on the entities referred to in interprocess communication constructs and that both increase with the adoption of explicit naming.

The checks on process interfaces are carried on in several tool of the complier front-end to achieve machine independence. To support separate complication, each tool can be applied to a subset of the processes of the program.

[Baia85] Abstract: This work discusses some issues in the debugging of concurrent programs. A set of desirable characteristics of a debugger for concurrent languages is deduced from an examination of the differences between the debugging of concurrent programs and that of sequential ones. A debugger for a concurrent language, derived from CSP, is then presented. It is based upon a semantic model of the supported language. The debugger enables [us] to compare a description of the program behavior to the actual behavior as well as to evaluate assertions on the process state. The description of the behavior is given by a formalism whose semantics is also specified. The formalism can specify program behaviors at various abstraction levels. Lastly, some guidelines for the implementation of the debugger are shown and a detailed example of program description is analyzed.

[Bail80] Abstract: One of the basic goals of software engineering is the establishment of useful models and equations to predict the cost of any given programming project. Many models have been proposed over the last several years, but, because of differences in the data collected, types of projects and environmental factors among software development sites, these models are not transportable and are only valid within the organization where they were developed. This result seems reasonable when one considers that a model developed at a certain environment will only be able to capture the impact of the factors which have a variable effect within that environment. Those factors which are constant at that environment, and therefore do not cause variations in the productivity among projects produced there, may have a different or variable effects at another environment.

This paper presents a model-generation process which permits the development of a resource estimation model for any particular organization. The model is based on data collected by that organization which captures its particular environmental factors and the differences among its particular projects. The process provides the capability of producing a model tailored to the organization which can be expected to be more effective than any model originally developed for another environment. It is demonstrated here using data collected from the Software Engineering Laboratory at the NASA/Goddard Space Flight Center.

[Bail81] Abstract: This paper describes an application of Maurice Halstead's software theory to a real time

switching system. The Halstead metrics and the software tool developed for computing them are discussed. Analysis of the metric data indicates that the level of the switching language was not constant across algorithms and that software error data was not a linear function of volume.

[Bake72a] Introduction: Experience in development and maintenance of large computer-based systems for government and industry has led the IBM Federal Systems Division to the formulation of a new approach to production programming. This approach, which couples a new kind of programming organization (a Chief Programmer Team) with formal tools for using structured programming in system development, was recently applied on a contract with The New York Times for an online information system. Compared to experience on similar contracts in the past, the approach resulted in increased programmer productivity coupled with improved quality. An earlier paper describes the approach in detail and gives productivity measures in a form which should allow comparability to other systems. Following a brief description of the system and a review of the approach, this paper discusses the quality of the system as observed during a thorough acceptance test and in the initial period of operation following its delivery.

[Bake72b] Seeking to demonstrate increased programmer productivity, a functional organization of specialists led by a chief programmer has combined and applied known techniques into a unified methodology.

Combined are a program production library, general-to-detail implementation, and structured programming. The overall methodology has been applied to an information storage and retrieval system.

Experimental results suggest significantly increased productivity and decreased system integration difficulties.

[Bake79b] Abstract: An investigation is made into the extent to which relationships from software science are useful in analyzing programming methodology principles that are concerned with modularity. Using previously published data from over 500 programs, it is shown that the software science effort measure provides quantitative answers to questions concerning the conditions under which modularization is beneficial. Among the issues discussed are the reduction of similar code sequences by temporary variable and subprogram definition, and the use of global variables. Using data flow analysis, environmental considerations which affect the applicability of alternative modularity techniques are also discussed.

The results obtained using software science are compared with certain generally accepted methodologies involving modularity, and show strong agreement. Finally, the results suggest some areas of potential improvement in the technique used to obtain the software science measurements.

[Bake80] Abstract: In attempting to describe the quality of computer software, one of the more frequently mentioned measurable attributes is complexity of the flow of control. During the past several years, there have been many attempts to quantify this aspect of computer programs, approaching the problem from such diverse points of view as graph theory and software science. Most notable measures in these areas are McCabe's cyclomatic complexity and Halstead's software effort. More recently, Woodward et al., have proposed a complexity measure based on the number of crossings, or "knots," of arcs in a linearization of the flowgraph.

Focusing on these three quantities, we establish their major properties as measures of control flow complexity. Particular attention is directed at the behavior of the measures in structured programming environments, including the effect of various structuring transformations on the measures.

As a result of this investigation, weaknesses of each of the measures taken individually are exposed. However, the software effort and cyclomatic complexity measures appear to have disjoint areas of weakness. This suggests that more comprehensive measures of control flow complexity can be motivated by consideration of combinations of these basic measures.

[Bake88] Abstract: The reliability of a program, when many copies are run in a multisite environment with the support of a software service organization, depends upon the inherent reliability of the program and certain characteristics of the service organization. In this paper we identify a small number of parameters that determine the relevant characteristics of the service organization, and analyze their effects upon the reliability of the

program as it is experienced at an average site. This is done with two software reliability models, a first discovery model and a total (defect) discovery model.

[Balz69] With the advent of the higher-level algebraic languages, the computer industry expected to be relieved of the detailed programming required at the assembly-language level. This expectation has largely been realized. Many systems are now being built in higher-level languages (most notably MULTICS).

However, our ability to debug programs has not advanced much with our increased use of the higher-level languages.

We have, in general, merely copied the on-line assembly-language debugging aids, rather than design totally new facilities for higher-level languages. We have neither created new graphical formats in which to present the debugging information, not provided a reasonable means by which users can specify the processing required on any available debugging data.

EXDAMS (EXtendable Debugging And Monitoring System) is an attempt to break this impasse by providing a single environment in which the users can easily add new on-line debugging aids to the system one-at-a-time without further modifying the source-level compilers, EXDAMS, or their programs to be debugged.

[Barr82] Abstract: An axiomatic proof system is developed for use in proving partial correctness and absence of deadlock in Ada tasks. Axioms for the Ada tasking primitives in isolation are presented, and then rules proposed that describe the logical interaction of tasks through the rendezvous mechanism. These axioms and rules are then used to present partial correctness proofs of parallel-processing examples written in Ada. The system is extended to deal with questions of blocking and detection of deadlock and, finally, the problem of task termination and exception handling are discussed.

[Barr84] Abstract: A compositional temporal logic proof system for the specification and verification of concurrent programs is presented. Versions of the system are developed for shared variables and communication based programming languages that include procedures.

[Barr85] Abstract: The book summarizes and review nine verification techniques for parallel programs, four of which are based on *shared variables* (viz., the methods proposed by Flon and Suzuki, Jones, Lamport, Owicki and Gries) while the other five are based on *message passing* (Apt, Fancez and de Roever, Barringer and Mearns, Levin and Gries, Misra and Chandy, Zhou and Hoare).

The following scheme has been assumed for the presentation of each method:

1. a list of major references to the work;
2. an overview of the method;
3. a summary of the examples the method was applied to in the major references;
4. a detailed exposition of the application of the approach to some different examples;
5. general comments on the method;
6. a summary of the proof system.

Mainly the *set partition problem* and the *bubble-lattice sort program* are used as test problems in the worked examples.

For newcomers, the book may perhaps be of some value as a first overview and guide to the literature. For experts, a much more detailed assessment and comparison of the methods reviewed would certainly have been desirable. (the *conclusions* section which compares the problems and benefits of the methods reviewed is only one and a half pages long!).

[Bart78] Abstract: A new interprocedural data flow analysis algorithm is presented and analyzed. The algorithm associates with each procedure in a program information about which variables may be modified, which may be used, and which are possibly preserved by a call on the procedure, and all of its subcalls. The algorithm is sufficiently powerful to be used on recursive programs and to deal with the sharing of variables which arises through reference parameters. The algorithm is unique in that it can compute all of this information in a single pass, not

requiring a prepass to compute calling relationships or sharing patterns. The algorithm is asymptotically optimal in time complexity. It has been implemented and is practical even on programs which are quite large.

[Basi75] Abstract: This paper recommends the "iterative enhancement" technique as a practical means of using a top-down, stepwise refinement approach to software development. This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the gradual enhancement of successive implementations in order to build the full implementation. The development and quantitative analysis of a production compiler for the language SIMPL-T is used to demonstrate that the application of iterative enhancement to software development is practical and efficient, encourages the generation of an easily modifiable product, and facilitates reliability.

[Basi78a] Abstract: The collection and analysis of data from programming projects is necessary for the appropriate evaluation of software engineering methodologies. Towards this end, the Software Engineering Laboratory was organized between the University of Maryland and NASA Goddard Space Flight Center. This paper describes the structure of the Laboratory and provides some data on project evaluation from some of the early projects that have been monitored. The analysis relates to resource forecasting using a model of the project life cycle based upon the Rayleigh equation and to error rates applying ideas developed by Belady and Lehman.

[Basi79a] Abstract: There is a need for a distinguishing set of useful automatable measures of the software development process and product. Measures are considered useful if they are sensitive to externally observable differences in development environments and their relative values correspond to some intuition regarding these characteristic differences. Such measures could provide an objective quantitative foundation for constructing quality assurance standards and for calibrating mathematical models of software reliability and resource estimation. This paper presents a set of automatable measures that were implemented, evaluated in a controlled experiment, and found to satisfy these usefulness criteria. The measures include computer job steps, program exchanges, program size, and cyclomatic complexity.

[Basi79b] Abstract: The effects of human factors on "high-level" software properties–too intangible to quantify directly–can be inferred from the collective behavior of related "low-level" aspects.

[Basi80b] Abstract: A family of structural complexity metrics which contains a number of current metrics is developed. The family may be used to give a framework for experimental analysis of metrics. By implementing the family or suitable subfamily as an automatic metric tool, may metrics become readily available and may even be merged to form new metrics in response to information obtained during exploratory analysis.

[Basi81a] Abstract: This paper presents an attempt to examine a set of basic relationships among various software development variables, such as size, effort, project duration, staff size, and productivity. These variables are plotted against each other for 15 Software Engineering Laboratory projects that were developed for NASA/Goddard Space Flight Center by Computer Sciences Corp. Certain relationships are derived in the form of equations, and these equations are compared with a set derived by Walston and Felix for IBM Federal Systems Division project data. Although the equations do not have the same coefficients, they are seen to have similar exponents. In fact, the Software Engineering Laboratory equations tend to be within one standard error of estimate of the IBM equations.

[Basi81b] Abstract: We describe in this paper an effective data collection method for evaluating software development methodologies, from definition of the objectives of the data collection to analysis of the results. We show how the data analysis can answer questions with respect to how successfully the goals of the development methodology are met. The A-7 requirements document is used as an example. We provide the results of data analyses conducted partway through the A-7 flight software development cycle, and discuss the utility of information obtained by such partial analyses. Results from the study show that data collection is feasible and useful when performed as part of configuration control, that data distributions based on partial data provide useful feedback

to the developers, and that the A-7 Requirements Document is easily maintained and changed.

[Basi81c] Abstract: A software engineering research study has been undertaken to empirically analyze and compare various software development approaches; its fundamental features and initial findings are presented in this paper. An experiment was designed and conducted to confirm certain suppositions concerning the beneficial effects of a particular disciplined methodology for software development. The disciplined methodology consisted of programming teams employing certain techniques and organizations commonly defined under the umbrella term structured programming. Other programming teams and individual programmers both served as control groups for comparison. The experimentally tested hypotheses involved a number of quantitative, objective, unobtrusive, and automatable measures of programming aspects dealing with the software development process and the developed software product. The experiment's results revealed several programming aspects for which statistically significant differences existed between the disciplined methodology and the control groups. The results were interpreted as confirmation of the original suppositions and evidence in favor of the disciplined methodology. This paper describes the specific features of the experiment; outlines the investigative approach used to plan, execute, and analyze it; reports its immediate results; and interprets them according to intuitions regarding the disciplined methodology.

[Basi81f] Abstract: This paper analyzes the resource utilization curve developed by Parr. The curve is compared with several other curves, including the Rayleigh curve, a parabola, and a trapezoid, with respect to how well they fit manpower utilization. The evaluation is performed for several projects developed in the Software Engineering Laboratory of the 6-12 man-year variety. The conclusion drawn is that the Parr curve can be made to fit the data better than the other curves. However, because of the noise in the data, it is difficult to confirm the shape of the manpower distribution from the data alone and therefore difficult to validate any particular model. Also, since the parameters used in the curve are not easily calculable or estimable from known data, the curve is not effective for resource estimation.

[Basi81g] Abbreviated Introduction: Among the most popular metrics have been the software science metrics of Halstead, and the cyclomatic complexity metric of McCabe. One question is whether these metrics actually measure such things as effort and complexity. One measure of effort may be the time required to produce a product. One measure of complexity might be the number of errors made during the development of a product. A second question is how these metrics compare with standard size measures, such as the number of source lines or the number of executable statements, i.e., do they do a better job of predicting the effort or the number of errors? Lastly, how do these metrics related to each other?

One simple way of checking the relationship between errors or effort and the various metrics is to examine the plots of variables against one another and correlations between the various variables. This provides us with a first look at attempting to shed some light on the questions posed and the relationships that may hold.

[Basi82a] Abstract: An effective data collection methodology for evaluating software development methodologies was applied to four different software development projects. Goals of the data collection included characterizing changes and errors, characterizing projects and programmers, identifying effective error detection and correction techniques, and investigating ripple effects.

The data collected consisted of changes (including error corrections) made to the software after code was written and baselined, but before testing began. Data collection and validation were concurrent with software development. Changes reported were verified by interviews with programmers. Analysis of the data showed patterns that were used in satisfying the goals of the data collection. Some of the results are summarized in the following: 1. Error corrections aside, the most frequent type of change was an unplanned design modification. 2. The most common type of error was one made in the design or implementation of a single component of the system. Incorrect requirements and misunderstandings of functional specifications, interfaces, support software and hardware, and languages and compilers were generally not significant sources of errors. 3. Despite a significant number of requirements changes imposed on some projects, there was no corresponding increase in frequency of requirements misunderstandings. 4. More than 75% of all changes took a day or less to make. 5.

Changes tended to be nonlocalized with respect to individual components but localized with respect to the subsystems. 6. Relatively few changes resulted in errors. Relatively few errors required more than one attempt at correction. 7. Most errors were detected by executing the program. The cause of most errors was found by reading code. Support facilities and techniques such as traces, dumps, cross-reference and attribute listings, and program proving were rarely used.

[Basi82b] Overview: In this newsletter, we briefly describe [the authors] approach to data collection followed by a description of the software development project that we are monitoring. We then address several central issues related to the use of Ada in the design phase of this project.

[Basi82c] Abstract: An effective data collection method for evaluating software development methodologies and for studying the software development process is described. The method uses goal-directed data collection to evaluate methodologies with respect to the claims made for them. Such claims are used as a basis for defining the goals of the data collection, establishing a list of questions of interest to be answered by data analysis, defining a set of data categorization schemes, and designing a data collection form.

The data to be collected are based on the changes made to the software during development, and are obtained when the changes are made. To ensure accuracy of the data, validation is performed concurrently with software development and data collection. Validation is based on interviews with those people supplying the data. Results from using the methodology show that data validation is a necessary part of change data collection. Without it, as much as 50 percent of the data may be erroneous.

Feasibility of the data collection methodology was demonstrated by applying it to five different projects in two different environments. The application showed that the methodology was both feasible and useful.

[Basi82d] Introduction: The identification of the various factors that have an effect on software development is of prime concern to software engineers. The specific focus of this paper is to analyze the relationships between the frequency and distribution of errors during software development, the maintenance of the developed software, and a variety of environmental factors. These factors include the complexity of the software, the developer's experience with the application, and the reuse of existing design and code. Such relationships can provide an insight into the characteristics of computer software and the effects that an environment can have on the software product. Such relationships can also improve the reliability and quality with respect to computer software. In an effort to acquire knowledge of these basic relationships, change data for a medium-scale software project were analyzed. (Change data include any documentation that reports an alteration made to the software for a particular reason.)

The overall objectives of this paper are threefold: first, to report the results of the analyses; second, to review the results in the context of those reported by other researchers; third, to draw some conclusions based on the first two objectives. The analyses presented in this paper encompass various types of distributions based on the collected change data. The most important are the error distributions observed within the software project.

[Basi83a] Abstract: The emergence of Ada provides the opportunity and necessity for measurement, analysis, and experimentation in software development. Over the past several months, we have been studying a software project developed in Ada. One of the goals of the study is to identify metrics which are useful for evaluating and predicting the complexity, quality, and cost of Ada programs. This paper defines a set of metrics for use with software development in Ada. The metrics are gathered into six categories: effort, changes, dimension, language use, data use, and execution. They are described further using formula generators, distributions, and formulas. Examples of each metric, as well as specific uses, are also included. Finally, [the authors] continuing research in this area is described.

[Basi83b] Abstract: The desire to predict the effort in developing or explain the quality of software has led to the proposal of several metrics in the literature. As a step toward validating these metrics, the Software Engineering Laboratory has analyzed the Software Science metrics, cyclomatic complexity, and various standard program

measures for their relation to 1) effort (including design through acceptance testing), 2) development errors (both discrete and weighted according to the amount of time to locate and fix), and 3) one other. The data investigated are collected from a production Fortran environment and examined across several projects at once, within individual projects and by individual programmers across projects, with three efforts reporting accuracy checks demonstrating the need to validate the database. When the data come from individual programmers or certain validated projects, the metrics' correlations with actual effort seem to be the strongest. For modules developed entirely by individual programmers, the validity ratios induce a statistically significant ordering of several of the metrics' correlations. When comparing the strongest correlations, neither Software Science's E metric, cyclomatic complexity nor source lines of code appears to relate convincingly better with effort than the others.

[Basi83c] Abstract: A family of syntactic complexity metrics is defined that generates several metrics commonly occurring in the literature. The paper uses the family to answer some questions about the relationship of these metrics to error-proneness and to each other. Two derived metrics are applied: "slope" which measures the relative skills of programmers at handling a given level of complexity and "r square" which is indirectly related to the consistency of performance of the programmer or team. The study suggests that individual differences have a large effect on the significance of results where many individuals are used. When an individual is isolated, better results are obtained. The metrics can also be used to differentiate between projects on which a methodology was used and those on which it was not.

[Basi84a] Abstract: A large, commercially available Fortran program was modified to produce structural coverage metrics. The modified program was executed on a set of functionally generated acceptance tests and a large sample of operational usage cases. The resulting structural coverage metrics are combined with fault and error data to evaluate structural coverage in the SEL environment.

We can show that in this environment the functionally generated test cases seem to be a good approximation of the operational use. The relative proportions of the exercise statement subclasses (executable, assignment, CALL, DO, IF, READ, WRITE) change as the structural coverage of the program increases. We propose a method for evaluating if two sets of input data exercise a program in a similar manner.

We also provide evidence that implies that in this environment, faults revealed in a procedure are independent of the number of times the procedure is executed and that it may be reasonable to use procedure coverage in software models that use statement coverage. Finally, the evidence suggests that it may be possible to use structural coverage to aid the management of the acceptance test process.

[Basi84d] Abstract: A considerable amount of money and resources has been spent on the development of the new programming language Ada. The University of Maryland and General Electric have studied the development of a software project written in Ada. This paper presents the analysis of the effort, change, and error data. The total effort spent on training and methodology was 20% of the total effort spent on the project; this was more than the effort spent on any other phase. The greatest error rates appeared to be associated with the most Ada-specific features; tasking, generics and compilation units. Experience with high-level languages seemed to be associated with a better ability to grasp Ada concepts. Finally, the results strongly indicate the need for support tools for an Ada programming environment.

[Basi85a] Abstract: Since both cost/quality goals and production environments differ, this study presents an approach for customizing a characteristic set of software metrics to an environment. The approach is applied in the Software Engineering Laboratory (SEL), a NASA Goddard production environment, to 49 candidate process and product metrics of 652 modules from six (51,000-112,000 line) projects. For this particular environment, the method yielded the characteristic metric set (source lines, fault correction effort per executable statement, design effort, code effort, number of I/O parameters, number of versions). The uses examined for a characteristic metric set include forecasting the effort for development, modification, and fault correction of modules based on historical data.

[Basi85b] Abstract: This study compares the strategies of code reading, functional testing, and structured

testing in three aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Thirty two professional programmers and 42 advanced students applied the three techniques to four unit-sized programs in a fractional factorial experimental design. The major results of this study are the following.

1. With the professional programmers, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate.
2. In one advanced student subject group, code reading and functional testing were not different in faults found, but were both superior to structural testing, while in the other advanced student subject group there was no difference among the techniques.
3. With the advanced student subjects, the three techniques were not different in fault detection rate.
4. Number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested.
5. Code reading detected more interface faults than did the other methods.
6. Functional testing detected more control faults than did the other methods.
7. When asked to estimate the percentage of faults detected, code readers gave the most accurate estimates while functional testers gave the least accurate estimates.

[Basi85c] Abstract: This paper presented a paradigm for evaluating software development methods and tools. The basic idea is to generate a set of goals which are refined into quantifiable questions which specify metrics to be collected on the software development and maintenance process and product. These metrics can be used to characterize, evaluate, predict and motivate. They can be used in an active as well as passive way by learning from analyzing the data and improving the methods and tools based upon what is learned from that analysis. Several examples were given representing each of the different approaches to evaluation.

[Basi85e] Abstract: Estimating the amount of effort required for a software development project is one of the major aspects of resource estimation for that project. In this study, the relationship between effort and other variables for 23 Software Engineering Laboratory projects that were developed for NASA/Goddard Space Flight Center was examined. These variables fell into categories: those which can be determined in the early stages of project development and may therefore be useful in a baseline equation for predicting effort in future projects, and those which can be used mainly to characterize or evaluate effort requirements and thus enhance [the authors] understanding of the software development process in this environment. Some results of the analyses are presented in this paper.

[Basi85h] Abbreviated Introduction: This article examines the use of Ada in a software project developed by the General Electric Company. The project was monitored by the University of Maryland and GE to identify areas of success and difficulty in learning and using Ada as both a design and a coding language. Since production-quality Ada translators were not readily available, the study focused on training and early software development. We focus on the use and effect of Ada on this project, which was conducted primarily in 1982. Our study also presents the major factors to consider before using Ada in software development, particularly when training in Ada is necessary. Although many of [the authors] conclusions may seem obvious now, they were unexpected when this project began.

[The authors] study attempts to meet several goals. The first focuses on characterization of the effort, the changes, and the errors of the project. The second considers how Ada was used on the project. The third concerns evaluation of the data collection and validation process, while the fourth concentrates on the development of measures for the Ada Programming Support Environment.

[Basi86a] Abstract: Experimentation in software engineering supports the advancement of the field through an iterative learning process. In this paper we present a framework for analyzing most of the experimental work performed in software engineering over the past several years. We describe a variety of experiments in the framework and discuss their contribution to the software engineering discipline. Some useful recommendations for the

application of the experimental process in software engineering are included.

[Basi87a] Abstract: More and more project environments will make the transition from traditional implementation languages to Ada. In this context, many open questions need to be answered, e.g., whether or not Ada language features and concepts are used appropriately, and how Ada projects should be managed and supported by methods and tools. It is therefore necessary to measure and evaluate the quality and productivity of process and product aspects of Ada projects. This can be done by either conducting case studies of ongoing Ada projects or experiments in controlled environments. In both cases concrete measurement and evaluation goals need to be established in a systematic way, measures need to be derived that can help in achieving these goals, and the necessary data need to be collected, validated and interpreted. We have established a methodology that allows us to perform these activities in a systematic way. However, the methodology must be supported by automated tools in order to allow on-line feedback of evaluation results into ongoing projects. In the long-run, these tools for on-line feedback should become part of each APSE supporting the decision making process of management, development, quality assurance personnel, and others. Such information would be based on data from the current project as well as previous projects in the same and other environments. In this paper we present and discuss the TAME (Tailoring an Ada Measurement Environment) project which aims at the development of a prototype measurement and evaluation environment that supports all the previously mentioned activities including the process of setting up measurement and evaluation goals and deriving supportive measures. We discuss the TAME requirements and architectural design, the status of the first prototype, and the expected impact of this project on Ada projects and APSEs. The prototype currently under development does not interface with an APSE; however, it is designed for being integrated into an APSE in the future.

[Basi87b] Abstract: This paper presents a methodology for improving the software process by tailoring it to the specific project goals and environment. This improvement process is aimed at the global software process model as well as methods and tools supporting that model. The basic idea is to use defect profiles to help characterize the environment and evaluate the project goals and the effectiveness of methods and tools in a quantitative way. The improvement process is implemented iteratively by setting project improvement goals, characterizing those goals and the environment, in part, via defect profiles in a quantitative way, choosing methods and tools fitting those characteristics, evaluating the actual behavior of the chosen set of methods and tools, and refining the project goals based on the evaluation results. All these activities require analysis of large amounts of data and, therefore, support by an automated tool. Such a tool - TAME (Tailoring A Measurement Environment) - is currently being developed.

[Basi88] Abstract: Experience from a dozen years of analyzing software engineering processes and products is summarized as a set of software engineering and measurement principles that argue for software engineering process models that integrate sound planning and analysis into the construction process.

In the TAME (Tailoring A Measurement Environment) project at the University of Maryland we have developed such an improvement-oriented software engineering process model that uses the goal/questions/metric paradigm to integrate the constructive and analytic aspects of software development. The model provides a mechanism for formalizing the characterization and planning tasks, controlling and improving projects based on quantitative analysis, learning in a deeper and more systematic way about the software process and product, and feeding the appropriate experience back into the current and future projects.

The TAME system is an instantiation of the TAME software engineering process model as an ISEE (Integrated Software Engineering Environment). The first in a series of TAME system prototypes has been developed. An assessment of experience with this first limited prototype is presented including a reassessment of its initial architecture. The long-term goal of this building effort is to develop a better understanding of appropriate ISEE architectures that optimally support the improvement-oriented TAME software engineering process model.

[Bate83a] Abbreviated Introduction: In this paper we consider the Behavioral Abstraction (BA) approach to high-level debugging of distributed systems. In Section 2, we discuss behavioral abstraction and the Event

Definition Language that is the basis for a debugging tool implementing this approach. Section 3 addresses one of the fundamental issues arising in actually providing debugging aid through the BA approach, that of recognizing the occurrence of abstracted behaviors. We conclude the paper with an assessment of [the authors] present status and outstanding problems.

[Baue79b] Overview: Formal program construction by transformations is a method of software development in which a program is derived from a formal problem specification by manageable, controlled transformation steps which guarantee that the final product meets the initial specification. This methodology has been investigated in the Munich project CIP (computer-aided, intuition-guided programming). The research includes the design of a wide-spectrum language specifically tailored to the needs of transformational programming, the construction of a transformation system to support the methodology, and the study of transformation rules and other methodological issues. Particular emphasis has been laid on developing a sound theoretical basis for the overall approach.

[Baue89] Abstract: Formal program construction by transformations is a method of software development in which a program is derived from a formal problem specification by manageable, controlled transformation steps which guarantee that the final product meets the initial specification. This methodology, has been investigated in the Munich project CIP (*c*omputer-aided, *i*ntuition-guided *p*rogramming). The research includes the design of a wide-spectrum language specifically tailored to the needs of transformational programming, the construction of a transformation system to support the methodology, and the study of transformation rules and other methodological issues. Particular emphasis has been laid on developing a sound theoretical basis for the overall approach.

[Bazz82] Abstract: A new method for testing compilers is presented. The compiler is exercised by compilable programs, automatically generated by a test generator. The generator is driven by a tabular description of the source language. This description is in a formalism which nicely extends context-free grammars in a context-dependent direction, but still retains the structure and reliability of BNF. The generator produces a set of programs which cover all grammatical constructions of the source language, unless user supplied directives instruct otherwise. The programs generated can also be used to evaluate the performance of difference compilers of the same source language.

A significant example from Pascal is presented, and experience with the generator is reported.

[Beck76] Abstract: Programs which perform partial evaluation, beta-expansion, and certain optimizations on programs, are studied with respect to implementation and application. Two implementations are described, one "interpretive" partial evaluation, which operated directly on the program to be partially evaluated, and a "compiling" system, where the program to be partially evaluated is used to generate a specialized program, which in its turn is executed to do the partial evaluation. Three applications with different requirements on these programs are described. Proofs are given for the equivalence of the use of the interpretive system and the compiling system in two of the three cases. The general use of the partial evaluator as a tool for the programmer in conjunction with certain programming techniques is discussed.

[Behr83] Abstract: The function point method of measuring application development productivity developed by Albrecht is reviewed and a productivity improvement measure introduced. The measurement methodology is then applied to 24 development projects. Size, environment, and language effects on productivity are examined. The concept of a productivity index which removes size effects is defined and an analysis of the statistical significance of results is presented.

[Beiz83] Table of Contents: Introduction. The taxonomy of bugs. Flowcharts and path testing. Path testing and transaction flows. Graphs, paths and complexity. Paths, path products, and regular expressions. Data validation and syntax testing. Data-base-driven testing design. Decision tables and boolean algebra. Boolean algebra the easy way. States, state graphs, and transition testing. Graph matrices and applications.

[Bela76] Abbreviated Introduction: As a need for a discipline of software engineering has been recognized, the

design, implementation, and maintenance of computer software has come into the forefront. The formulation of concepts of programming methodology, exemplified by Dijkstra's structured programming, strikes at the roots of the problem. The realization is that a program, much as a mathematical theorem, should and can be provable. Recognition that a program can be proved correct as it is developed and maintained, and before its results are used, may ultimately change the nature of the programming task and the face of the programming world. Clearly these developments are of fundamental importance. They appear to point to long-term solutions to problems that will be encountered in creating the great amount of program text that the world appears to require. But even though progress in mastering the science of program creation, maintenance, and expansion has also been made, there is still a long way to go.

[Bela81] Abstract: Program modules and data structures are interconnected by calls and references in software systems. Partitioning these entities into clusters reduces complexity. For very large systems manual clustering is impractical. A method to perform automatic clustering is described and a metric to quantify the complexity of the resulting partition is developed.

[Belk86] Abstract: The development of correct specifications is a critical task in the software development process. This paper describes an alternative approach for the development of specifications. The approach relies on a specification language for abstract data types and a synthesis system. The system is capable of translating an abstract data type specification into an executable program. This process defines an alternative methodology that provides the necessary tools for the early testing of the specifications and for the development of prototypes and implementation models.

[Bene85] Abstract: Modern complex system reliability has to take into account more and more programmed system reliability. This raises two kinds of problems:
— Software reliability: i.e., software quality assurance, specifications, development methods, languages, programming, test policies,...
— Hardware reliability at three levels: input, processors, output.
A method among others enabling to modelize software and hardware behaviour from the reliability point of view is the stochastic Markov process method.

First, the principles of the method will be given and advantages will be pointed out in comparison with other more classical methods for reliability analysis.

In the second part of the paper, software tools to solve this kind of problems will be described and in the final part of the presentation an example of successful use of these computer codes will be given.

[Beng87] Abstract: Operational analysis, an area of study first defined in the computer science field, has been used in the analysis of systems performance. System performance measures for a specific set of output data are obtained using operational analysis formulas derived from assumptions which are verifiable by the observed data. This paper gives relationships which may be used to quantify the errors in these assumptions. Additionally, basic propositions are given which help in understanding operational analysis assumptions. These propositions are used in developing correction terms which can be used to adjust performance measures so that their values are exact for a set of data no matter how much the assumptions used in deriving the performance measure relations are violated.

[Bens81] Abstract: An experiment was performed in which executable assertions were used in conjunction with search techniques in order to test a computer program automatically. The program chosen for the experiment computes a position on an orbit from the description of the orbit and the desired point.

Errors were interested in the program randomly using an error generation method based on published data defining common error types. Assertions were written for program and it was tested using two different techniques. The first divided up the range of the input variables and selected test cases from within the subranges. In this way a "grid" of test values was constructed over the program's input space.

The second used a search algorithm from optimization theory. This entailed using the assertions to define

an error function and then maximizing its value. The program was then tested by varying all of them. The results indicate that this search testing technique was as effective as the grid testing technique in locating errors and was more efficient. In addition, the search testing technique located critical input values which helped in writing correct assertions.

[Bera83] Introduction: This column is the first in a series of articles dealing with the utilization of Ada in software engineering. All examples will use Ada, even though virtually all of the techniques discussed could apply to any programming language. This will be particularly advantageous when we wish to compare Ada to other programming languages.

[Berg82] Table of Contents: Introduction to fundamental techniques. Formal models of computation. Verification methods and techniques. Approaches to proofs of partial correctness. Approaches to proofs of total correctness. Correctness of parallel programs. Application of verification approaches. Approaches to specification. State of the art and summary. References.

[Berl80] Introduction: There have been numerous measures proposed to measure program complexity. Some are completely heuristic, comparing certain measurable program features against a set of predefined standards. Some are topological, based on the number of regions on the control or data graph of the program or a combination of the avove, and of course, there is Halstead's Software Physics.

All of these measures have their deficiencies and, no doubt, so will ours. We have, however, set ourselves the goal of eliminating some of them and to provide a measure which has mathematical and intuitive correctness and which will have a good correlation with observed facts.

[Bern84] Introduction: The job of software maintenance-correcting errors and changing program operation as requirements change-generally devolves upon personnel not involved in the original software development cycle who must learn how a program works before they can competently change it. Among the variables involved in this learning process are the accuracy, currency, and completeness of program documentation; programmer skill and experience; environmental factors such as urgency, the programming language, and especially, the attributes of the program itself.

Program maintainability and program understandability are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain. And the more difficult it is to maintain, the higher its maintainability risk. Since it is to the source program that maintenance staff must ultimately come, it would be useful to be able to quantify the relative magnitude of the task through an analysis solely of the attributes of the program.

Attempts have been made to quantify program difficulty by manipulative simple counts of *selected* program attributes, *e.g.*, lines of code, bifurcation points (cyclomatic number), and operations and operands (Halstead length). Although these manipulations may be informative, none has been persuasively shown to be a reliable measure of program difficulty.

This paper presents an approach based on the tenet that program difficulty represents the sum of the difficulties of its constituent elements, and that these elements can be quantified by the use of carefully selected weights and factors.

[Berr87] Abstract: In carrying out SDC's Formal Development Method, one writes a specification of a system under design in the Ina Jo specification language and proves that the specification meets the requirements of the system. This paper develops an abstract machine model of what is specified by a level specification in an Ina Jo specification. It describes the state as defined by the front matter, computations as defined by initial states and transforms, and invariants, criteria, and constraints as properties of computations. The paper then describes a number of formal design methods and the kinds of abstractions that they require. For each of these kinds of abstractions, there is a characteristic relationship between refinements that should be proved as one is carrying out the method.

[Besh85] Abstract: Hardware manufacturers usually provide a "data sheet" listing parameters that describe the hardware's functionality and limitations on its use. This paper explores the feasibility of developing a similar document to describe software quality. It defines parameters that are useful for reporting quality in three major areas: reliability, maintainability, and robustness.

[Bess87] Overview: The key purpose of the Test Environment Generator (GET) is to provide the automatic generation of drivers and possible virtual bodies associated with a software component. Drivers and virtual bodies will constitute together with the component itself, a complete compilable, linkable and executable program. This program (called Test Environment) is interactive; when executed, it allows the user to perform most of the operations that are usually executed through a test program. For example, the user can create variables, assign objects, call subprograms, indicating the values of in or in-out parameters, and examine, after the call, the result values.

In addition to important time saving, the automatic generation of a testing environment facilitates the performance of the tests by a separate testing team. It also adds to the efficiency and comfort of such a team by providing a standard and powerful user interface to be used for all the components to be tested.

[Bish86] Abstract: The Project On Diverse Software (PODS) was a collaborative software reliability research project whose main objectives were:

- To evaluate the merits of using diverse (or n-version) software.
- To evaluate the computer-based specification language "X."
- To compare the effects of representative high-level and low-level languages on productivity and reliability.

In addition, there was a secondary objective to monitor the software development process to produce three diverse programs to the same requirements. The requirement was for a reactor over-power protection (trip) system. Diversity was ensured by having three independent teams to produce the software, using different specification methods (formal and informal) and different implementation languages (assembly language and Fortran). This also allowed the comparison of specification methods and programming languages to be made. After careful independent development and testing, the three programs were tested against each other in a special test harness to locate residual faults. All phases of the project were carefully documented for subsequent analysis.

The major conclusions for this particular project were that:

- Diverse software with majority voting failed less frequently than any individual program, but some common faults did exist at the end of normal software development.
- Testing diverse programs "back-to-back" proved to be a powerful method of detecting residual faults.
- The residual faults were all related to the specification of requirements, and hence, the requirement specification was the only known cause of common mode failure.

[Bjor87] Abstract: We propose a total framework for the software development stages of specification (definition), design and coding. This framework is based on three cornerstones: (a) the concept of software development graphs which specify all the stages and steps of development; (b) the use of formal methods, in [the authors] case VDM, the Vienna Software Development Method, in all stages and steps of development; and (c) the clearly separate roles of theoretical computer scientists, programmers, software engineers, and development managers in all aspects of software development. Thus not only programming is formalised (i.e., the entire programming itself is also considered a formal object about which to reason).

[Blac81] Abstract: The addition of redundancy to data structures can be used to improve the ability of a software system to detect and correct errors, and to continue to operate according to its specifications. A case study is presented which indicates how such redundancy can be deployed and exploited at reasonable cost to improve software fault tolerance. Experimental results are reported for the small data base system considered.

[Blai71] Abstract: The Purdue Extendable Debugging system (PEBUG) is a general purpose debugging system which operates under the Purdue version of the Mace operating system on the CDC 6500. PEBUG is designed to

provide flexible debugging in either an interactive or non-interactive mode. The basic construction of PEBUG primitives, debugging commands, and the interface used for extension, are described.

[Blai85a] Abstract: This paper presents the results of a study of the software complexity characteristics of a large real-time signal processing system for which there is a 6-yr maintenance history. The objective of the study was to compare values generated by software metrics to the maintenance history in order to determine which software complexity metrics would be most useful for estimating maintenance effort. The metrics that were analyzed were program size measures, software science measures, and control flow measures. During the course of the study two new software metrics were defined. The new metrics, maximum knot depth and knots per jump ratio, are both extensions of the knot count metric. When comparing the metrics to the maintenance data the control flow measures showed the strongest positive correlation.

[Bloo86] Abstract: The increasing use of computers to protect or control potentially hazardous processes leads to a need for effective methods to assess the software they execute. This correspondence presents a case study in which the Vienna development method (VDM), a formal specification and development methodology, was used during the analysis phase of the assessment of a prototype nuclear reactor protection system. The VDM specification was also translated into the logic language Prolog to animate the specification and to provide a diverse implementation for use in back-to-back testing. The authors claim that this technique provides a visible and effective method of analysis which is superior to the informal alternatives.

[Blum75] Abstract: Intelligence tests occasionally require the extrapolation of an effective sequence (e.g. 1661, 2552, 3663, ...) that is produced by some easily discernible algorithm. In this paper, we investigate the theoretical capabilities and limitations of a computer to infer such sequences. We design Turing machines that in principle are extremely powerful for this purpose and place upper bounds on the capabilities of machines that would do better.

[Boch87b] Abstract: The use of formal specifications in software development allows the use of certain automated tools during the specification and software development process. Formal description techniques have been developed for the specification of communication protocols and services. This paper describes the partial automation of the protocol implementation process based on a formal specification of the protocol to be implemented. An implementation strategy and a related software structure for the implementation of state transition oriented specifications is presented. Its application is demonstrated with a much simplified Transport protocol. The automated translation of specifications into implementation code in a high-level language is also discussed. A semiautomated implementation strategy is explained which highlights several refinement steps, part of which are automated, which lead from a formal protocol specification to an implementation. Experience with several full implementations of the OSI Transport protocol is described.

[Boeh75a] Abstract: This paper summarizes some recent experience in analyzing and eliminating sources of error in the design phase of large software projects. It begins by pointing out some of the significant differences in software error incidence between large and small software projects. The most striking contrast, illustrated by project data, is the large preponderance of design errors over coding errors on large scale projects, not only with respect to numbers of errors, but also with respect to the relative time and effort required to detect them and correct them.

The paper next presents a taxonomy of software error causes, and some analyses of the design error data, performed to obtain a better understanding of the nature of large-scale software design errors and to evaluate alternative methods of preventing, detecting, and eliminating them.

Based on this analysis of observational data, a hypothesis was derived regarding the potential cost-effectiveness of an automated aid to detecting inconsistencies between assertions about the nature of inputs and outputs of the various elements (functions, modules, data bases, data sources, etc.) of the software design. This hypothesis was tested by developing a prototype version of such an aid, the Design Assertion Consistency Checker (DACC), using TRW's Generalized Information Management (GIM) System, and using it on a

large-scale software project with 186 elements and 967 assertions about their inputs and outputs.

Of the 121,000 possible mismatches between input and output assertions, DACC found 818, at a cost in computer time of $30. Most of the mismatches resulted from shortfalls in the initial version of DACC or the initial data preparation, such as lack of a synonym capability and a lack of explicit statements about external inputs and outputs. However, a number of serious mismatches were exposed at a time when they were easy to correct, and a most useful worklist generated of items needing resolution before allowing the design effort to proceed to further detail.

In general, the data confirmed the hypothesis about the general utility of a DACC capability for large software projects. However, a number of additional features should be considered to compensate for current deficiencies (in areas such as manuscript preparation) and to fully take advantage of having the software design in machine-readable form.

[Boeh75b] Introduction: The high cost of software should be considered more of an opportunity than a problem. Nobody can say for sure to what extent software "costs too much." However, the high cost implies that additional improvements will lead to significant savings, which should justify additional investments to streamline some of the institutions, techniques, and procedures which often hinder software productivity.

This chapter will address three main questions: How high is the cost of software? Where do costs go? What factors influence costs? (or, what can we do about them?)

For reference, "software production" here includes all the effort involved in producing and maintaining the necessary executive, support, and applications programs and their documentation, starting from a reasonably well-defined functional specification. Most of the software data comes from the Air Force, primarily from the CCIP-85 study and the recent Air Force-Industry Software Cost Workshop, but they are probably fairly representative of other software activities elsewhere.

[Boeh78] Abstract: The study reported in this paper establishes a conceptual framework and some key initial results in the analysis of the characteristics of software quality. Its main results and conclusions are:

- Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs.
- The current software state-of-the-art imposes specific limitations on our ability to automatically and quantitatively evaluate the quality of software.
- A definitive hierarchy of well-defined, well-differentiated characteristics of software quality is developed. Its higher-level structure reflects the actual uses to which software quality evaluation would be put; its lower-level characteristics are closely correlated with actual software metric evaluations which can be performed.
- A large number of software quality-evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation.
- Particular software life-cycle activities have been identified which have significant leverage on software quality.

Most importantly, we believe that the study reported in this paper provides for the first time a clear, well-defined framework for assessing the often slippery issues associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experiences cited. This framework is certainly not complete, but it has been brought to a point sufficient to serve as a viable basis for future refinements and extensions.

The bulk of the work reported in this book was performed in a study by TRW for the National Bureau of Standards in 1973. The book presents this original material and subsequent updates in the following order:

- A preface which introduces, summarizes, and updates the 1973 study;
- The text of the 1973 study;
- A revised and updated version of the annotated bibliography prepared for the 1973 study.

[Boeh81] Abbreviated Preface: A course in engineering economics has become a fairly standard component of the hardware engineer's education. So far, the opportunities for software engineers to take a similar course tailored to software engineering economics have been rare. As a result, [the author thinks] most software

engineers miss out on a chance to acquire and use a number of significant economic concepts, techniques, and facts which can play a vital part in their future careers–and a vital part in making our software easier to live with and more worthwhile.

Not surprisingly, then, the major objective of this book is to provide a basis for a software engineering economics course, intended to be taken at the college senior/first-year graduate level.

[Boeh84a] Abstract: In this experiment, seven software teams developed versions of the same small-size (2000-4000 source instruction) application software product. Four teams used the Specifying approach. These teams used the Prototyping approach.

The main results of the experiment were the following.

1. Prototyping yielded products with roughly equivalent performance, but with about 40 percent less code and 45 percent less effort.
2. The prototyped products rated somewhat lower on functionality and robustness, but higher on ease of use and ease of learning.
3. Specifying produced more coherent designs and software that was easier to integrated.

The paper presents the experimental data supporting these and a number of additional conclusions.

[Boeh84b] Introduction: A major effort at improving productivity at TRW led to the creation of the software productivity project, or SPP, in 1981. The major thrust of this project is the establishment of a software development environment to support project activities; this environment is called the software productivity system, or SPS. It involves a set of strategies, including the work environment; the evaluation and procurement of hardware equipment; the provision for immediate access to computing resources through local area networks; the building of an integrated set of tools to support the software development life cycle and all project personnel; and a user support function to transfer new technology. All of these strategies are being accomplished incrementally. The current architecture is Vax-based and uses the Unix operating system, a wideband local network, and a set of software tools.

This article describes the steps that led to the creation of the SPP, summarizes the requirements analyses on which the SPS is based, describes the components which make up the SPS, and presents our conclusions.

[Boeh86] Overview: The spiral model of software development and enhancement presented here provides a new framework for guiding the software process. Its major distinguishing feature is that it creates a *risk-driven* approach to the software process, rather than a strictly specification-driven or prototype-driven process. It incorporates many of the strengths of other models, while resolving many of their difficulties.

This section presents a short historical background of software process models and the issues they address. Section 2 summarizes the process steps involved in the spiral model. Section 3 illustrates the application of the spiral model to a software project, using the TRW Software Productivity Project as an example. Section 4 summarizes the primary advantages, challenges, and implications involved in using the spiral model, and Section 5 presents the resulting conclusions.

[Boeh87] Summary: A candidate top 10 list of software metric relationships, in terms of their value in industrial situations. Here they are, in rough priority order:

1. Finding and fixing a software problem after delivery is 100 times more expensive than finding and fixing it during the requirements and early design phases.
2. You can compress a software development schedule up to 25 percent of nominal, but no more.
3. For every dollar you spend on software development you will spend two dollars on software maintenance.
4. Software development and maintenance costs are primarily a function of the number of source instructions in the product.
5. Variations between people account for the biggest differences in software productivity.
6. The overall ratio of computer software to hardware costs has gone from 15:85 in 1955 to 85:15 in 1985, and it is still growing.

7. Only about 15 percent of software product-development effort is devoted to programming.
8. Software systems and software products each typically cost three times as much per instruction to fully develop as does an individual software program. Software-system products cost nine times as much.
9. Walkthroughs catch 60 percent of the errors.
10. Many software phenomena follow a Pareto distribution: 80 percent of the contribution comes from 20 percent of the contributors. Some examples: 20 percent of the modules contribute 80 percent of the cost, and 20 percent of the modules contribute 80 percent of the errors (not necessarily the same ones).

[Boot80] Abstract: The concept of abstract data types is extended to associate performance information with each abstract data type representation. The resulting performance abstract data type contains a functional part which described the functional properties of the data type and a performance part which describes the performance characteristics of the data type. The performance part depends upon 1) the algorithms and data representation selected to represent the data type, 2) the particular machine on which the software realization of the data type is realized, and 3) the statistical properties of the actual data represented by the data objects involved in the data type. Methods for determining the necessary information to specify the performance part of the representation are discussed.

[Boro72] Abstract: Some consequences of the Blum axioms for step counting functions are investigated. Complexity classes of recursive functions are introduced analogous to the Hartmanis-Stearns classes of recursive sequences. Arbitrarily large "gaps" are shown to occur throughout any complexity hierarchy.

[Boug86] Abstract: [The authors] present a method and a tool for generating test sets from algebraic data type specifications. [The authors] give formal definitions of the basic concepts required in our approach of functional testing. Then [the authors] discuss the problem of testing algebraic data types implementations. This allows the introduction of additional hypotheses and thus the description of the method is based on logic programming. Some limitations of PROLOG are discussed and two extensions are presented, METALOG and SLOG, which allow good implementations of [the authors] method.

[Bowe79] Abbreviated Introduction: This article addresses the integration and test phase by surveying military standards for software quality, proposed quality metrics, and techniques that evaluate the readiness of software for acceptance testing. Some of the techniques discussed, such as providing test result visibility to the customer, test effectiveness, and regression testing, apply to any software life-cycle phase.

[Bowe80] Summary: A standard software error classification is viable based on experimental use of different schemes on Hughes-Fullerton projects. Error classification schemes have proliferated independently due to varied emphasis on depth of causal traceability and when error data was collected. A standard classification is proposed that can be applied to all phases of software development. It includes a major causal category for design errors. Software error classification is a prerequisite for both feedback for error prevention and detection, and for prediction of residual errors in operational software.

[Bowe83] Abstract: Software metrics (or measurements) which are used to indicate and predict levels of software quality were extended from previous research to include considerations for distributed computing systems. Aspects of the products of software life-cycle activities which could affect the quality levels of software, and metrics to measure them, were identified. Two new quality factors, survivability and expandability, were validated. A guidebook for Software Quality Measurement was produced to aid in setting quality goals, applying metric measurements, and making quality level assessments. New metrics for interoperability and reusability were also included in the guidebook.

[Bowe85] Abbreviated Preface: The purpose of this contract was to (1) consolidate results of previous RADC contracts dealing with software quality measurement, (2) enhance the software quality framework, and (3) develop a methodology to enable a software acquisition manager to determine and specify software quality

factors requirements. We developed the methodology and framework elements to focus on an Air Force software acquisition manager specifying quality requirements for embedded software that is part of a command and control application. This methodology and most of the framework elements are generally useful for other applications and different environments. The Final Technical Report consists of three volumes:

- Volume I, Specification of Software Quality Attributes - Final Report.
- Volume II, Specification of Software Quality Attributes - Software Quality Specification Guidebook.
- Volume III, Specification of Software Quality Attributes - Software Quality Evaluation Guidebook.

Volume I describes the results of research efforts conducted under this contract, including recommendations for integrating quality metrics technology into the Air Force software acquisition management process, recommended changes to Air Force software acquisition documentation, and summaries of software quality framework changes and specification methodology features.

Volumes II and III describe the methodology for using the quality metrics technology and include an overview of the software acquisition process using this technology and the quality framework. Volume II describes methods for specifying software quality requirements and addresses the needs of the software acquisition manager. Volume III describes methods for evaluating achieved quality levels of software products and describes the needs of data collection and analysis personnel.

Volume II also describes procedures and techniques for specifying software quality requirements in terms of quality factors and criteria. Factor interrelationships, relative costs to develop high quality levels, and an example for a command and control application are described. Procedures for assessing compliance with specified requirements are included.

Volume III also describes procedures and techniques for evaluating achieved quality levels of software products. Worksheets for collecting metric data by software life-cycle phase and scoresheets for scoring each factor are provided in the appendices. Detailed metric questions on worksheets are nearly identical to questions in the Software Evaluation Reports proposed as part of the STARS measurement data item descriptions.

[Boye75] Abstract: SELECT is an experimental system for assisting in the formal systematic debugging of programs. It is intended to be a compromise between an automated program proving system and the current ad hoc debugging practice, and is similar to a system being developed by King et al of IBM[2]. SELECT systematically handles the paths of programs written in a LISP subset that includes arrays. For each execution path SELECT returns simplified conditions on input variables that cause the path to be executed, and simplified symbolic values for program variables at the path output. For conditions which form a system of linear equalities and inequalities SELECT will return the input variable values that can serve as sample test data. The user can insert constraint conditions, at any point in the program including the output, in the form of symbolically executable assertions. These conditions can induce the system to select test data in user-specified regions. SELECT can also determine if the path is correct with respect to an output assertion. We present four examples demonstrating the various modes of system operation and their effectiveness in finding bugs. In some examples, SELECT was successful in automatically finding useful test data. In others, user interaction was required in the form of output assertions. SELECT appears to be a useful tool for rapidly revealing program errors, but for the future there is a need to expand its expressive and deductive power.

[Boye79] Abbreviated Preface: This book is a user's guide to a computational logic. A "computational logic" is a mathematical logic that is both oriented towards discussion of computation and mechanized so that proofs can be checked by computation. The computational logic discussed in this handbook is that developed by Boyer and Moore.

This handbook contains a precise and complete description of our logic and a detailed reference guide to the associated mechanical theorem proving system. In addition, the handbook includes a primer for the logic as

2. IBM is a registered trademark of International Business Machines Corp.

a functional programming language, an introduction to proofs in the logic, a primer for the mechanical theorem prover, stylistic advice on how to use the logic and theorem prover effectively, and many examples.

The logic was last described completely in our book *A Computational Logic*, published in 1979. In the eight years since [this book] was published, the logic and the theorem prover have changed. On two occasions we changed the logic, both times concerned with the problem of axiomatizing an interpreter for the logic as a function in the logic but motivated by different applications.

There have been two truly important changes to the theorem prover since 1979, neither of which has to do with additions to the logic. One was the integration of a linear arithmetic decision procedure. The other was the addition of a rather primitive facility permitting the user to give hints to the theorem prover.

The most important changes have occurred not in the logic or the code but it our understanding and use of them. The most impressive number theoretic result proved in 1979 was the existence and uniqueness of prime factorizations; it is now Gauss's law of quadratic reciprocity. The most impressive metamathematical result was the soundness and completeness of a propositional calculus decision procedure; it is now *Gödel's* incompleteness theorem. These results are not isolated peaks on a plain but just the highest ones in ranges explored with the system.

[Boye80] Abstract: This note discusses two theorem-proving questions that received substantial discussion during the Workshop. It does not pretend to be a thorough or impartial summary of every significant theorem-proving issue raised.

[Boye84a] Abstract: This article consists of three parts: a tutorial introduction to a computer program which proves theorems by induction; a brief description of recent applications of that theorem-prover; and a discussion of several nontechnical aspects of the problem of building automatic theorem-provers. The theorem-prover described has proven theorems such as the uniqueness of prime factorisations, Fermat's theorem and the recursive unsolvability of the halting problem.

[Bran80] Abstract: Guidelines are given for program testing and verification to insure quality software for the programmer working alone in a computing environment with limited resources. The emphasis is on verification as an integral part of the software development. Guidance includes developing and planning testing as well as the application of other verification techniques at each lifecycle stage. Relying upon neither automated tools or formal quality assurance support, the guidelines should be appropriate for applications programmers doing small development projects.

[Brin73] Summary: A central problem in program design is to structure a large program such that it can be tested systematically by the simplest possible techniques. This paper describes the method used to test the RC 4000 multiprogramming system. During testing, the system records all transitions of processes and messages between various queues. The test mechanism consists of fifty machine instructions centralized in two procedures. By using this mechanism in a series of carefully selected test cases, the system was made virtually error free within a few weeks. The test procedure is illustrated by examples.

[Brin78] Summary: This paper describes a systematic method for testing monitor modules which control process interactions in concurrent programs. A monitor is tested by executing a concurrent program in which the processes are synchronized by a clock to make the sequence of interactions reproducible. The method separates the construction and implementation of test cases and makes the analysis of a concurrent experiment similar to the analysis of a sequential program. The implementation of a test program is almost mechanical. The method, which is illustrated by an example, has been successfully used to test a multicomputer network program written in Concurrent Pascal.

[Brin85] Abstract: Now that several Ada compilers and interpreters have been validated, increased attention is being given to Ada Programming Support Environments and the tools needed for Ada program development. This paper discusses the capabilities needed in an Ada debugger in light of the language's tasking constructs, and

presents the design for a debugger which operates in concert with a single-processor Ada interpreter. This debugger design demonstrates the extensions to sequential debugging techniques that are necessary to handle concurrency, and shows that significant debugging functionality can be provided even without the inclusion of automatic error diagnosis methods. The issues considered here include isolation of effects and display of the full dynamic execution status, both of which are essential to diagnosis of concurrent programs.

[Brit88] Conclusion: As we develop better tools for recording and compiling software designs and code, those who think about and practice programming will take greater interest in the more obscure aspects of a program: its intent, meaning, resilience, and developmental history. Although the problem of writing correct programs, especially those embedded within large systems or products, remains largely unsolved in practice, the situation is improving. We can use inspections to further the investigation into how correct programs are constructed. Several such inspections will be carried out to determine their usefulness and refine their practice. The purpose of incorporating correctness arguments into inspections is not to improve inspections, but to improve programming. This is not a modest objective. Steps will necessarily be small.

[Broo75] Abbreviated Preface: In many ways, managing a large computer programming project is like managing any other large undertaking-in more ways than most programmers believe. But in many other ways it is different-in more ways than most professional managers expect.

Managing OS/360 development was a very educational experience, albeit a very frustrating one. The team, including F.M. Trapnell who succeeded [the author] as manager, has much to be proud of. The system contains many excellencies in design and execution, and it has been successful in achieving widespread use. Certain ideas, most noticeably device-independent input-output and external library management, were technical innovations now widely copied. It is now quite reliable, reasonably efficient, and very versatile. The effort cannot be called wholly successful, however. The flaws in design and execution pervade especially the control program, as distinguished from the language compilers. Furthermore, the product was late, it took more memory than planned, the costs were several times the estimate, and it did not perform very well until several releases after the first.

After leaving IBM in 1965, [the author] began to analyze the OS/360 experience to see what management and technical lessons were to be learned. In particular, [the author] wanted to explain the quite different management experiences encountered in System/360 hardware development and OS/360 software development.

My own conclusions are embodied in the essays that follow, which are intended for professional programmers, professional managers, and especially professional managers of programmers.

Although written as separable essays, there is a central argument contained especially in Chapters 2-7. Briefly, [the author] believe that large programming projects suffer management problems different in kind from small ones, due to division of labor. [The author] believes the critical need to be the preservation of the conceptual integrity of the product itself. These chapters explore both the difficulties of achieving this unity and methods for doing so. The later chapters explore other aspects of software engineering management.

[Broo80a] Abstract: The application of behavioral or psychological techniques to the evaluation of programming languages and techniques is an approach which has found increased applicability over the past decade. In order to use this approach successfully, investigators must pay close attention to methodological issues, both in order to insure the generalizability of their findings and to defend the quality of their work to researchers in other fields. Three major areas of methodological concern, the selection of subjects, materials, and measures, are reviewed. The first two of these areas continue to present major difficulties for this type of research.

[Broo81] Introduction: A statistical analysis was conducted of structured programming and programmer performance, with productivity measured as lines of code per man-month. The study findings support the following productivity hypotheses: (1) Increasing the complexity of programming projects tends to lower productivity. (2) The use of structured programming results in increased productivity. (3) Structured programming technology has the highest payoff for severely constrained complex projects, the improvement ranging from 200% to over 600% as compared to similar projects using conventional technology. The study tends to rule out the possibility that the following factors could be responsible for the higher productivity of projects using structured programming: (1)

the application of structured programming only to less complex, less constrained projects that would be likely to exhibit productivity anyway; (2) the assignment of more experienced programmers, who would probably exhibit higher productivity, to projects using structured programming; and (3) a "code explosion" (an increase in the number of lines of code produced due to the tabular format of structured programs).

[Brop87] Abstract: As Ada is introduced into new environments, both managers and developers need to understand the ways in which the decision to use Ada as the target language will affect the software development lifecycle. The Flight Dynamics division at NASA Goddard Space Flight Center is involved in a study analyzing the effects of Ada on the development of their software. This project is one of the first to use Ada in this environment. In the study, two teams are each developing satellite simulators from the same specifications, one in Ada and one in FORTRAN, the standard language in this environment. This paper will address the lessons learned during the design phase including the effect of specifications on Ada-oriented design, the importance of the documentation style for the chosen design method, and the effects of Ada-oriented design on the software development lifecycle. It is hoped that the issues faced in this project will show more clearly what may be expected in designing with Ada-oriented design methods.

[Brow72a] Abbreviated Introduction: From the point of the user, a reliable computer program is one which performs satisfactorily according to the computer program's specifications. The ability to determine if a computer program does indeed satisfy its specifications is most often based upon accumulated experience in using the software. This is due in part to general agreement that the quality of computer software increases as the software is extensively used and failures are discovered and corrected. In keeping with this philosophy, increasing emphasis has been placed on exhaustive testing computer programs as the principal means of assuring sufficient quality.

Nevertheless, a significant problem which pervades all software development is a lack of knowledge as to how much testing of a software system or component constitutes sufficient verification. As a result, we often lack sufficient confidence that the software will continue to operate successfully for unanticipated combinations of data in a real-world environment.

In recognition of the high cost and uncertainty of software verification, TRW Systems' Product Assurance Office initiated a company-funded effort to improve upon current testing methodology. The result of the study, experimentation, design and development thus far conducted comprises the TRW Product Assurance Confidence Evaluator (PACE) system, an evolving collection of automated tools which provide support in various phases of software testing.

The initial PACE instance was the FLOW program to support test evaluation activities. FLOW monitors statement usage during test execution, thus providing a basic evaluation of test effectiveness. In addition, FLOW supports the test planning activity by indicating the unexercised code, and consequently, the additional tests required for more comprehensive testing.

[Brow75] Abstract: This paper presents a formulation of a novel methodology for evaluation of testing in support of operational reliability assessment and prediction. The methodology features an incremental evaluation of the representativeness of a set of development and validation test cases together with definition of additional test cases to enhance those qualities.

If test cases are derived in typical fashion (i.e., to find and remove bugs, to investigate software performance under off-nominal conditions, to exercise structural elements and functional capabilities of the software, and to demonstrate satisfaction of software requirements), then the complete set of test cases is not necessarily representative of anticipated operational usage. The paper reports on initial research into formulation of valid measures of testing representativeness.

Several techniques which permit specification of expected operational usage are described, and a technique for evaluating the correlation between actual testing accomplished and expected operational usage is defined. An unbiased estimator for operational usage reliability is proposed and justified as a function of a specified operational profile; confidence in the estimate is derived from a measure of the degree to which testing is representative of expected operational application.

An experimental application of the techniques to a small program is provided as an illustration of the proposed use of the methodology for operational software reliability estimation. The relationship between structural exercise testing thoroughness and operational usage representativeness is discussed; the specification of a quantified reliability requirement and an explicit, required representativeness measure (or confidence) is identified as integral to effective application of the proposed reliability testing methodology; efforts to extend, formalize and generalize the methodology are described; and expected benefits, as well as potential problems and limitations are identified.

[Brow78] Abstract: FAST (Fortran Analysis System) implements a powerful set of analysis capabilities on Fortran source language programs. Its implementation was accomplished through the integration of existing software systems and by the use of modern language system development tools. The result is an order of magnitude reduction in effort of implementation coupled with a sizable increase in system capabilities. The use of a general purpose, commercially available data management system as a data handler and data correlator is a dominant factor in both reduction of effort of implementation and generation of additional power and flexibility in the analysis capabilities for systematically qualified program analyses which is unique among existing program analyzers. This capability should be particularly useful in the program maintenance environment.

[Brow89] Abstract: A probabilistic model is presented which determines the optimal number of software test cases required in situations where the following can be estimated as independent parameters: 1) the cost per test, 2) the cost per error if undetected until field implementation, 3) the number of software executions over its lifetime, 4) the number of possible different executions, and 5) the number of faults embedded in the software. A formula is derived by the use of calculus which is solved by approximation techniques. Tables of optimal number of tests over a range of parameter values are presented to illustrate the results. The model serves as a basis to crystallize further research efforts to improve the accuracy of input variable estimation.

[Brue83] Abstract: This paper introduces a modified version of path expressions called Path Rules which can be used as a debugging mechanism to monitor the dynamic behavior of a computation. Path rules have been implemented in a remote symbolic debugger running on the Three Rivers Computer Corporation PERQ computer under the Accent operating system.

[Brya80] Abstract: This paper discusses the application of software product assurance to actual on-going projects. Several facets of software product assurance are presented in terms of their application to real-life situations. The performance of product assurance usually enhances product integrity. This benefit is obtained to a lesser or greater degree regardless of when product assurance is first introduced in a project.

[Bryk89] Preface: The purpose of IDA Memorandum M-496, *Bibliography of Testing and Evaluation Reference Material*, is to present the reference material acquired in the course of developing IDA Paper P-2132, *SDS Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation With Recommended R&D Tasks*. This document was prepared for the Strategic Defense Initiative Organization (SDIO).

[Buck79] Abstract: The increasing criticality of software mandates a standard for software quality assurance plans. Such a standard, developed by the Computer Society Software Engineering Standards Subcommittee, appears here.

[Budd78a] Introduction: When testing software the major question which must always be addressed is "If a program is correct for a finite number of test cases, can we assume it is correct in general." Test data which possess this property is called Adequate test data, and, although adequate test data cannot in general be derived algorithmically, several methods have recently emerged which allow one to gain confidence in one's test data adequacy.

Program mutation is a radically new approach to determining test data adequacy which hold promise of being a major breakthrough in the field of software testing. The concepts and philosophy of program mutation

have been given elsewhere, the following will merely present a brief introduction to the ideas underlying the system.

Unlike previous work, program mutation assumes that competent programmers will produce programs which, if they are not correct, are "almost" correct. That is, if a program is not correct it is a "mutant" – it differs from a correct program by simple errors. Assuming this natural premise, a program P which is correct on test data T is subjected to a series of mutant operators to produce mutant programs which differ from P in very simple ways. The mutants are then executed on T. If all mutants give incorrect results then it is very likely that P is correct (i.e., T is adequate). On the other hand, if some mutants are correct on T then either: (1) the mutants are equivalent to P, or (2) the test data T is inadequate. In the latter case, T must be augmented by examining the non-equivalent mutants which are correct on T: a procedure which forces close examination of P with respect to the mutants.

At first glance it would appear that if T is determined adequate by mutation analysis, then P might still contain some complex errors which are not explicitly mutants of P. To this end there is a COUPLING EFFECT which states that test data in which all simple mutants fail is so sensitive that it is highly likely that all complex mutants must also fail.

[Budd80a] Abbreviated Introduction: In [this paper] we will present two types of theoretical results concerning the questions: 1) If a program P is written by a competent programmer and if P passes the ϕ mutant test with test data D, does the function actually computed by P equal the partial recursive function that specifies the intended behavior of P? 2) (Coupling Effect): If P passes the subset of μ mutant test with data D, does P pass the ϕ mutant test with data D? General results are expressed in terms of properties of the language class L, and specific results for a class of decision table programs and for a subset of LISP. Portions of the work on decision tables and LISP have appeared elsewhere, but the presentations given here are both simpler and more unified. In the final section we present a system for applying program mutation to FORTRAN and we introduce a new type of software experiment, called a "beat the system" experiment, for evaluating how well our system approximates an affirmative response to the program mutation questions.

[Budd80d] Abstract: We consider two interpretations for what it means for test data to demonstrate correctness. For each interpretation, we examine under what conditions data sufficient to demonstrate correctness exists, and whether it can be automatically detected and/or generated. We establish the relation between these questions and the problem of deciding equivalence of two programs.

[Budd85] Abstract: Both theoretical and empirical arguments suggest that specifications and implementations are equally important sources of information for generating test cases. Nevertheless, the majority of test generation procedures described in the literature deal only with the program source, ignoring specifications. In this paper we outline a procedure for measuring test case effectiveness using specifications given in predicate calculus form. This method is similar to the mutation analysis method of testing programs.

[Bunc80] Abstract: An approach to analyzing the interaction of hardware failure modes with computer software is described. The approach considers the software requirements, not the design or implementation and is an extension of the FMEA (failure mode and effects analysis) discipline. It has been developed to address the needs of the Space Shuttle Orbiter Project and is being applied to Orbiter subsystems. The basic approach is applicable to other hardware/software systems, and guidelines for its application are presented.

[Burs74] Abstract: A method of proving facts about programs is presented in an informal manner, in the hope that it will have some intuitive appeal to programmers. It derives essentially from Manna's method, but it is influenced by the recent idea of "executing" a program symbolically as part of the proof process. Some examples are worked out, including one to invert a permutation *in situ* and one to traverse a tree; the latter seems to come out rather easily this way. Finally this technique and the Floyd one are related to a system of modal logic.

[Cail79] Comment: In his article "A Controlled Experiment in Program Testing and Code

Walkthroughs/Inspections," Glenford J. Myers states that the overall results showing people's ability to find errors are rather dismal. Dr. Myers does not indicate whether any of the 59 participants complained about the program's style and the tricks it employs. It this were not the case, then not only [is the author] not surprised at the result, but [he is] also quite shocked - it would show the participants' meek acceptance of tricks and bad style as "normal."

One may wonder if it is not less painful to sit down and code the program anew and more reliably in the time spent on tracking down the errors. That experiment has not been done, and it would be interesting to know its results.

It is also not obvious whether some of the errors listed ought not, in fact, to be classified as omissions from the specification; for example, from the specification it follows that a tab character is not a break character, and nowhere is there any mention of how many printing positions an output line may occupy. Clearly the fact that a tab is a single character and the fact that it occupies a number of printing positions on a certain device are two different things. A formatting program is especially sensitive to the character set and to the effects characters have on the output device.

[**Camp79**] **Abstract:** This paper describes the enhancement of Pascal to specify synchronization between concurrent processes by path expressions. The extended language is being used to gain experience in the design and construction of practical real-time systems and operating systems. An encapsulation mechanism is included to synchronize all accesses to encapsulate data. A network message transfer system is presented as an extended example of the use of path expressions.

[**Card86a**] **Abstract:** Software engineers have developed a large body of software design theory and folklore, much of which has never been validated. This paper reports the results of an empirical study of software design practices in one specific environment. The practices examined affect module size, module strength, data coupling, descendant span, unreferenced variables, and software reuse. Measures characteristic of these practices were extracted from 887 Fortran modules developed for five flight dynamics software projects monitored by the Software Engineering Laboratory. The relationship of these measures to cost and fault rate was analyzed using a contingency table procedure. The results show that some recommended design practices, despite their intuitive appeal are ineffective in this environment, whereas others are very effective.

[**Card87a**] **Abstract:** The theory of software science proposed by Halstead appears to provide a comprehensive model of the program construction process. Although software science has been widely criticized on theoretical grounds, its measures continue to be used because of apparently strong empirical support. This study reexamined one basic relationship proposed by the theory: that between estimated and actual program length. The results show that the apparent agreement between these quantities is a mathematic artifact. Analyses of both Halstead's own data and another larger dataset confirm this conclusion. Software science has neither a firm theoretical nor empirical foundation.

[**Card87b**] **Abstract:** Many new software development practices, tools, and techniques have been introduced in recent years. Few, however, have been empirically evaluated. The objectives of this study were to measure technology use in a production environment, develop a statistical model for evaluating the effectiveness of technologies, and evaluate the effects of some specific technologies on productivity and reliability. A carefully matched sample of 22 projects from the Software Engineering Laboratory database was studied using an analysis-of-covariance procedure. Limited use of the technologies considered in the analysis produced approximately a 30 percent increase in software reliability. These technologies did not demonstrate any direct effect on development productivity.

[**Care77**] **Abstract:** This paper presents a software testing and Quality Assurance technology based on a special set of development methodologies. A specific example employing a top-down design process is explored in depth to demonstrate traceability from requirements to system test. The peripheral advantages of this technology are also explored.

[Carp75] Abstract: DECA is a computer program which is use in conjunction with a top-down dominated design methodology. The program organizes, validates, and produces a document depicting the design of a software system. The use of DECA significantly enhances the quality of the software design. The quality of the design in turn significantly benefits the quality of the implemented software system.

[Carv88] Abbreviation: One general approach to detecting synchronization errors, called *static analysis*, is to analyze (not execute) the program to derive an approximation of the feasibility set of a concurrent program. A number of (static) analysis techniques have been developed for detecting synchronization errors. Generally, these analysis techniques derive an approximation set which is the set of syntactically possible SYN-sequences; such techniques are referred to as *syntax-based synchronization analysis* techniques.

We have developed a new approach to analyzing concurrent programs, which is to derive constraints on the feasible SYN-sequences of a concurrent program according to the program's syntactic and semantic information. These constraints, called *feasibility constraints* or (*constraints* if there is no ambiguity), show restrictions on the ordering of synchronization events allowed by the program. By using feasibility constraints, we can obtain a better approximation of the feasibility set of a concurrent program and improve the effectiveness of error detection by static analysis.

[Cava78] Abstract: Research in software metrics incorporated in a framework established for software quality measurement can potentially provide significant benefits to software quality assurance programs. The research described has been conducted by General Electric Company for the Air Force Systems Command Rome Air Development Center. The problems encountered defining software quality and the approach taken to establish a framework for the measurement of software quality are described in this paper.

[Cha88] Abstract: MURPHY is a language-independent, experimental methodology for building safety-critical, real time software, which will include an integrated tool set. Using Ada as an example, this paper presents a technique for verifying the safety of complex, real-time software using Software Fault Tree Analysis. The templates for Ada are presented along with an example of applying the technique to an Ada program. The tools in the MURPHY tool set to aid in this type of analysis are described.

[Chan73] Abstract: The notion of a program structure has inspired several authors to describe techniques for producing programs that have "good" structure. These techniques, however, do not include definitions for program structure or good structure. It is simple asserted that programs produced using these techniques either have good structure or are more likely to have good structure than programs produced without using these techniques. Instead, good structure has been characterized by certain properties. For example, the work of Dijkstra and Parnas as well as the work of Simon and Alexander, concerning complexity in systems, suggest that programs or a system of programs having good structure possess several properties.

[Chan84] Abstract: A programmer often writes and tests programs in a bottom-up manner, producing code fragment and testing each fragment on a few examples to convince himself that the program works so far. These intermediate tests are typically lost without full utilization. The objective of this project is to create a kind of information retrieval system for test cases to remedy this situation. The "program testing assistant" described herein is intended to aid BASIC-PLUS programmers during incremental program development. As in the production of any piece of software tool, issues of ease of use and user-friendliness are of main concern in our testing assistant, along with software engineering considerations such as maintainability and reliability.

[Chan85] Introduction: The importance of software testing in high-reliability, real-time applications such as telecommunications switching cannot be overemphasized. The software of these systems must meet high reliability requirements while supporting a wide range of functions.

Requirements validation depends largely on the techniques used for specification, so the augmented finite-state machine (FSM) model used to capture the external behavior of a real-time system is central to the environment. Therefore, this paper describes the validation techniques used and explores those aspects of the

specification model that facilitate test generation and execution using the behavioral description. We refer to this theme as requirements modeling for testability.

Our thesis is: although an FSM or an augmented FSM may be relatively limited in its ability to capture the whole range of practical systems' behaviors, it is adequate for real-time systems in which sequential computations dominate. The FSM model should be preferred for applications in which high reliability is a primary concern; expressive power can be traded off to ensure quality.

[Chan89] Abbreviated Introduction: System performance concerns most system analysts. To model performance, you need a practical modeling method tailored to your environment. Existing modeling methods, which can model either queuing behavior or asynchronous concurrent behavior, are inadequate for today's complex systems. Many applications, including multiprocessor operating systems and distributed systems, require both modeling capabilities.

We present an approach that uses an enhanced model based on two familiar modeling methods, the queuing network and petri net. Our approach includes a graphical modeling tool, called TPQN, a textual specification language, called TPQL, and a simulator, called TPQS.

TPQN can represent a system with both synchronization conditions and queuing behaviors, something that cannot be modeled with either the petri net or queuing network alone. We illustrate TPQN's capabilities with a performance analysis of a real-time multitasking scheduler that had been previously implemented on top of SunOS on a Sun-3 workstation.

We have conducted some experiments with the TPQS simulator on a Sun-3. The results have helped us to analyze the system's performance and validate our modeling tool.

[Chap79] Introduction: Intuition and common sense generally agree that software which appears simple is superior to software that appears complex, whatever the inherent complexity of the job. This position is in fact incorporated in the appraisal guidelines of structured design as "simplicity." But applying intuition and common sense is not really sufficient to obtain consistently simple software. What is needed is objective, quantitative, reliable, valid and convenient ways of measuring either the complexity or the simplicity in software. To that end, a number of proposals have been advanced.

This paper proposes an alternative measure of software complexity. The background of the measure is briefly given, and its computational procedure described. Then it is applied to a given software design of a small modular structured program. Afterward, the measure is compared with other alternative measures and with programmer ratings of the program. The paper closes with a discussion of the validity of the proposed measure of software complexity.

[Chap82] Abstract: This paper describes the design and implementation of a program testing assistant which aids a programmer in the definition, execution, and modification of test cases during incremental program development. The testing assistant helps in the interactive definition of test cases and executes them automatically when appropriate. It modifies test cases to preserve their usefulness when the program they test undergoes certain types of design changes. The testing assistant acts as a fully integrated part of the programming environment and cooperates with existing programming tools such as a display editor, compiler, interpreter, and debugger.

[Chea79] Abstract: Symbolic evaluation is a form of static program analysis in which symbolic expressions are used to denote the values of program variables and computations. It does not require the user to specify which path at a conditional branch to follow nor how many cycles of a loop to consider. Instead, a symbolic evaluator uses conditional expressions to represent the uncertainty that arises from branching and develops and attempts to solve recurrence relations that describe the behavior of loop variables.

We describe a symbolic evaluator for part of the EL1 language, with particular emphasis on techniques for handling conditional data sharing patterns, the behavior of array variables, and the behavior of variables in loops and during procedure calls. An expression simplifier, which is the heart of the system, is described in some detail. Potential applications of the symbolic evaluator to problems in program validation, verification, and

optimization are mentioned.

[Cheh81] Abstract: Four automated specification and verification environments are surveyed and compared: HDM, FDM, Gypsy, and AFFIRM. The emphasis of the comparison is on the way these systems could be used to prove security properties of an operating system design.

[Chen78a] Abstract: This paper proposes a measure of program control complexity from an information theory viewpoint. A set of empirical data showing programmer productivity as a function of program control complexity is also presented. The data reveals a step-function-like contour to programmer productivity with increasing program control complexity.

[Chen81] Abstract: The development of quantitative measures to evaluate software development techniques is necessary if we are going to develop appropriate methodologies for software production. Data is collected by the Software Engineering Laboratory at NASA Goddard Space Flight Center on developing medium scale projects of up to ten man years effort. In this study, cluster analysis was used on this collected data and several measures are proposed. These measurements are objective, quantifiable, are the results of the methodology, and most important, seem relevant.

[Chen83] Abstract: Computations of distributed systems are extremely difficult to specify and verify using traditional techniques because the systems are inherently concurrent, asynchronous, and nondeterministic. Furthermore, computing nodes in a distributed system may be highly independent of each other, and the entire system may lack an accurate global clock.

In this paper, we develop an event-based model to specify formally the behavior (the external view) and the structure (the internal view) of distributed systems. Both control-related and data-related properties of distributed systems are specified using two fundamental relationships among events: the "precedes" relation, representing time order; and the "enables" relations, representing causualty. No assumption about the existence of a global clock is made in the specifications.

The specification technique has a rather wide range of applications. Examples from different classes of distributed systems, include communication systems, process control systems, and a distributed prime number generator, are used to demonstrate the power of the technique.

The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and the structure specification of a system. Both safety and liveness properties can be specified and verified. Furthermore, since the specification technique defines the orthogonal properties of a system separately, each of them can then be verified independently. Thus, the proof technique avoids the exponential state-explosion problem found in state-machine specification techniques.

[Cher80a] Abbreviated Introduction: As part of its current standards initiative, NBS is studying methods to ensure the quality of both software procured by the government and software developed within the government. In this paper we discuss the use of programming environments in developing and procuring quality software.

Since software quality cannot be assured using standard control methods, we propose a different approach to ensure quality. This approach rests on the thesis that quality in the software product can be achieved through control of the development process. Hence we propose the specification of software quality standards not in terms of properties of the final software product but by specifying how the product should be developed. The use of development tools and techniques with specific properties, e.g., the use of a design specification system that includes data flow and consistency analysis, would be standard for government procured software. Properties of the processes for producing requirements, design, code, and testing would be specified with software quality standards. In addition, the products produced at each development stage would be recorded in adherence to documentation standards. The various pieces of the development process when incorporated into a single system would constitute a programming environment.

[Cher80b] Abstract: This paper is oriented towards those quality control problems peculiar to the procurement

of software. We discuss the deficiencies, and possible corrections, of several current methodologies. We propose a set of software management and development tools for software quality assurance which enables better contractor-developer communication during the development. The paper also includes a discussion of how sophisticated programming environments can play a central role in procured software development and a discussion of the associated research issues.

[Cher86] Abstract: Inductive inference, the automatic synthesis of programs, bears certain ostensible relationships with program testing. For inductive inference, one must take a finite sample of the desired input/output behavior of some program and produce (synthesize) an equivalent program. In the testing paradigm, one seeks a finite sample for a function such that any program (in a given set) which computes something other than the object function differs from the object function on the finite sample. In both cases, the finite sample embodies sufficient information to isolate the desired program from all other possibilities. Techniques from inductive inference are used to investigate the theoretical limits of program testing and to provide techniques for effective program testing.

[Cher87b] Abstract: Inductive inference, the automatic synthesis of programs, bears certain ostensible relationships with program testing. For inductive inference, one must take a finite sample of the desired input/output behavior of some program and produce (synthesize) an equivalent program. In the testing paradigm, one seeks a finite sample for a function such that any program (in a given set) which computes something other than the object function differs from the object function on the finite sample. In both cases, the finite sample embodies sufficient knowledge to isolate the desired program from all other possibilities. These relationships are investigated and general recursion theoretic properties of testable sets of functions are exposed.

[Cher88] Abstract: In this paper we take an abstract set based approach to testing. With this approach we are able to discuss testing issues which are totally representation free. We develop a game theoretic approach to testing and obtain some complexity results from this approach. We develop a notion of testing in the limit and discuss alternative definitions of testing.

[Ches77] Abstract: In this paper we introduce a software design methodology in which the design is constantly being evaluated as it develops. Our thesis is that proper evaluation methods can aid a designer to make sure that i) his specifications are consistent with his intuition (or requirements); ii) the quality of his design is reasonable. Another way to view our methodology is that it helps a designer to build mock-up models of his design for evaluation before actual construction begins.

Our approach is intended to fulfill three goals:

1. To allow the execution of designs as programs either symbolically or with a specification interpreter which does the equivalent of a hand simulation.
2. To determine the performance characteristics of a design.
3. To evaluate the "quality" of a design and aid the designer in choosing alternative designs.

To reach these goals, we are developing a specification language for defining abstract models of a proposed system. Another language is being developed to document the hierarchical design process. Finally, some software tools to aid our methodology are under development, including an interpreter and some performance modeling systems.

[Choq86] Abstract: This paper deals with the generation of test data sets from algebraic data type specifications. We base ourselves on a previous work where basic concepts required in our approach of functional testing were defined and where a general method for generating test data sets was elaborated. Because of the similarity between a conditional equation and between a clause in a logic program, logic programming appears to be well adapted to implement this method and derive automatically test data from a specification. But some limitations of standard logic programming interpreters need to be alleviated. Especially, a logic interpreter handling "constraints" is necessary to apply in their whole generality some hypotheses made on an implementation under test. We study such an extension of a Prolog interpreter and explain the improvements effected. A simple example

and a realistic one are given.

[Chow78] Abstract: We propose a method of testing the correctness of control structures that can be modeled by a finite-state machine. Test results derived from the design are evaluated against the specification. No "executable" prototype is required. The method is based on a result in automata theory and can be applied to software testing. Its error-detecting capability is compared with that of other approaches. Application experience is summarized.

[Chri81] Overview: This paper provides an overview of a new approach to the measurement of software. The measurements are based on the count of operators and operands contained in a program. The measurement methodologies are consistent across programming language barriers. Practical significance is discussed, and areas are identified for additional research and validation.

[Chry78] Abstract: The purpose of this research was to examine the relationship between processing characteristics of programs and experience characteristics of programmers and program development time. The ultimate objective was to develop a technique for predicting the amount of time necessary to create a computer program. The fifteen program characteristics hypothesized as being associated with an increase in programming time required are objectively measurable from preprogramming specifications. The five programmer characteristics are experience-related and are also measurable before a programming task is begun. Nine program characteristics emerged as major influences on program development time, each associated with increased program development time. All five programmer characteristics were found to be related to reduced program development time. A multiple regression equation which contained one programmer characteristic and four program characteristics gave evidence of good predictive power for forecasting program development time.

[Chur86] Abstract: A procedure for evaluating a software prototype is presented. The need to assess the prototype itself arises from the use of prototyping to demonstrate the feasibility of a design or development strategy. The assessment procedure can also be of use in deciding whether to evolve a prototype into a complete system. The procedure consists of identifying evaluation criteria, defining alternative design approaches, and ranking the alternatives according to the criteria.

[Chus87] Abstract: A new coverage measure is proposed for efficient and effective software testing. The conventional coverage measure for branch testing has such defects as overestimation of software quality and redundant test data selection because all branches are treated equally. These problems can be avoided by paying attention to only those branches essential for path testing. That is, if one branch is executed whenever another particular branch is executed, the former branch is nonessential for path testing. This is because a path covering the latter branch also covers the former branch. Branches other than such nonessential branches will be referred to as essential branches.

A testing tool for the new measure is developed in order to discriminate essential branches from nonessential branches and to measure the coverage rate of these essential branches. By using this tool, it is ascertained that the number of essential branches is about 60 percent of all branches.

As a result, the new measure reduces software quality overestimation because the accumulative curve of the new measure to the number of executed test data is closer to linearity than that of the conventional measure. Another advantage is the prevention of redundant test data selection. It results from a 40 percent reduction in the number of branches to be monitored and is confirmed by a reasonable algorithm for test data selection. Furthermore, an efficient algorithm for redundancy elimination of a selected test data set is presented.

[Cinl75] Contents: Chapters in this book address the following topics: Probability spaces and random variables; Expectations and independence; Bernoulli processes and sums of independent random variables; Poisson processes; Markov chains; Limiting behavior and applications of Markov chains; Potentials, excessive functions, and optimal stopping of Markov chains; Markov processes; Renewal theory; and Markov renewal theory.

[Clar76b] Abstract: This paper describes a system that attempts to generate test data for programs written in ANSI Fortran. Given a path, the system symbolically executes the path and creates a set of constraints on the program's input variables. If the set of constraints is linear, linear programming techniques are employed to obtain a solution. A solution to the set of constraints is test data that will drive execution down the given path. If it can be determined that the set of constraints is inconsistent, then the given path is shown to be nonexecutable. To increase the chance of detecting some of the more common programming errors, artificial constraints are temporarily created that simulate error conditions and then attempt is made to solve each augmented set of constraints. A symbolic representation of the program's output variables in terms of the program's input variables is also created. The symbolic representation is in a human readable form the facilitates error detection as well as being a possible aid in assertion generation and automatic program documentation.

[Clar78a] Abstract: An overview of some of the current program validation techniques is given. Though a variety of such techniques exist, it is now commonly agreed that program testing is an essential part of the program development process. Two testing methodologies, functional and structural, are described and the case made for combining both methodologies. Finally, a system that aids in structural testing is described.

[Clar83b] Abstract: Program errors can be considered from two perspectives-cause and effect. The goal of program testing is to detect errors by discovering their effects, while the goal of debugging is to search for the associated cause. In this paper, we explore ways in which some of the results of testing research can be applied to the debugging process. In particular, computational testing and domain testing, which are two error-sensitive test data selection strategies, are described. Ways in which these selection strategies can be used as debugging aids are then discussed.

[Clar84] Abstract: Symbolic evaluation is a program analysis method that represents a program's computations and domain by symbolic expressions. This method has been the foundation for much of the current research on software testing. Most path selection and test data selection techniques, which are two of the primary concerns of testing research, require the information provided by symbolic evaluation. Symbolic evaluation is also employed by verification techniques. In addition to formal verification, several less rigorous verification techniques utilize the symbolic expressions created by symbolic evaluation to certify program properties.

In this paper, the general symbolic evaluation method is explained. Several path selection and test data selection techniques that utilize the information provided by symbolic evaluation are then described. Some informal verification techniques, which also employ this information, are discussed. Finally, the partition analysis method, which uses symbolic evaluation to combine both testing and verification is described.

[Clar85a] Abstract: A number of path selection testing criteria have been proposed throughout the years. Unfortunately, little work has been done on comparing these criteria. To determine what would be an effective path selection criteria for revealing errors in programs, we have undertaken an evaluation of these criteria. This paper reports on the results of our evaluation for those path selection criteria based on data flow relationships. We show how these criteria relate to each other, thereby demonstrating some of their strengths and weaknesses.

[Clar85b] Abstract: Symbolic evaluation is a program analysis method that represents a program's computations and domain by symbolic expressions. In this paper a general functional model of a program is presented first. Then, three related methods of symbolic evaluation, which create this functional description from a program, are described: path-dependent symbolic evaluation provides a representation of a specified path; dynamic symbolic evaluation, which is more restrictive but less costly than path-dependent symbolic evaluation, is a data-dependent method; and global symbolic evaluation, which is the most general yet most costly method, captures the functional behavior of an entire program when successful. All three methods have been implemented in experimental systems. Some of the major implementation concerns, which include effectively representing loops, determining path feasibility, dealing with compound data structures, and handling routine invocations, are explained. The remainder of the paper surveys the range of applications to which symbolic evaluation techniques are being applied. The current and potential role of symbolic evaluation in verification, testing, debugging,

optimization, and software development is explored.

[Clar86a] Abstract: A number of path selection criteria have been proposed throughout the years. Unfortunately, little work has been done on comparing these criteria. To determine what would be an effective path selection criterion for revealing errors in programs, we have undertaken an evaluation of these criteria. In our initial work, we compared three families of data flow path selection criteria and found that the strongest criteria in these families are incomparable and that, in two of the families, the strongest criteria fail to satisfy certain minimal coverage requirements. In this paper, we provide an overview of our previous results. We then introduce minor changes to the original criteria to assure that they each satisfy these minimal coverage requirements and show how these modified criteria relate to each other. We conclude with a discussion on directions for future work in this area.

[Clar86c] Abstract: Tools in a software development environment often manipulate objects that are instances of *attributed graphs*. Moreover, an individual attributed-graph instance may be manipulated by several different tools in an environment. During the prototyping phase in the design of a software development environment, experimentation with tools may dictate changes to the high-level structure of an attributed graph as well as changes to the graph's underlying representation. We have developed a meta-tool called GRAPHITE to facilitate both kinds of experimentation while minimizing the impact of that experimentation on the tools in the environment. This meta-tool and its potential contributions to an experimental effort to build an advanced Ada software development environment are described in this paper.

[Clar88a] Abstract: Current research indicates that software reliability needs to be achieved through the careful integration of a number of diverse testing and analysis techniques. To address this need, the Team environment has been designed to support the integration of and experimentation with an ever growing number of software testing and analysis tools. To achieve this flexibility, we exploit three design principles: component technology so that common underlying functionality is recognized; generic realizations so that these common functions can be instantiated as diversely as possible; and language independence so that tools can work on multiple languages, even allowing some tools to be applicable to different phases of the software lifecycle. The result is an environment that contains building blocks for easily constructing and experimenting with new testing and analysis techniques. Although the first prototype has just recently been implemented, we feel it demonstrates how modularity, genericity, and language independence further extensibility and integration.

[Clar88b] Introduction: It is clear from recent research that to achieve highly reliable software a number of testing techniques will need to be effectively automated and integrated together into a powerful testing system. To achieve this goal we have been pursuing two research directions. One direction has been an investigation into which testing techniques should be included in such a system. As part of this effort we have evaluated several different techniques to understand their strengths and weaknesses [Clar85a, Rich86b]. This evaluation has led to the preliminary development of a model for integrating testing techniques that appears quite promising for tackling this difficult task.

The second research direction is the design and development of a testing system that can support the integration of various testing techniques. Certainly the results of the first direction will have a major impact on the second. Many of the basic underlying capabilities of various testing techniques are the same, however. In particular, most rely upon symbolic or data flow information such as could be gathered by symbolic evaluation or data flow analysis tools. Thus, we have been able to explore the design of a testing system that would provide these basic testing capabilities as well as hold the potential for supporting the integration of more advanced techniques as work on the first research direction has progressed.

This document reports on our progress to date on designing and developing the testing system. The next section provides a high level description of the system architecture and gives a brief overview of each of the major components and their interaction. Then each ensuing section describes one component of the system and the status of our work on that component. The appendices contain the actual design documents and user manuals. The research on evaluating and integrating advanced testing techniques is described in a separate document.

[Coch50] Abbreviated Preface: Work on this book was started when [the authors] were members of the staff of Iowa State College. At that time requests were received rather frequently from research workers. Some wanted advice on the conduct of a specific experiment: others, who had decided to use one of the more complex designs that have been discovered in recent years, asked for a plan or layout that could be followed during the experimental operations. Although the logical principles governing the subject of experimentation are admirably expounded in Fisher's book *The Design of Experiments*, these requests indicated a need for a different type of book, one which would describe in some detail the most useful of the designs that have been developed, with accompanying plans and an account of the experimental situations for which each design is most suitable.

[Coch53] Abbreviated Preface: This book was developed from a course of lectures on sample survey techniques. The purpose of the book is to present a reasonably comprehensive account of sampling theory as it has been developed for use in sample surveys, with sufficient illustrations to show how the theory is applied in practice, and with a supply of exercises to be worked by the student.

[Cohe77] Summary: This paper describes a language for studying the behaviour of programs, based upon the data collected while these programs are executed by a computer. Besides being a useful tool in debugging, the language is also valuable in the experimental evaluation of the complexity of algorithms, in studying the interdependence of conditionals in a program and in determining the feasibility of transporting programs from one machine to another. The program one wishes to analyse is written in an Algol 60-like language; when the program is executed it automatically stores, in a data base, the information needed to answer general questions about computational events which occurred during execution. This information consists (basically) of the list of labels passed while the program is being executed, and the current values of the variables. Since the list of labels is describable by regular expressions, these expressions can also be used to identify specific subparts of the list and therefore allow access to the values of the variables. This constitutes the basis for the design of the inquiry language. The user's questions are automatically answered by a processor which inspects the previously generated data base. The paper also presented examples of the use of the language and describes the implementation of its processor.

[Cohe82] Abstract Prototypes are built for a variety of reasons. This paper offers an alternative to the use of a prototype as a means of testing a specification (i.e. someone who "knows" what he wants compares his intuitive understanding with the behavior of the prototype on particular test cases). The alternative is symbolic execution of a formal specification, i.e. the specification is the prototype and its behavior determined by symbolic execution rather than the traditional "concrete" execution. This is an extension of the approach to rapid prototyping based on operational specification and an alternative to testing prototypes whether manually constructed or developed mechanically from such an operational specification. One advantage of this approach is that the prototype need not be built at all. Of course, the formal specification must be written, but this is often necessary anyway, especially if the specifier and implementor are different people. A more important advantage arising from symbolic execution is that a large subset of the possible behaviors can be examined at once.

[Come79] Abstract: In this paper we:
1. discuss the need for quantitatively reproducible experiments in the study of top-down design;
2. propose the design and writing of tutorial papers as a suitably general and inexpensive vehicle;
3. suggest the software science parameters as appropriate metrics;
4. report two experiments validating the use of these metrics on outlines and prose; and
5. demonstrate that the experiments tended toward the same optimal modularity.

The last point appears to offer a quantitative approach to the estimation of the total length or volume (and the mental effort required to produce it) from an early stage of the top-down design process. If results of these experiments are validated elsewhere, then they will provide basic guidelines for the design process.

[Conn87] Abstract: The Ada Software Repository (ASR) is a collection of Ada programs, software components, information files, and educational material that resides on the computer known as SIMTEL20 on the

Defense Data Network (DDN), a world-wide network of computer networks supported by the US Department of Defense. This repository has been accessible to any host computer on the DDN since November 26, 1984, and is available to any member of the Ada community in the United States and its allies.

The Ada Software Repository (ASR) is a free source of Ada programs and information. It serves two roles: to promote the exchange and use of Ada programs (including reusable software components) and to promote Ada education (by providing information on items of interest to the Ada community and by providing examples of working, useful Ada programs). There is over 40M bytes of source code, documentation, and information files in the ASR.

[Cont86] Abbreviated Preface: This book is intended to be used by both practitioners and students who are or who expect to be involved in managing or producing software. It is our firm belief that software engineering in general and software metrics in particular should be a part of the curriculum of all computer science programs.

Our goal is that, through this book, many readers will be introduced to the world of software metrics and models. We also believe that this material will serve as an impetus to researchers in software engineering and software developers to derive new metrics and models, and to gather data to help confirm the utility of new and existing metrics and models.

[Cook82] Abstract: Software metrics, an area of software engineering, is concerned with various measurements of computer software and its development. Software metrics, its importance, some current areas of investigation, and problems are described. An annotated bibliography of work in software metrics is included.

[Coul83] Abstract: Halstead proposed a methodology for studying the process of programming known as software science. This methodology merges theories from cognitive psychology with theories from computer science. There is evidence that some of the assumptions of software science incorrectly apply the results of cognitive psychology studies. Halstead proposed theories relative to human memory models that appear to be without support from psychologists. Other software scientists, however, report empirical evidence that may support some of those theories. This anomaly places aspects of software science in a precarious position. The three conflicting issues discussed in this paper are 1) limitations of short-term memory and number of subroutine parameters, 2) searches in human memory and programming effort, and 3) psychological time and programming time.

[Cox81] Abstract: In this paper [the author] describe the practical problems of designing a regression test set for an existing mini-computer operating system. The ideal regression test would test each function with all possible combinations of the options for each variation of the operating system. This is impractical if not impossible so the alternative is to choose the individual cases for maximum coverage. To do that the system is viewed both functionally and structurally and cases are selected for inclusion in the test set. The method of selecting the tests is described along with the tools that will be needed to measure the coverage and to maintain the test set.

[Crai88a] Abstract: The Trusted Systems Group of I.P. Sharp Associates Limited has recently released a prototype formal verification system, called m-EVES. m-EVES consists of a new language, called m-Verdi, for implementing and specifying software; a new logic (which has been proven sound); and a new theorem prover, called m-NEVER, which integrates many state-of-the-art techniques drawn from the theorem proving literature.

In this paper, after a brief overview of the m-EVES system, an application of m-EVES to a proof of a non-trivial security property (non-interference) for a pedagogical computer system (the Low Water Mark system) is discussed. An example demonstrates some of the power and novel features of m-EVES. The paper concludes with a comparison of the m-EVES solution with similar efforts using the Gypsy Verification Environment and the Boyer-Moore theorem prover.

[Crai88b] Abstract: This paper describes the development of a new tool for formally verifying software. The tool is called m-EVES and consists of a new language, called m-Verdi, for implementing and specifying software; a new logic, which has been proven sound; and a new theorem prover, called m-NEVER, which integrates many state-of-the-art techniques drawn from the theorem proving literature. Two simple examples are used to present

the fundamental ideas embodied within the system.

[Curr86] Abstract: The accepted approach to software development is to specify and design a product in response to a requirements analysis and then to test the software selectively with cases perceived to be typical to those requirements. In contrast it is possible to embed the software development and testing process within a formal statistical design. In such a design, software testing can be used to make statistical inferences about the reliability of the future operation of the software. This paper describes a procedure for certifying the reliability of software before its release to users. The ingredients of this procedure are a life cycle of executable product increments, representative statistical testing, and a standard estimate of the MTTF (mean time to failure) of the product at the time of its release.

[Curt79a] Abstract: This experiment is the third in a series investigating characteristics of software which are related to its psychological complexity. A major focus of this research has been to validate the use of software complexity metrics for predicting programmer performance. In this experiment we improved experimental procedures which produced only modest results in the previous two studies. The experimental task required 54 experienced Fortran programmers to locate a single bug in each of three programs. Performance was measured by the time to locate and successfully correct the bug. Much stronger results were obtained than in earlier studies. Halstead's E proved to be the best predictor of performance, followed by McCabe's V(G) and the number of lines of code.

[Curt81] Abstract: Three software complexity measures (Halstead's E, McCabe's V(G), and the length as measured by number of statements) were compared to programmer performance on two software maintenance tasks. In an experiment on understanding, length and V(G) correlated with the percent of statements correctly recalled. In an experiment on modification, most significant correlations were obtained with metrics computed on modified rather than unmodified code. All three metrics correlated with both the accuracy of the modification and the time to completion. Relationships in both experiments occurred primarily in unstructured rather than structured code, and in code with no comments. The metrics were also most predictive of performance for less experienced programmers. Thus, these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance in understanding the code.

[DACS79b] Preface: The purpose of this document is to record, as accurately as is possible in a still-evolving discipline, the terminology currently being used in the field of software engineering. We hope that the DACS GLOSSARY will help to improve communication within the software engineering community and will also provide an impetus toward the sorely needed standardization of terminology.

This software engineering glossary is one of the products of the Data and Analysis Center for Software (DACS). The DACS will continue to update this glossary to reflect current term usage. Suggestions, comments, and critiques are welcome.

[DOD88a] Forward:

1. This standard establishes uniform requirements for software development that are applicable throughout the system life cycle. The requirements of this standard provide the basis for Government insight into a contractor's software development, testing, and evaluation efforts.
2. This standard is not intended to specify or discourage the use of any particular software development method. The contractor is responsible for selecting software development methods (for example, rapid prototyping) that best support the achievement of contract requirements.
3. This standard, together with the other DOD and military documents referenced in Section 2, provides the means for establishing, evaluating, and maintaining quality in software and associated documentation.
4. Data Item Descriptions (DIDs) applicable to this standard are listed in Section 6. These DIDs describe a set of documents for recording the information required by this standard. Production of deliverable data using automated techniques is encouraged.

5. Per DODD 5000.43, Acquisition Streamlining, this standard must be appropriately tailored by the program manager to ensure that only cost-effective requirements are cited in defense solicitations and contracts. Tailoring guidance can be found in DOD-HDBK-248, Guide for Application and Tailoring of Requirements for Defense Material Acquisitions.

[DODD86a] Abstract: The purpose of this Manual is to describe the format, content, and submission procedures for Test and Evaluation Master Plans (TEMPS) of major defense acquisition programs. The TEMP is the basic planning document for all test and evaluation (T&E) related to a particular system acquisition and is used by OSD and all DoD Components in planning, reviewing, and approving T&E. The TEMP provides the basis for all other detailed T&E planning documents and serves as an essential element of the Joint Resources Management Board (JRMB) decision-making process outlined in DoD Instruction 5000.2, "Major System Acquisition Procedures."

[DODD86b] Purpose: This Manual describes the procedures to be followed in preparing Test and Evaluation Master Plans (TEMPs) for major defense acquisition programs, including those major (to include OSD designated) programs that are exempted from the Joint Resources Management Board (JRMB) process or when JRMB authority has been delegated to the DoD Components.

[DODD87] Overview: This Manual describes procedures for preparing the software portion of the Test and Evaluation Master Plan (TEMP). The Manual's objective is to establish a disciplined framework that will result in software testing that is methodically planned, results oriented, and designed to produce meaningful evaluations.

[DODS86] Abbreviated Forward: This standard contains requirements for the development, documentation, and implementation of a software quality program. This program includes planning for and conducting evaluations of the quality of software, associated documentation, and related activities, and planning for and conducting the follow-up activities necessary to assure timely and effective resolution of problems.

This standard, together with other DOD and military specifications and standards governing software development, configuration management, specification practices, project reviews and audits, and subcontractor management, provide a means for achieving, determining, and maintaining quality in software and associated documentation. This standard incorporates the applicable requirements of MIL-STD-1520 and MIL-STD-1535.

This standard implements the policies of DODD 4155.1, Quality Program, and provides all of the necessary elements of a comprehensive quality program applicable to software development and support. This standard interprets the requirements of MIL-Q-9858, Quality Program Requirements, for software and is to be used in conjunction with MIL-Q-9858 for system development and support projects.

[Dahl72] Table of Contents: Notes on structured programming, correctness of proofs, validity of proofs. Notes on data structuring, the concept of type, unstructured data types, recursive data structures, axiomatisation, references. Hierarchical program structures, object classes, coroutines, list structures, program concatenation, concept hierarchies, references.

[Daly77] Abstract: This paper describes four major aspects of software management: development statistics, development process, development objectives, and software maintenance. The control of both large and small software projects is included in the analysis.

[Darr78] Abstract: Symbolic execution provides a basis for a program analysis tool that allows one to choose intermediate points in a spectrum ranging between individual test runs and general correctness proofs. One can perform a single "symbolic execution" of a program that is equivalent to a large (possibly unbounded) number of normal test runs. Not only can test results be checked by careful manual inspection, but if a machine interpretable specification is supplied, the results can be checked automatically. Furthermore, by varying the amount of symbolic data and program specification introduced, one can move from a normal execution (no symbolic data)

to a symbolic execution that provides a proof of correctness.

[Davi77] Abstract: The paper considers informally the relationship between computer aided mathematical proof, formal algebraic languages, computation with transcendental numbers, and proof by sampling.

[Davi82b] Abbreviated Introduction: There are at least four phases in the development of "correct" software:
— Understanding the problem. The program designer may work with intended users of the system to develop an intuitive understanding of the problem and possible approaches to its solution.
— Formal specification. Once the designer knows intuitively how to solve the problem, the solution must be specified unambiguously.
— Programming. An implementation of the specification is programmed.
— Verification. The implementation developed in step three is shown to satisfy the specification of step two.
There is a certain amount of testing and debugging that goes on at each of these stages until one is satisfied with the current step and moves on to the next. Several verification techniques have been developed to assist in accomplishing step four. However, even after a proof is completed we cannot claim to have a "correct" program, only one that satisfies the given specification.

How does one "debug" a specification? We cannot hope to formally prove that a specification is "correct" with respect to our intuition, but we can at least test it to see that it conforms to our intuition in specific cases.

[Davi83a] Preface: Theoretical computer science is the mathematical study of models of computation. As such, it originated in the 1930s, well before the existence of modern computers, in the work of the logicians Church, Gödel, Kleene, Post, and Turing. This early work has had a profound influence on the practical and theoretical development of computer science. Not only has the Turing-machine model proved basic for theory, but the work of these pioneers presaged many aspects of computational practice that are now commonplace and whose intellectual antecedents are typically unknown to users. Included among these are the existence in principle of all-purpose (or universal) digital computers, the concept of a program as a list of instructions in a formal language, the possibility of interpretive programs, the duality between software and hardware, and the representation of languages by formal structures based on productions. While the spotlight in computer science has tended to fall on the truly breathtaking technological advances that have been taking place, important work in the foundations of the subject has continued as well. It is our purpose in writing this book to provide an introduction to the various aspects of theoretical computer science for undergraduate and graduate students that is sufficiently comprehensive that the professional literature of treatises and research papers will become accessible to our readers.

We are dealing with a very young field that is still finding itself. Computer scientists have by no means been unanimous in judging which parts of the subject will turn out to have enduring significance. In this situation, fraught with peril for authors, we have attempted to select topics that have already achieved a polished classic form, and that we believe will play an important role in future research.

[Davi83b] Introduction: We propose a definition of the notion of adequacy of software test data and discuss justification, difficulties, and properties of the notion. It is not the purpose of this paper to suggest a definite practically applicable criterion of test data adequacy. Rather we present a theoretical analysis which, it is believed, gives insight into such questions as:
1. For a given program, what points must belong to a test set in order that it may be deemed adequate?
2. For a given program how many points must belong to an adequate test set?
3. What kind of approximation to "correctness" can be provided by the knowledge that a program has been "adequately" tested?
We believe, in general, that an adequacy criterion should be invoked only after the test data fails to expose errors. Clearly, as long as there is an element of the test set on which the program does not agree with the specification, we know that the test data is still doing its job and that testing (and subsequent debugging) must continue. Once the program does agree with the specification on all elements of a set of test data, we must decide whether the testing phase can end, and hence we will need to invoke some kind of adequacy criterion.

[Davi88a] Abstract: A study of the predictive value of a variety of syntax-based program complexity measures is described. Experimentation with variants of new chunk-oriented measures showed that one should judiciously select measurable software attributes as proper indicators of what one wishes to predict, rather than hoping for a single, all purpose complexity measure. This study has shown that it is possible for particular complexity measures or other factors to serve as good predictors of some properties of program but not for others. For example, a good predictor of construction time will not necessarily correlate well with the number of error occurrences.

Halstead's effort measure (E) was found to be a better predictor than the other two nonchunk measures we evaluated: McCabe's V(G) and lines of code, but at least one chunk measure predicted better than E in every case.

[DeFr85] Abstract: This work deals with issues of interactive debugging for the concurrent language ECSP. The debugger matches a formal specification of the expected behavior. This specification can be given at different levels of abstraction. Control is returned to the user when an error is detected. The user can then modify the flow of the computation and/or dynamically change the specification of the expected behavior. The debugger implementation is based on program transformation techniques.

[DeMi77] Abbreviated Introduction: Until very recently, research in software reliability was divided quite neatly into two – usually warring – camps: methodologies with a mathematical basis and methodologies without such a basis. In the former view, "reliability" is identified with "correctness" and the principle tool has been formal and informal verification. In the latter view, "reliability" is taken to mean the ability to meet overall functional goals to within some predefined limits. We have argued that the latter view holds a great deal of promise for further development at both the practical and analytical levels. Howden proposes a first step in this direction by describing a method for "testing" a certain restricted class of programs whose behavior can – in a sense Howden makes precise – be *algebraicized*. In this way, "testing" a program is reduced to an equivalence test, the major components of which become

1. a combinatorial identification of "equivalent" structures;
2. an algebraic test

$$f_1 \equiv f_2,$$

where f_i, i = 1, 2 is a multivariable polynomial (multinomial) of degree specified by the program being considered.

We are inspired by Rabin and, less directly, by the many successes of *Erdös* and Spencer to attempt a *probabilistic* solution to (ii). Using these methods, we show that (ii) can be tested with probability of error ϵ with only $o((g\epsilon))$ evaluations of multinomials, where g is a slowly growing function of only ϵ. In particular, 30 or so evaluations should give sufficiently small probability of error for most practical situations.

[DeMi78] Abstract: In many cases tests of a program that uncover simple errors are also effective for uncovering much more complex errors. This so called coupling effect can be used to save work during the testing process.

[DeMi79a] Abstract: It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage. It is felt that ease of formal verification should not dominate program language design.

[DeMi87a] Abbreviated Preface: This book is an updated and edited version of the report of the Software Test and Evaluation Project of the Secretary of Defense (Research and Engineering). The primary objective of STEP was (and remains) the development of improved policy and guidance for the use by the U.S. Department of Defense for the test and evaluation of computer software for so-called "mission-critical" applications.

[The book provides] state-of-the-art and state-of-the-practice overviews. These overviews contain brief

descriptions of major test methodologies, catalogs of automated tools to support them, essentially exhaustive bibliographies, case studies of good and bad examples of software testing and exegeses of major standards.

[DeMi87b] Abbreviated Abstract: The Mothra environment is an integrated set of tools and interfaces that support the planning, definition, preparation, execution, analysis and evaluation of tests of software systems. The support provided by Mothra is applicable from the earliest stages of software design and development through the progressively later stages of system integration, acceptance testing, operation and maintenance. Mothra has been designed to address [various] cost concerns. Two primary design criteria, in particular, are significant in this regard. First, the Mothra interfaces-particularly user interfaces-are high-bandwidth. This allows us to present more information during testing and retesting. Coupled with proper design and integration with familiar displays, it should obviate the need for extensive training to use Mothra.

Secondly, the overall Mothra architecture imposes no a priori constraints on the size of the software systems that can be tested in the environment. The practical meaning of this criterion is that the same architecture is able to service programs varying in size from individual modules of less than 10^2 source lines to fully integrated systems of more than 10^8 lines. The human user-the tester-is able to apply comparable functions across a familiar interface as the software being tested evolves in size and complexity by several orders of magnitude. In fact, the only indicators of size or complexity that have ties to the Mothra architecture are the operating system cost penalties and performance delays inherent in manipulating massive objects. All other costs and resource demands are under the direct control of the tester.

An important mechanism for meeting these criteria is that Mothra is reconfigurable, allowing the integration of user and system tools with which the tester may already be familiar, and allowing the system to make use of different underlying hardware architectures or different capabilities. We address this in Mothra by the use of thematic tools for software testing. For example, programmers in modern development environments interact increasing with an array of very powerful source language debuggers. Even though formal testing methodologies and debugging are very different activities, the debugging theme can be used as a metaphor to carry the tester from tool to tool as the software being tested evolves.

One Mothra system has been constructed using the AT&T Bell Labs built interactive bitmap display terminal running under the control of a UNIX[3] window manager called Layers. The host environment is a modestly configured VAX 11/780 running UNIX 4.3 BSD. Another version has been implemented on VAX stations running Ultrix[4] 1.2 and the X Window System. However, the architecture of Mothra encourages rehosting. Furthermore, explicit operations allow Mothra processes to spawn parallel and vectorized processes for execution by a Cyber 205 (or any other powerful parallel machine.

[DeMi87c] Abstract: This paper presents a new technique for automatically generating test data. The method is based on mutation analysis and uses constraints to specify test cases designed to find particular types of errors. A prototype implementation has been used to effectively kill mutants in a mutation system. The technique also combines the capabilities of previous test data generation methods. The paper includes an initial set of constraints and discusses some of the problems that must be solved in order to develop a complete implementation of the technique.

[DeMi87d] Abstract: Mothra is a software testing environment that supports mutation-based testing of software systems. Mothra is interactive; it provides a high-bandwidth user interface to make software testing faster and less painful. Mothra currently runs on a variety of systems under 4.3 BSD UNIX, UNIX System V, and ULTRIX-32 1.2. This paper begins with a brief introduction to mutation analysis. We then take the reader on a guided tour of Mothra, emphasizing how it interacts with the tester. We conclude with a short discussion of

3. UNIX is a registered trademark of AT&T

4. Ultrix is a registered trademark of Digital Equipment Corporation

Mothra's internal design.

[DeMi88a] Abstract: Mothra is a software testing environment that supports mutation-based testing of software systems. Mothra is interactive; it provides a high-bandwidth user interface to make software testing faster and less painful. Mothra currently runs on a variety of systems under 4.3 BSD UNIX, UNIX System V, and ULTRIX-32 1.2. This paper begins with a brief introduction to mutation analysis. We then take the reader on a guided tour of Mothra, emphasizing how it interacts with the tester. Then we present with a short discussion of Mothra's internal design. Next, we discuss some major problems with using mutation analysis and discuss possible solutions. We conclude by presenting a solution to one of these problem-a new method of automatically generating mutation-adequate test data.

[DeMi88b] Abstract: The purpose of this IDA Paper is to document the results of an analysis of software testing and verification technology conducted for the Ada Joint Program Office (AJPO) and the Rome Air Development Center (RADC) by the Institute for Defense Analyses (IDA). The Paper presents a coordinated strategy for meeting a critical technology goal of the U.S. Department of Defense – the development of computer software for these systems upon which the Armed Forces can rely for the success of missions with extreme and often life critical requirements.

[DeRe76] Abstract: We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), usually written by different people.

We need languages for programming-in-the-small, i.e., languages not unlike the common programming languages of today, for writing modules. We also need a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler.

[Dela88] Abstract: Metrics are the quantification of environmental and performance factors to measure the effectiveness of activities in the areas of resources, schedule, quality, and risk. Metrics provide both a prospective and retrospective measure of accomplishment. Retrospective data provides a baseline for the next project. Prospective data support forecasting, planning, and control of on-going activities. The latter is obviously preferable.

This paper summarizes the types of metrics developed during the foundation phase of the Army WWMCCS Information System (AWIS), and the methodology applied to achieve a selected subset of these metrics during full scale development, which starts early in 1988 and is expected to last for five years.

[Denn78] Abstract: Queueing network models have proved to be cost effective tools for analyzing modern computer systems. This tutorial paper presents the basic results using the operational approach, a framework which allows the analyst to test whether each assumption is met in a given system. The early sections describe the nature of queueing network models and their applications for calculating and predicting performance quantities. The basic performance quantities - such as utilizations, mean queue lengths, and mean response times - are defined, and operational relationships among them are derived. Following this, the concept of job flow balance is introduced and use to study asymptotic throughputs and response times. The concepts of state transition balance, one-step behavior, and homogeneity are then used to relate the proportions of time that each system state is occupied to the parameters of job demand and to device characteristics. Efficient methods for computing basic performance quantities are also described. Finally the concept of decomposition is used to simplify analyses by replacing subsystems with equivalent devices. All concepts are illustrated liberally with examples.

[DiMa85] Abstract: This symbolic run-time debugger for Ada provides facilities for observing and manipulating the execution of a monitored program, also for concurrent aspects. The debugger can be used interactively, and also as a monitoring program to control the application. A feature of this project is the use of relational algebra for defining compiler and kernel interfaces and for handling debugger information. The implementation is based

on an Ada task to interface with the debugging operator and a set of user-defined Ada monitoring tasks. A prototype of the debugger was completed as a part of ART, a relational translator and interpreter for Ada.

[Dijk76a] Table of Contents: Executional abstractions. The role of programming languages. States and their characterization. The characterization of semantics. The semantic characterization of a programming language. Two theorems. On the design of properly terminating constructs. Euclid's algorithm revised. The formal treatment of some small examples. On nondeterminancy being bounded. An essay on the notion: "the scope of variables." Array variables. The linear search theorem. The problem of the next permutation. The problem of the Dutch national flag. Updating a sequential file. Merging problems revisited. An exercise attributed to R.W. Hamming. The pattern matching problem. Writing a number as the sum of two squares. The problem of the smallest prime factor of a large number. The problem of the most isolated villages. The problem of the shortest subspanning tree. Rem's algorithm for recording of equivalence classes. The problem of convex Hull in three dimensions. Finding the maximal strong components in a directed graph. On manuals and implementations. In retrospect.

[Dijk76b] Abbreviated Introduction: Reviewing recent experiences gained during the design and construction of a multiprogramming system [the author] finds [himself] torn between two apparently conflicting conclusions. Confining [himself] to the difficulties more or less mastered [the author] feels that such a job is (or at least should be) rather easy; turning [the authors] attention to the remaining problems such a job strikes [the author] as cruelly difficult. The difficulties that have been overcome reasonably well are related to the reliability and the producibility of the system, the unsolved problems are related to the sequencing of the decisions in the design process itself.

[The author] shall mainly describe where we feel that we have been successful. This choice has not been motivated by reasons of advertisement for one's own achievements; it is more that a good knowledge of what-and what little!-we can do successfully, seems a safe starting point for further efforts, safer at least than starting with a long list of requirements without a careful analysis whether these requirements are compatible with each other.

[Dill88a] Abstract: There have been several efforts to use symbolic execution to test and analyze concurrent programs. Recently proof systems have also emerged for concurrent programs and for the Ada language in particular. This paper reports on an experience with developing two different approaches, which use symbolic execution, to prove partial correctness and general safety properties of Ada programs. One approach is based upon interleaving the task components while the other is based upon verifying the tasks in isolation and then performing cooperation proofs. Both approaches extend past efforts by incorporating tasking proof rules into the symbolic executor allowing Ada programs with tasking to be formally verified.

The limitations of each approach are presented, along with each approach's advantages and disadvantages. In particular, the difficulty of dealing with communication statements in a loop structure are addressed in detail.

[Dill88b] Abstract: Symbolic execution has been used successfully with sequential programs for generating the *verification conditions* required for correctness proofs. This paper shows how the symbolic execution model for sequential programs can be extended to a tasking subset of Ada. The criteria for correct operation of a concurrent program include *safety properties*, such as mutual exclusion and freedom from deadlock. The extended model, therefore, provides a basis for the automatic generation of verification conditions for proving general safety properties of Ada tasking programs.

[Dill88c] Abstract: An approach to the design of concurrent software systems based on the constrained expression formalism is described. This formalism provides a rigorous conceptual model for the semantics of concurrent computations, thereby supporting analysis of important system properties as part of the design process. This approach allows designers to use standard specification and design languages, rather than forcing them to deal with the benefits of formal rigor without the associated pain of unnatural concepts or notations for its users. The conceptual model of concurrency underlying the constrained expression formalism treats the collection of

possible behaviors of a concurrent system as a set of sequences of events. The constrained expression formalism provides a useful closed-form description of these sequences. Algorithms were developed for translating designs expressed in a wide variety of notations into these constrained expression descriptions. A number of powerful analysis techniques that can be applied to these descriptions have also been developed.

[Doer85] Abstract: This paper describes research conducted by the Software Engineering Laboratory (SEL) on the use of dynamic variables as a tool to monitor software development. The intent of the project is to identify project independent measures which may be used in a management tool for monitoring software development. This study examines several Fortran projects with similar profiles. The staff was experienced in developing these types of projects. The projects developed serve similar functions. Because these projects are similar we believe some underlying relationships exist that are invariant between the projects. These relationships, once well defined, may be used to compare the development of different projects to determine whether they are evolving the same way previous projects in this environment evolved.

[Down85a] Abstract: In this paper, an approach to the modeling of software testing is described. A major aim of this approach is to allow the assessment of the effects of different testing (and debugging) strategies in different situations. It is shown how the techniques developed can be used to estimate, prior to the commencement of testing, the optimum allocation of test effort for software which is to be nonuniformly executed in its operational phase. In addition, the question of application of statistical models in cases where the data environment undergoes changes is discussed. Finally, two models are presented for the assessment of the effects of imperfections in the debugging process.

[Down86] Abstract: This paper shows how a major (and questionable) assumption underlying a previously reported approach to the modeling of software testing can be relaxed in order to provide a more realistic model. Under the assumption of uniform execution the new model is found to perform only marginally better than the previous model, indicating that the uniform execution assumption is a poor one. A nonuniform execution model, also developed in the paper, is then shown to give very good performance on application to three sets of software reliability data. The results obtained point the way to further developments which are likely to lead to models whose performance is superior to that of the nonuniform execution model presented here. The paper also devotes some attention to the problem of comparison of performance of different models and points out some difficulties in this area.

[Drap66] Abbreviated Preface: We have tried to bring together in this book a number of procedures developed for regression problems in current use. Since our emphasis is on *practical* application, we have stated theoretical results without proofs in many cases. While the text can be used without any computing equipment at all (or perhaps with only a desk calculator), we have made use of computer printouts in some parts of the book. We have also provided various exercises, some of which can be solved easily "by hand," and other more extensive ones for which use of an electronic computer would be helpful, though not absolutely essential.

This book provides a standard, basic course in multiple linear regression, but it also includes material that either has not previously appeared in a textbook or, if it has appeared, is not generally available. For example, Chapter 3 discusses the examination of residuals; Chapter 6 examines the methods employed as selection procedures in various types of regression programs; Chapter 8 discusses the planning of large regression studies; and Chapter 10 provides a basic introduction to the theory of nonlinear estimation.

[Duke89] Abbreviated Introduction: Verifying and validating flight and mission-critical systems is a major activity at the Dryden Flight Research Facility of the National Aeronautics and Space Administration's Ames Research Center. The Ames-Dryden staff is responsible for flight safety for all vehicles flown at the Dryden facility, which is located in the desert north of Los Angeles. Because these systems are used in research aircraft, the V&V experience at Ames-Dryden is primarily with one-of-a-kind research systems on experimental vehicles.

The Ames-Dryden V&V methodology relies on testing, peer review, abstract models, simulations, and validation by actual flight. This methodology also relies, in a large part, on engineering judgement and a tradition

that has evolved from experience with flight-critical systems that include the digital flight-control system on the F-8, the three-eights-scale remotely piloted F-15, the highly maneuverable Himat, the Advanced-Fighter Technology Integration F-16, and the X-29 forward-swept-wing aircraft.

[Dunc81] Abstract: We present a method for generating test cases that can be used throughout the entire life cycle of a program. This method uses attributed translation grammars to generate both inputs and outputs, which can then be used either as is, in order to test the specifications, or in conjunction with automatic test drivers to test an implementation against the specifications.

The grammar can generate test cases either randomly or systematically. The attributes are used to guide the generation process, thereby avoiding the generation of many superfluous test cases. The grammar itself not only drives the generation of test cases but also serves as a concise documentation of the test plan.

In this paper, we describe the test case generator, show how it works in typical examples, compare it with related techniques, and discuss how it can be used in conjunction with various testing heuristics.

[Dunh83] Abbreviated Introduction: Software engineering can attain the status of scientific discipline only if it is built upon a solid foundation of objective measurement. In fact, its maturity as a discipline will be reflected in the degree to which measurement becomes a normal part of the software development and maintenance process.

Measurement is a difficult area to discuss as an isolated topic because it is fundamental to virtually all aspects of software engineering and management. But this is precisely what makes it so important. In one article, we cannot hope to cover all possible uses for measurement or all possible types of measures and models. We can, however, provide a framework for discussion.

Measurement from the perspective of those involved in software development and maintenance has practical benefits as a management, development, and contractual tool. From the scientist's perspective, it is useful in the development of quantitative models. This article reviews measurement and modeling activities: resource expenditures, software and system *reliability*, system performance, and user performance. It then describes the measurement activities in the STARS program, which are designed to further advance the technology of measurement and to facilitate its widespread use.

[Dunh86] Digital computers are being used more frequently for process control applications in which the cost of system failure is high. Consideration of the potentially life-threatening risk, resulting from the high degree of functionality being ascribed to the software components of these systems, has stimulated the recommendation of various designs for tolerating software faults. Such designs are not panaceas, for they still entail-as did the fault intolerant designs they are superceding-an unknown probability of failure. The paper discusses four reliability data gathering experiments which were conducted using a small sample of programs for two problems having ultrareliability requirements, n-version programming for fault detection, and repetitive run modeling for failure and fault rate estimation. The experimental results agree with those of Nagel and Skrivan in that the program error rates suggest an approximate log-linear pattern and the individual faults occurred with significantly different error rates. Additional analysis of the experimental data raises new questions concerning the phenomenon of interacting faults. This phenomenon may provide one explanation for software reliability decay. The fourth experiment underscored the difficulty in distinguishing between observations of deficiencies in the design of the algorithm and observations of software faults for real-time process control software. These experiments are a part of a program of serial experiments being pursued by the System Validation Methods of NASA-Langley Research Center to find a means of credibly performing reliability evaluations of flight control software.

[Dunn74] Abbreviated Preface: This book has been based on notes originally developed for a one-semester course in analysis of variance, regression, and covariance. We had two general objectives in [it's development]. [To prepare] a self-contained textbook, [and one] that might be useful as a reference.

Chapters 1 through 4 provide the introduction necessary for studying analysis of variance and regression. Chapters 5,6, and 7 deal with the fixed effects model analysis of variance Model I. Chapter 8 gives a brief introduction to confounding, still with Model I. Chapter 9 presents variable effects models (Models II and III). Chapters 10, 11, and 12 introduce linear, multiple, and polynomial regression; Chapter 13 is concerned with

covariance analysis. In Chapter 14 we discuss various techniques for screening data before analysis. We consider this material so important that it must be gathered together into a single chapter and placed as conspicuously as possible; clearly it cannot be the first chapter, and so it must be the last.

[Dunn82] Abbreviated Introduction: As we see it – and it is heartening to note that this is becoming the prevailing view – software quality assurance is the mapping of the managerial precepts and design disciplines of quality assurance onto the applicable management and technological space of software engineering. In the transfer, familiar quality assurance approaches to improving control and performance metamorphose into techniques and tools different from those to which the quality community is accustomed. For its part, software approaches to the production and maintenance of computer software are given new form as well as procedural efficiency. Yet both communities, to their mutual advantage, can easily relate to this concept of software quality assurance.

Software quality assurance can be constructive, can avoid being a bureaucratic impediment, only by drawing upon fundamental concepts of both software engineering and quality assurance. It is the resulting amalgam which we set out to describe and how many ingredients can be seen in the road map to the balance of the book, which is provided toward the end of Chapter 1.

[Dunn84] Table of Contents: Introduction. To err is human; to find the bug, divine, an overview ot development methodologies. Static methods. Requirements and design reviews, code reviews, static analysis, proof of correctness. Dynamic testing. Matters of strategy, glass box testing, black box testing, analysis of defect and failure data. Operational phase. Configuration control, maintenance and modification.

[Duns77] Abstract: One measure of programming complexity is the number of "program changes" that must be made from the initial version of a program until it is in final form. A count of errors occurring in the debugging process is an accepted measure of difficulty in programming. Using source modules from an experiment involving thirty-three subjects developing a moderately difficult program, it has been demonstrated that "program changes" correlates well with the count of errors. In addition, subjects whose initial version of a program had either a moderate average nesting depth and/or a moderate usage of global variables made fewer program changes during development.

[Duns78b] Abstract: Programming complexity (the amount of difficulty in constructing a program) may depend upon certain programming factors (choices of programming language features). Using program changes as a programming complexity measure, previous research has identified five potential programming factors. This paper suggests that subjects tend to use the same levels of these factors in two different programming languages supporting the conjecture that these factors are elements of individual programming style. It also describes five potential programming factors, and although each has intuitive appeal, only average procedure length was related to programming complexity.

[Dura78] Abstract: Program testing remains the major way in which program designers convince themselves of the validity of their programs. Software reliability measures based on hardware reliability concepts have been proposed, but adequate models of software reliability have not yet been developed. Investigators have recently studied formal program testing concepts, with promising results, but have not seriously considered quantitative measures of the "degree of correctness" of a program. We present models for determining, via testing, such probabilistic measures of program correctness as the probability that a program will run correctly on randomly chosen input data, confidence intervals on the number of errors remaining in a program, and the probability that the program has been completely tested. We also introduce a procedure for enhancing correctness estimates by quantifying the error reducing performance of the methods used to develop and debug a program.

[Dura80] Abstract: The point of all validation techniques is to raise assurance about the program under study, but no current methods can be realistically thought to give 100% assurance that a validated program will perform correctly. There are currently no useful ways for quantifying how "well-validated" a program is. One measure of program correctness is the proportion of elements in the program's input domain for which it fails to execute

correctly, since the proportion is zero i.f.f. the program is correct. This proportion can be estimated statistically from the results of program tests and from prior subjective assessments of the program's correctness. Three examples are presented of methods for determining *s*-confidence bounds on the failure proportion. It is shown that there are reasonable conditions (for programs with a finite number of paths) for which ensuring the testing of all paths does *not* give better assurance of program correctness.

[Dura81a] Abstract: Random testing of programs is usually (but not always) viewed as a worst case of program testing. Test case generation that takes into account the program structure is usually preferred. Path testing is an often proposed ideal for structural testing. Path testing is treated here as an instance of partition testing, where by partition testing is meant any testing scheme which forces execution of at least one test case from each subset of a partition of the input domain. Simulation results are presented which suggest that random testing may often be more cost effective than partition testing schemes. Also, results of actual random testing experiments are presented which confirm the viability of random testing as a useful validation tool.

[Dura81b] Abstract: Mill's capture-recapture sampling method allows the estimation of the number of errors in a program by randomly inserting known errors and then testing the program for both inserted and indigenous errors. This correspondence shows how correct confidence limits and maximum likelihood estimates can be obtained from the test results. Both fixed sample size testing and sequential testing are considered.

[Duva80] Abstract: This paper summarizes the results of a study to determine the data requirements for software reliability modeling. The major assumptions of the models are presented along with a brief description of their uses and the data needed to exercise the models. Methodologies for evaluating failure databases are presented including a sample evaluation to determine the adequacy of the data to do comparisons across a wide variety of projects and to determine if the database contains data elements as required by the various models.

[Dyer80] Abbreviated Introduction: In this paper on software development, the focus is on the blend of modern software methods with established development practices. Reducing diversity, increasing visibility, and improving productivity in the development process are the principal means of intellectual control of development. Improved product quality, product transportability, and product adaptability are longer-range goals.

The development methodology is defined in terms of practices that recognize the increased precision introduced by modern design methods and that attempt to introduce the rigor of modern design into the methods of software product development. Code management practices deal with the implementation of software and the control of its release as a product. Integration engineering practices address plans for building software products.

[Dyer85a] Introduction: Testing to confirm that the implemented software (and its design) satisfies its intended requirements is performed by someone other than the software developer. In this case, black box or function testing is performed, not to verify that the code executes, but, more importantly, that it performs its intended job. Independent testers are disassociated from the product design and are more objective in verifying that a product operates as expected. This independent testing is commonly defined as the software verification step in the software life cycle.

This chapter discusses an approach to software verification which downplays the current error detection focus and promotes an operational testing focus. Functional testing from an operational use perspective demonstrates not only that the software performs its job, but also that it does it in the planned user environments. The emphasis on user perspective should help ensure the development of executing products which are also usable in the field and whose field reliability (MTTF) can be estimated during development.

To implement this approach, a statistical testing procedure is defined for function verification. The procedure uses the probability distributions of the product inputs and randomized sampling techniques to organize test material. The randomization supports statistical inferences about the product's operational characteristics and an estimation of its expected reliability (MTTF).

[East72] Abstract: A method is presented for obtaining system confidence limits based on component test results. The techniques consists of estimating the asymptotic variance of the maximum likelihood estimate of system reliability, equating this to the estimate of the variance in binomial sampling, and solving for n and x, the pseudo-numbers of system tests and successes. These are then substituted into the incomplete beta function and confidence limits obtained in the usual way for binomial sampling.

[Eckh85] Summary: Fundamental to the development of redundant software techniques (known as fault-tolerant software) is an understanding of the impact of multiple joint occurrences of errors, referred to here as coincident errors. A theoretical basis for the study of redundant software is developed which (1) provides a probabilistic framework for empirically evaluating the effectiveness of the general (N-version) strategy when component versions are subject to coincident errors, and (2) permits an analytical study of the effects of these errors. The basic assumptions of the model are: (i) independently designed software components are chosen in a random sample and (ii) in the user environment, the system is required to execute on a stationary input series. An intensity function, called the intensity of coincident errors, has a central role in the model. This function describes the propensity of a population of programmers to introduce design faults in such a way that software components fail together when executing in the user environment. The model is used to give conditions under which an N-version system is a better strategy for reducing system failure probability than relying on a single version of software. In addition, a condition which limits the effectiveness of a fault-tolerant strategy is studied, and we ask whether system failure probability varies monotonically with increasing N or whether an optimal choice of N exists.

[Ehre76] Abstract: The number of tests, which are necessary to prove the performance of a program, can be reduced to an executable number, if the structure of the program is investigated. The analysis starts from the memory dump. The program is first divided into those pieces, which are without labels or branchings. Then the mappings of the program and their input and output areas are identified, further those areas which influence branchings. The next step states which ranges of values in the individual areas are distinguished by the program and which junctions of areas are relevant. From this, the kind and the number of the necessary tests can be derived. By means of observing their main variables loops are divided into simpler structures

The method has been applied for the verification of the user programs of the protection system of the 800 ME boiling water reactor plant in Brunsbuttel.

[Ehrl87] Abstract: A number of time-domain software reliability models attempt to predict the growth of a system's reliability during the system test phase of the development life cycle. In this paper we examine the results of applying several types of Poisson-process models to the development of a large system for which system test was performed in two parallel tracks, using different strategies for test data selection. We show that the reliability growth predicted by non-homogeneous Poisson process models was found for only one of these testing strategies. These results imply that the applicability of a reliability growth model to a given software development project will depend on the nature of that project's system test process; they also raise theoretical questions about the assumption of certain statistical properties for failure occurrence during testing.

[Elme69] Abstract: Functional testing of operating systems is in transition from a predominantly imprecise art to an increasingly precise science. The process that controls this testing is maturing correspondingly. The laissez-faire approach is giving way to a disciplined approach characterized by rigorous definition of the test plan, systematic control of the test effort, and objective quantitative measurement of the test coverage. This paper describes just such a disciplined test control process, which is composed of five steps: 1) the survey, which establishes the intended extent of testing; 2) the identification, which creates a list of functional variations eligible for testing; 3) the appraisal, which ranks and subsets the eligible variations so that test resources can be directed a those with the higher payoff; (4) the review, which calculates the test coverage of the test case library; and 5) the monitor, which verifies attainment of the planned testing coverage. Throughout the test process, specification testing is distinguished from program testing.

[Elme71] Abbreviated Abstract: This paper discusses some lessons [the author has] learned from testing large,

complex software systems. Actually, [the author] believe these lessons are equally applicable to small software packages. While the large systems are admittedly particularly vulnerable to error, the small systems may have many more users and thus the impact of error can be equally great.

[The author] will be addressing functional testing but not performance testing. Just as performance testing involves space and time measures of system utilization, so functional testing involves spatial and temporal measures of system quality.

[Elme73] Abstract: This paper describes a partially automated and disciplined process for testing the equivalence between a functional specification and its implementation in software, firmware or hardware. The semantic content of a specification expressed in a natural language is restated in boolean graph form as a logical relationship between causes and effects. This cause-effect graph is processed to yield (1) a list of the functional primitives, or variations, (2) the definition of a syntactically feasible test library which will efficiently validate these variations, and (3) the faulty variations detectable by each test. The correspondence between functional variations and decision table rules is also addressed.

[Elsh76b] Abstract: The source code for 120 production PL/I programs from several General Motors' commercial computing installations has been collected. The programs have been scanned both manually and automatically. Some data from the scanning process are presented and interpreted.

The programs are considered with respect to five attributes: 1) the size of the programs, 2) the readability of the programs, 3) the complexity of the programs, 4) the discipline followed by the programmers, and 5) the use of the programming language. Each areas is reviewed with pertinent data presented whenever it is available.

The report should be of interest to anyone involved with programming. The report helps explicitly identify some areas of programming in which a better job could be done. Although the programs analyzed are written in PL/I, those persons from installations using other languages, particularly Cobol, have indicated that the information presented is typical.

[Elsh78c] Abstract: The October 1988 issue of SIGPLAN Notices carries an article that compares functionally equivalent programs that differ in their internal structure. The basis for comparing the programs is a measure called cyclomatic complexity whose value is the cyclomatic number of the graph that corresponds to the flow of control of the program. One program is of particular interest since all of the well-structured versions of the program that are discussed have a higher cyclomatic complexity than the unstructured version. In this paper another well-structured version of the program is presented for which the cyclomatic complexity is reduced to that of the original unstructured version. In the process, some of the shortcomings of the cyclomatic number as a complexity measure are revealed.

[Elsh84] Abstract: Twenty program complexity measures are studied with respect to how well they identify the more complex procedures in a software system. The measures have been applied to three large sets of PL/I procedures representing three different types of applications. Four of these complexity measures have been found to form a characteristic set. That is, when procedures are kept within reasonable bounds for the four selected measures, they will most likely be within reasonable bounds for all of the other measures. The measures and their interpreted meanings are:

- length - the quantity of source code,
- unique operators - the variety of programming language actions,
- data difficulty - the average number of variable appearances, and
- unique operands - the variety of constants and variables.

[Elsp72a] Abstract: The purpose of this paper is to point out the significant quantity of work in progress on techniques that will enable programmers to prove their programs correct. This work has included: investigations in the theory of program schemas or abstract programs; development of the art of the informal or manual proof of correctness; and development of mechanical or semi-mechanical approaches to proving correctness. At present, these mechanical approaches rely upon the availability of powerful theorem-provers, development of which is

being actively pursued. All of these technical areas are here surveyed in detail, and recommendations are made concerning the direction of future research toward producing a semi-mechanical program verifier.

[Emde81] Abstract: Our goal is to obtain a specification of a relational data base as an abstract data type in such a way that a computer program can simulate on a small scale the inteded use of the data base by generating formal consequences of the specification (that is, without the existence of any implementation of the data base). There are two candidates for the specification formalism to be used: equations and the Horn clauses of logic.

Apart from a specification of a relational data base, the paper is devoted entirely to a comparison between equations and clauses. We compare three aspects: mathematical semantics, the computational aspects, and expressiveness. We propose to discard equations as a distinct formalism, but will regard them as a special case of clauses. In principle we use as specification a clausal sentence containing literally the equations conventionally used in data type specification, but we find certain slight departures conducive to clarity. As a program (to be executed by a PROLOG processor) we use another sentence obtained from the specification by a translation process that guarantees correctness.

[Emer84] Abstract: The decomposition of a large program into modules can be guided by the use of a property called cohesion, first described by Constantine. Cohesion is a quality that describes the degree to which the different actions performed by a module contribute to a unified function. However, this technique may be difficult to apply due to the subjective nature of the definitions of levels of cohesion. In this paper a software metric is defined and proposed as a discriminant for classifying modules according to their cohesion. Formal properties of the metric are derived which can be used to set the metric value ranges for module classification.

[Endr75] Abstract: Program errors detected during internal testing of the operating system DOS/VS form the basis for an investigation of error distributions in system programs. Using a classification of the errors according to various attributes, conclusions can be drawn concerning the possible causes of these errors. The information thus obtained is applied in a discussion of the most effective methods for the detecting and prevention of errors.

[Eric85] Abstract: Deployment of software controlled systems for providing communications services has grown very rapidly. For example, a large proportion of telephone central offices are now Stored Program Control Systems (SPCS). In the course of this growth, it has been found that software, like hardware, is subject to various kinds of problems throughout the software life cycle which may seriously affect the software cost, intended delivery date, field performance, and inservice support. As a result, Bell Communications Research, Inc. (BELLCORE) has proposed generic software reliability and quality requirements for telecommunications software to meet typical telephone company needs. These requirements are intended to reduce software life cycle costs by assuring that the software is designed, developed, tested, produced, installed, and supported in a manner that is consistent with modern software quality concepts and practices.

[Evan83a] Abstract: Several studies have appeared in recent years examining the sensitivity of standard software complexity metrics to common rules of program structuring. In most cases, these studies found support for the use of certain metrics as indices of program quality as represented by program structure. In the research described in this paper, a broader analysis of metric sensitivity to the structuring rules was conducted. The conclusions reached differ greatly from those previously advocated in the literature; i.e., the metrics under consideration are shown to be relatively insensitive to program structure.

[Evan83b] Abbreviated Introduction: In their study of the psychological complexity of software, Curtis, *et. al.*, remark that "no simple relationship between computational and psychological complexity is expected." In the discussion below, we elaborate on this observation through a series of examples.

[Evan84a] Abstract: The complexity of control flow in a program is generally believed to be an important determinant of the testability and the comprehensibility of the program. Several metrics have been proposed to measure this aspect of complexity, including the nesting level metric of Harrison and Magel and the cyclomatic

complexity of McCabe. In this paper, the theory underlying cyclomatic complexity is analyzed and shown to be poorly developed, and the nesting level metric is reconstructed from a simpler conceptual basis. In each case, the emphasis is on the need to provide software metrics with an adequate theoretical foundation.

[Evan84b] Abbreviated Preface: This book guides the software manager through the software testing morass. The book will identify the individual components and test levels that must be integrated into a cohesive structure, and outline how the testing program is to be planned and managed. It will identify tools, techniques, and methodologies that must be incorporated if testing is to succeed.

The book offers solutions to the recurring management problems characteristic of software testing, which invariably turn into crises during the later stages of a software project. Problems all center around a single theme: The project planners have not adequately estimated what will occur during the testing period and cannot control resources and activities during this critical implementation stage.

[Evan84c] Abstract: In a recent paper the author has presented evidence that, contrary to several studies in the literature, certain software complexity metrics are not consistently sensitive to the application of program style rules. In the present paper, these results are summarized and extended to two additional metrics. The new results indicate both that current complexity metrics are improper indices of program quality, as measured by style, and that many commonly used style rules do not address questions of minimizing interprocedural information flow complexity.

[Faga76] Abstract: Substantial net improvements in programming quality and productivity have been obtained through the use of formal inspections of design and of code. Improvements are made possible by a systematic and efficient design and code verification process, with well-defined roles for inspection participants. The manner in which inspection data is categorized and made suitable for process analysis is an important factor in attaining the improvements. It is shown that by using inspection results, a mechanism for initial error reduction followed by ever-improving error rates can be achieved.

[Faga86] Abstract: This paper presents new studies and experiences that enhance the use of the inspection process and improve its contribution to development of defect-free software on time and at lower costs. Examples of benefits are cited followed by descriptions of the process and some methods of obtaining the enhanced results.

Software inspection is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product-and at lower cost-than does machine testing. Users of the method report very significant improvements in quality that are accompanied by lower development costs and greatly reduced maintenance efforts. Excellent results have been obtained by small and large organizations in all aspects of new development as well as in maintenance. There is some evidence that developers who participate in the inspection of their own product actually create fewer defects in future work. Because inspections formalize the development process, productivity and quality enhancing tools can be adopted more easily and rapidly.

[Fair75] Abstract: This paper describes an experimental program testing facility called the interactive semantic modeling system (ISMS). The ISMS is designed to allow experimentation with a wide variety of tools for collecting, analyzing, and displaying testing information. The design methodology is applicable to procedural programming languages, and Algol 60 is being used as the vehicle for elaboration of design principles and implementation techniques.

This paper discusses the ISMS design, and describes the various types of analysis and display tools being developed to facilitate program testing. The ISM Preprocessor is described, and an example is presented to illustrate the data structures utilized in the ISMS.

[Fair79] Abstract: ALADDIN is an interactive facility for debugging and testing of assembly language programs. ALADDIN differs from traditional debuggers by allowing the user to specify breakpoint assertions,

rather than breakpoint locations. Assertions are logical relations among various components of the program state. If an assertion becomes false during execution of the object program a breakpoint is executed and control is passed to the user's terminal. ALADDIN can also be used as a testing tool to verify that asserted behavior matches actual behavior under various sets of input data and test conditions.

[Farr83] Abstract: With the ever-increasing role that software is playing in the weapon systems, a great need has arisen for tools that are useful in developing cost-effective software. An area of research has arisen over the last 10 years in providing a software manager quantitative statements about the reliability of the software. Using this quantitative measure, the manager can make a determination of when software testing should terminate and how to best utilize testing personnel. This report discusses the various approaches that have been advocated for reliability estimation. It reviews the various model assumptions, the estimates of reliability, the precision of those estimates, and the data required for their implementation. A comparison is then made among some of these models based upon studies that have been done. General comments concerning software reliability implementation are discussed in the final section of the report.

[Farr88] Abstract: The concept of software reliability and its measurement is receiving a lot of attention in the software development community. With the ever increasing role that software is playing in today's and tomorrow's society, software developers and users are asking: "Just how 'good' is the software?" and "how much testing should be done before the software is released?" The software reliability methodology attempts to provide quantitative measures to help answer these questions. Unfortunately, to arrive at these measures requires complex numerical computations, usually requiring the assistance of a computer. A software reliability analysis tool called the "Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS)" was developed several years ago for this purpose. Originally, the tool was designed for a mainframe/mini computer environment. This paper describes an adaptation of that tool for the personal computer (PC) and relates how it differs from the one for larger computer systems.

The PC version of SMERFS includes several features which are illustrated in this paper, using as data either time-between-error occurrence (wall clock or central processing unit) or error counts per-time-period. These features are data input and data management, editing capability, data transformations, model fitting for software reliability measurement, and features of the software that allow the user to determine the adequacy of fit as well as aids in determining the best model. The current version of SMERFS has incorporated eight software reliability models. These models include the following: John Musa's Execution Model, Goel's Non-homogeneous Poisson Models using both types of data, the Geometric Model, a Generalized Poisson Model, Schneidewind's Model, and Brooks and Motley's Model.

[Feat89] Abstract: Constructing specifications of complex tasks is often a laborious activity in spite of the rich vocabulary provided by specification languages. An incremental approach to construction is proposed, with the virtue of offering considerable opportunity for mechanized support. Following this approach one builds a specification through a series of elaborations that incrementally adjust a simple initial specification. Elaborations perform both refinements, adding further detail, and adaptations, retracting over simplifications and tailoring approximations to the specifics of the task. It is anticipated that the vast majority of elaborations can be concisely described to a mechanism which will then perform them automatically. When elaborations are independent, they can be applied in parallel, leading to diverging specifications which must later be recombined.

The approach is intended to facilitate comprehension and maintenance of specifications, as well as their initial construction. The advantages of following this approach stem from the gradual nature of the elaboration process, the separation of concerns through following independent elaborations in parallel, the simplicity of the individual elaboration steps (the effects of each step are well delineated), and the availability of an explicit record of construction.

[Feld89] Abstract: The advent of high-resolution graphics workstations at reasonable cost offers great potential in the development of high-level, graphics-oriented debugging tools. The advent of programming languages, Ada, in particular, which support concurrency with high-level primitives, affords the opportunity to develop new

models of debugging for programs incorporating concurrent tasks. The marriage of graphics and concurrency-oriented debugging can provide powerful tools indeed.

We have development a demonstration-quality graphics-assisted debugger for intertask communication in Ada. Based on the static task-specification diagrams of Booch, the debugger animates the activity of a collection of communicating tasks, and runs on a DEC GIGI terminal connected to a VAX 11-780 under TeleSoft's Ada compiler.

The model has been subjected to empirical validation, using under-graduate students as experimental subjects. Subjects were required to debug erroneous tasking programs using both the graphical debugger and a textual one.

[Ferr77] Abstract: State machines provide a convenient and indispensable mathematical framework for defining precise specifications of complex software systems. Such specifications stand as pivotal elements between requirements and designs, and permit the definition of three levels of correctness in software systems, namely 1) interpretation correctness between requirements and specification, 2) program correctness between specification and design, and 3) implementation correctness between design and programmed hardware. Formal techniques already exist for program correctness and implementation correctness. However, there is a need for formal techniques for interpretation correctness; methods of semantics are proposed as a basis for such techniques.

[Fetz88] Abbreviated Introduction: The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.

[Feue79a] Abstract: It is no longer a surprise that the program maintenance dominates the total cost of a large software system over its lifetime. In response to these costs, the emphasis in program design has largely shifted from the time and space issues of machine efficiency to issues of clear and flexible program structures that can be easily maintained.

The goal of this project is to identify measurable program properties that influence maintainability. More precisely, we examine the effect of various program characteristics on the subsequent frequency and magnitude of program errors.

[Fisc77] Abstract: The literature in the areas of software management and software engineering admit to a possible reduction in the reliability of software after modifications have been made. Validation of maintenance modifications is commonly referred to as retest, and has yet to be adequately resolved. The problem is how to efficiently select previously run test cases to be rerun on the software to assure no degradation of reliability. This paper develops several alternative retest philosophies and identifies a common operations research technique for solution. Detailed examples show how 0-1 integer programming can identify a minimum number of previously executed tests necessary to fully retest every affected program element at least once. Use of this model to determine proper selection of test cases can reduce the cost of software maintenance and increase confidence in the reliability of the code.

[Fitz78a] Abstract: During recent years, there have been many attempts to define and measure the "complexity" of a computer program. Maurice Halstead has developed a theory that gives objective measures of software complexity. Various studies and experiments have shown that the theory's predictions of the number of bugs in programs and of the time required to implement a program are amazingly accurate. It is a promising theory worthy of much more probing scientific investigation.

This paper reviews the theory, called "software science," and the evidence supporting it. A brief description of a related theory, called "software physics," is included.

[Flon77] Abstract: This thesis applies and extends mathematical program verification to systems programs. The design methodology is based upon the use of abstract data types and the construction and verification of both

specifications and implementations for them. The abstract data type is a means of modularization which encapsulates the representation of a data structure and the algorithms which operate directly upon it. The specification technique appeals to various mathematical structures (e.g. sets and sequences) to describe an abstract state for objects of a given type. The correctness of the formal specifications is cast in terms of the proof rule is given to formulate the theorems necessary for proving the invariance of predicates across formal specifications. The applicability of the methodology to operating systems is explored. It is found that a hierarchical decomposition is most amenable to verification, and that the implementation language used is a function of that hierarchy. The example of a process dispatcher module of a hypothetical operating system is used to illustrate the process of design, specification, implementation, and verification using the methodology. Various properties are proven of the abstract specifications, including one representation of the concept of fair service. Programs are then written for the specifications and their correctness is verified.

[Flon78a] Abstract: We present weakest pre-conditions which describe weak correctness, blocking, deadlock, and starvation for nondeterministic programs. A procedure for converting parallel programs to nondeterministic programs is described, and the correctness of various example parallel programs is treated in this manner. Among these are a busy-wait mutual exclusion scheme, and the problem of the Five Dining Philosophers.

[Flon81] Abstract: We describe a formal theory of the total correctness of parallel programs, including such heretofore theoretically incomplete properties as safety from deadlock and starvation under fair-scheduling. We present a sound and complete set of proof rules for the total correctness of parallel programs expressed in nondeterministic form.

The proof of soundness and completeness is novel in that we show that the weakest pre-conditions for the correctness criteria are actually fixed-points (least or greatest) of continuous functions over the complete lattice of total predicates. We have obtained proof rule schemata which can universally be applied to least or greatest fixed-points of continuous functions. Therefore, a system of proof rules is a priori sound and complete once it is shown that certain weakest pre-conditions are extremum fixed-points. The relationship between true parallelism and nondeterminism is also discussed.

[Form77] Abstract: In this article we introduce the problem of computer software reliability and discuss a probabilistic model for describing the failure of software. We suggest a procedure for estimating the parameters of the model and propose a stopping rule for debugging the software. We apply our procedure to some published data on software failures.

[Form79] Abstract: This paper discusses certain stochastic aspects of the software reliability problem. First an empirical stopping rule for debugging and testing computer software is discussed. Then some results are presented on choosing a time interval for testing the hypothesis that a software system contains no errors, given certain costs and risk constraints.

[Fosd76a] Abstract: The ways that the methods of data flow analysis can be applied to improve software reliability are described. There is also a review of the basic terminology from graph theory and from analysis in global program optimization. The notation of regular expressions is used to describe actions on data for sets of paths. These expressions provide the basis of a classification scheme for data flow which represents patterns of data flow along paths within subprograms and along paths which cross subprogram boundaries. Fast algorithms, originally introduced for global optimization, are described and it is shown how they can be used to implement the classification scheme. It is then shown how these same algorithms can also be used to detect the presence of data flow anomalies which are symptomatic of programming errors. Finally, some characteristics of an experience with Dave, a data flow analysis system embodying some of these ideas, are described.

[Fosd76b] Abstract: In an earlier paper, the authors have defined type 1 and type 2 data flow anomalies to be, respectively, the reference to an undefined variable and the definition of a variable without subsequent reference. It is not difficult to devise search techniques to detect such anomalies when the anomalous data flow is contained

in a single procedure. When the data flow crosses procedure boundaries, however, many difficulties may arise. In this paper, we carefully define the conditions under which interprocedural anomalies occur. We also show how algorithms currently used in global program optimization can easily be adapted to yield highly efficient algorithms for the detection of such interprocedural anomalies.

[Fost80] Abstract: A hardware failure analysis technique adapted to software yielded three rules for generating test cases sensitive to code errors. These rules, and a procedure for generating these cases, are given with examples. Areas for further study are recommended.

[Fost83] Introduction: The subject paper uses the phrase "error-sensitive" and includes the [initial] ESTCA paper in its references. The latter paper uses the same phrase to describe a different method. This comment challenges the claim that domain t sting is a "more promising" error detection strategy. Figures in the subject paper are referenced but not reproduced. Therefore the comment cannot be verified without that paper at hand.

[Fost84] Abstract: The usual approach to testing software with logic expressions considers that n variables can realize 2^(2^2n) functions. Therefore, to distinguish one Boolean expression in n variables (a "correct" one) from all others ("errors") 2^n tests are necessary.

[Fost85] Abbreviated Introduction: This note revises and re-substantiates the original ESTCA rule [Fost80] for generating or evaluating test data for simple conditional or comparison expressions. Comments on other methods are included.

The rules apply to the lowest program component - "variable operator variable-or-constant" expressions. Clearly, if any part of any expression is incorrect, then so is the program. Rules developed in this continuing study are intended to identify data that meets error sensitivity criteria for all expression types. Rules for logic expressions and combinations of comparisons are in [Fost84].

[Fran80] Abstract: Discussed is a distributed system based on communication among disjoint processes, where each process is capable of achieving a post-condition of its logic space in such a way that the conjunction of local post-conditions implies a global post-condition of the whole system. The system is then augmented with extra control communication in order to achieve distributed termination, without adding new channels of communication. The algorithm is applied to a problem of constructing a sorted partition.

[Fran85a] Abstract: This paper describes ASSET, a tool which uses information about a program's data flow to aid in selecting test data for the program and to evaluate test data adequacy. ASSET is based on the family of data flow test selection and test data adequacy criteria developed by Rapps and Weyuker. ASSET accepts as input a program written in a subset of Pascal, a set of test data, and one of the data flow adequacy criteria and indicates to what extent the criterion has been satisfied by the test data.

[Fran86] Abstract: Most test data adequacy criteria based upon path selection have the unfortunate property that for some programs with unexecutable paths, no set of test data is adequate. In this paper we define a new family of adequacy criteria, derived from the data flow testing criteria, which circumvent this problem by only requiring the test data to exercise those definition-use associations which are executable. The inclusion relationship among these criteria is explored.

[Fran88] Abstract: A test data adequacy criterion is a predicate which is used to determine whether a program has been tested "enough." An adequacy criterion is applicable if for every program there exists a set of test data for the program which satisfies the criterion. Most test data adequacy criteria based on path selection fail to satisfy the applicability property because, for some programs with unexecutable paths, no adequate set of test data exists. In this paper, we extend the definitions of the previously introduced family of data flow testing criteria to apply to programs written in a large subset of Pascal. We then define a new family of adequacy criteria called feasible data flow testing criteria, which are derived from the data flow testing criteria. The feasible data

flow testing criteria circumvent the problem of nonapplicability of the data flow testing criteria by requiring the test data to exercise only those definition-use associations which are executable. We show that there are significant differences between the relationships among the feasible data flow testing criteria.

We also discuss a generalized notion of the executability of a path through a program unit. A script of a testing session using our data flow testing tool, ASSET, is included in the Appendix.

[Fuji77] Abstract: Verification is presented as a method of ensuring high reliability of software systems. Verification consists of an early analysis of program requirements and design specifications, followed by extensive program analysis and system/program execution testing. It is performed in parallel with software development by an organization independent of the development group, with the objective of detecting conceptual and implementation errors before program acceptance. Software analysis and testing aids for cost-effectively automating routine tasks are described. The results of several verification projects are discussed to illustrate common types of errors and techniques for their detection. Key aspects of verification planning are presented for projects that are required to achieve highly reliable software.

[Gabo76] Abstract: In this paper we analyze the complexity of algorithms for two problems that arise in automatic test path generation for programs: the problem of building a path through a specified set of program statements and the problem of building a path which satisfies impossible-pairs restrictions on statement pairs. These problems are both reduced to graph traversal problems. We give an efficient algorithm for the first, and show that the second is NP-complete.

[Gaff79] Abstract: McCabe has shown that the complexity of a computer program may be defined as the cyclomatic number of the graph theoretic representation of its control structure and that that number (less one) is equal to the number of conditional jumps (J) in the structure. It can be shown that a range of values for this figure can be derived from an estimate of the number of inputs and outputs in a program, such as obtainable from its specification. *If* ≠ Inputs = = Outputs = N, then the least value of J ($=J_{min}$) = N $\log_2$ N. This figure is derived from information theoretic arguments. It is noted to be equal to the least number of switches in a telephone exchange and to the least number of operations in a sort. If we know that a program whose size is to be estimated is expected to be similar to others in which the proportion of conditional jumps is "L," then the minimum number of instructions would be equal to L*J_{min}, while the more likely number would be L*J_{max}, where J_{max} = N^2. This formulation is compared with several of Halstead's formulas which can be used to estimate the size of a program using estimates of the number of input/output parameters, the numbers of operators, and the language level. Comparisons are made using the new "complexity"-based estimators, and those of Halstead.

[Gaff81a] Abstract: This paper describes some of the potential for applying software metrics to the management of the software development process. It also considers some of the practical difficulties one typically faces in evolving and validating a software metric. One difficulty is the collection of baseline data in the real world of software production in which controlled experiments typically are not possible. The results of some recent quantitative 'metrics' investigations are presented and their practical implementations for software estimation and control are cited. These investigations are thought to be representative of the process of evaluating software data not obtained under 'controlled' conditions such as is typically the situation in the natural science laboratory.

[Gaff81b] Abstract: The nature of "software quality" and some software metrics are defined and their relationship to traditional software indicators such as "maintainability" and "reliability" are suggested. Recent work in the field is summarized and an outlook for software metrics in quality assurance is provided. The material was originally presented as a tutorial at the "ACM SIGMETRICS Workshop/Symposium of Measurement and Evaluation of Software Quality" on March 25, 1981.

[Gaff88] Abstract: Availability is a significant measure of software performance. A system's availability is a function of the availability of its software component which is directly related to the number of errors remaining in it at delivery, the latent error content. This paper presents a method for estimating the latent error content of

an element of software; this can be done commencing with data obtained during design and code inspections. The availability of the software unit, then, is a function of the rate of discovery of these errors.

[Gall81] Abstract: In the second part of this work, the author formulates a new inductive assertion method applying to the class of nondeterministic flowchart programs with recursive procedures studied in part 1. Using results on unfolding proved in part 1, he proves that this method is sound and complete with a finite number of assertions. He studies four notions of correctness: two notions of partial correctness (existential and universal) and the corresponding notions of total correctness. He also formalizes two notions of extension and equivalence (existential and universal) in the second-order predicate calculus.

[Gann75] Abstract: The language in which programs are written can have a substantial effect on their reliability. This paper discusses the design of programming languages to enhance reliability. It presents several general design principles, and then applies them to particular languages constructs. Since we can not logically prove the validity of such design principles, empirical evidence is needed to support or discredit them. Gannon has performed a major experiment to measure the effect of nine specific language-design decisions in one context. Analysis of the frequency and persistence of errors shows that several decisions had a significant impact on reliability.

[Gann76] Abstract: The goal of reliable programming is to minimize the number of errors in completed programs. The language in which programs are written can have a significant impact on the reliability of the programming process.

One common language feature that appears in different forms in many programming languages is the data type. Data types may be associated with operands in one of three ways: statically typed operands. Furthermore, programmers trying to convert an operand from one type to another, who were forced to grapple with the representation of the operand, committed errors that cast doubt upon the ability of programmers to write reliably in a language which treats its operands as collections of bits.

This preliminary data suggests that a more detailed and controlled experiment will be required to enable us to draw firm conclusions about the role of data types in reliable programming. A proposal for such an experiment is described in the paper.

[Gann77] Abstract: The language in which programs are written can have a substantial effect on the reliability of the resulting programs. This paper discusses an experiment that compares the programming reliability of subjects using a statically typed language and a "typeless" language. Analysis of the number of errors and the number of runs containing errors shows that, at least in one environment, the use of a statically typed language can increase programming reliability. Detailed analysis of the errors made by the subjects in programming solutions to reasonably small problems shows that the subjects had difficulty manipulating the representation of data.

[Gann79] Abstract: Two software testing techniques-static analysis and dynamic path (branch) testing-are receiving a great deal of attention in the world of software engineering these days. However, empirical evidence of their ability to detect errors is very limited, as is data concerning the resource investment their use requires. Researchers such as Goodenough and Howden have estimated or graded these testing methods, as well as such other techniques as interface consistency, symbolic testing, and special values testing. However, this paper seeks to demonstrate *empirically* the types of errors one can expect to uncover and to measure the engineering and computer time which may be required by the two testing techniques for each class of errors during system-level testing.

[Gann80] Abbreviated Introduction: A series of articles have made the data type "traversable stack" something of a cause celebre. At the University of Maryland we are constructing a system called DAISTS (Data Abstraction Implementation, Specification, and Testing System) for testing data abstraction implementations, and here report how DAISTS fared on two traversable stack specifications. We concluded that subtle errors (by definition those undetected by an author in her or his own work) are sometimes easy to discover through testing, but that other, more obvious mistakes may slip by tests.

[Gann81] Abstract: A compiler-based system DAISTS that combines a data-abstraction implementation language (derived from the SIMULA class) with specification by algebraic axioms is described. The compiler, presented with two independent syntactic objects in the axioms and implementing code, compiles a "program" that consists of the former as test driver for the latter. Data points, in the form of expressions using the abstract functions and constant values, are fed to this program to determine if the implementation and axioms agree. Along the way, structural testing measures can be applied to both code and axioms to evaluate the test data. Although a successful test does not conclusively demonstrate the consistency of axioms and code, in practice the tests are seldom successful, revealing errors. The advantage over conventional programming systems is three-fold:

1. The presence of the axioms eliminates the need for a test oracle; only inputs need be supplied.
2. Testing is automated: a user writes axioms, implementation, and test points; the system writes the test drivers.
3. The results of tests are often surprising and helpful because it is difficult to get away with "trivial" tests: what is not significant for the code is liable to be severe test of the axioms, and vice versa.

[Gann85] Abstract: Packages are one of the primary features of Ada. They can be used to group declarations or subprograms or to create new encapsulated types. Metrics are presented which characterize the use of Ada packages, indicating where program structure may make changes difficult, and suggesting how the structure may be improved. The use of such metrics should aid in the transition to, and better use of Ada. The metrics are applied to examples of a ground-support satellite system.

[Gann86] Abstract: Modules allow programmers to group related data and/or procedures and to limit the amount of information that is accessible to the rest of the program. Splitting a program into modules should localize the effects of program changes to correct errors or to improve the implementation (i.e., making it more robust or more efficient). In addition, since modules are usually self-contained, they can be reused from project to project. The designers of Ada recognized three major uses for modules:

1. A named collection of declarations that makes a group of types and variables available much like a Fortran common block;
2. A group of related subprograms that provides a library facility;
3. An encapsulated data type that provides the names of the type and it operations, but hides the details of the representation of objects of the type and implementation of the type's operations.

While the first two uses are familiar to many programmers, the third use is not supported by many commonly used programming languages. Strong syntactic clues are available to help programmers decide what objects comprise the first two kinds of modules (e.g., all types and constants, a collection of global variables, or a set of utility routines), but fewer hints are available to aid in grouping objects in problem-oriented terms. Deciding what objects to encapsulate in a system is a formidable challenge.

[Garc84] Abstract: In this paper we discuss the issues involved in debugging a distributed computing system. We describe the major differences between debugging a distributed system and debugging a sequential program. We suggest a methodology for distributed debugging, and we propose various tools or aids.

[Garm81] Discussion of the software problem which delayed the first Shuttle orbital flight.

[Geig79] Abbreviated Introduction: The validation strategy presented here can be considered as a first step toward proving the correct functioning of a real-time software system, in this case for an advanced computerized nuclear reactor protection system. It may also serve as a guideline for the systematic validation and testing of other safety oriented systems.

[Gell78] Abstract: Proofs of program correctness tend to be long and tedious, whereas testing, though useful in detecting errors, usually does not guarantee correctness. This paper introduces a technique whereby test data can be used in proving program correctness. In addition to simplifying the process of proving correctness, this method simplifies the process of providing accurate specification for a program. The applicability of this

technique to procedures and recursive programs is demonstrated.

[Gelp79] Introduction: Testing, as practiced today, is almost exclusively concerned with the verification of presently-required function. The purpose of this note is to focus on future function, i.e. change, and to propose that the concerns of testing be broadened to include maintainability. What follows is meant to clarify this proposal and to suggest some maintenance testing methodologies in order to stimulate research.

[Gelp88] Abbreviated Introduction: We can trace the evolution of software test engineering by examining changes in the testing process model and the level of professionalism over the years. The current definition of a good software testing practice involves some preventive methodology.

[Gerh76a] Abstract: Errors, inconsistencies, or confusing points are noted in a variety of published algorithms, many of which are being used as examples in formulating or teaching principles of such modern programming methodologies as formal specification, systematic construction, and correctness proving. Common properties of these points of contention are abstracted. These properties are then used to pinpoint possible causes of the errors and to formulate general guidelines which might help to avoid further errors. The common characteristic of mathematical rigor and reasoning in these examples is noted, leading to some discussion about fallibility in mathematics, and its relationship to fallibility in these programming methodologies. The overriding goal is to cast a more realistic perspective on the methodologies, particularly with respect to older methodologies, such as testing, and to provide constructive recommendations for their improvements.

[Gerh76b] Abstract: Backtracking is a well-known technique for solving combinatorial problems. It is of interest to programming methodologists because 1) correctness of backtracking programs may be difficult to ascertain experimentally and 2) efficiency is often of paramount importance. This paper applies a programming methodology, which we call control structure abstraction, to the backtracking technique. The value of control structure abstraction in the context of correctness is that proofs of general properties of a class of programs with similar control structures are separated from proofs of specific properties of individual programs of the class. In the context of efficiency, it provides sufficient conditions for correctness of an initial program which may subsequently be improved for efficiency while preserving correctness.

The paper provides several abstract variations of backtracking programs, along with correctness statements and assertions, and an overall parameterization of the backtracking technique to facilitate selection of the appropriate variant abstraction for a concrete problem. The methodology is illustrated on the eight queens, knight's tour, malicious secretary, and good sequences problems. Also discussed are the amount of work involved in the control structure abstraction approach for this particular application area, its relationship to the data structure abstraction method, and its possible application to other areas.

[Gerh80] Abstract: AFFIRM is an experimental system for the algebraic specification and verification of abstract data types and Pascal-like programs using these types. The heart of the system is a natural deduction theorem prover for the interactive proof of verification conditions and properties of data types. Additional features include tools for the analysis of algebraic specifications, verification of small programs, the specification and partial proof of a large file updating module, and the proof of high level properties of protocols and security kernels.

[Gerh84] Abstract: The goal of this paper was to model a specification language and its analyzer using axiomatic methods derived from those applied previously to abstract data type and state transition specifications. The models attempt to cover many interesting features of PSL/PSA, a widely used specification language and analyzer for information systems. Simple properties expected to hold for actual PSL/PSA were formalized and proved about some models, with assumptions about undefined parts. Both model formulation and property proofs were performed within the AFFIRM Specification and Verification System. The results show (1) the applicability of axiomatic methods for modeling a new kind of software system, (2) insights into the PSL/PSA class of specification system, (3) a possible route for formal definition of such analyzers, and (4) additional

lessons on the art of specification, modeling, verification, and validation.

[Gerh88a] Abstract: This paper describes a suite of tools to support analysis of properties of sequences associated with a specification, with input or output to a program, or with simple behavioral models of a system under design. The toolset's capabilities include: generating sequences to satisfy combinations of conditions, organizing these condition combinations as tables of cases to serve as test data, and visualizing the effects of executing a chosen sequence. The technology base is Prolog extended with a powerful window package.

[Gerh88b] Position Statements Included:
1. Gaudel, M-C., and B. Marre. "Generation of Test Data from Algebraic Specifications."
2. Wild, C. "Generic Constraint Logic Programming and Incompleteness in the Analysis of Software."

[Germ82a] Abstract: Most high level languages with multiprocessing do not have built in mechanisms to detect deadlocks during program execution. We present transformation rules for taking an original Ada program P and deriving a new program P ', such the P ' has a potential deadlock if P does, and P ' signals whenever a deadlock is about to occur. In principle, the transformations can be applied mechanically, giving a practical tool for debugging deadlocks. Since this method modifies the source program, it can be used with any implementation of the language, without special knowledge of the implementation of tasking. The transformations that we have developed thus far are sufficient to handle most of the complexities of Ada tasking, including arbitrary task types, conditional entry calls, selective waits, timed entry calls, and intertask exceptions.

In the course of this work, we have developed some generally useful source program transformations, such as one to uniformly introduce task identifiers. We have also developed some interesting concurrent algorithms for the deadlock monitoring.

An actual monitor program for detecting deadlocks has been implemented in Ada. Our basic approach and monitoring algorithms are applicable to other languages with multiple processes.

[Germ84] Abstract: We present a deadlock monitoring algorithm for Ada tasking programs which is based on transforming the source program. The transformations introduce a new task called the monitor, which receives information from all other tasks about their tasking activities. The monitor detects deadlocks consisting of circular entry calls as well as some noncircular blocking situations. The correctness of the program transformations is formulated and proved using an operational state graph model of tasking. The main issue in the correctness proof is to show that the deadlock monitor algorithm works correctly without having simultaneous information about the state of the program.

In the course of this work, we have developed some useful techniques for programming tasking applications, such as a method for uniformly introducing task identifiers.

We argue that the ease of finding and justifying program transformations is a good test of the generality and uniformity of a programming language. The complexity of the full Ada language makes it difficult to safely apply transformational methods to arbitrary programs. We discuss several problems with the current semantics of Ada's tasks.

[Getz83] Abstract: A very high-level trace for data structures is one which displays a data structure in the shape in which the user conceptualizes it, be it a tree, an array, or a graph. GRAPHTRACE is a system that facilitates the very high-level graphic display of interrelationships among dynamically allocated Pascal records. It offers the user a wide range of options to enable him to "see" the data structures on a graphics screen in a format as close as possible to that in which he visualizes it, thereby providing a useful display capability when the user's conceptual model is a directed graph or tree.

The system is interactive, allowing the user to refine his plotting instructions stepwise. He may specify different combinations of pointer directions, omit certain records, and select a root record if desired. As an additional diagnostic aid, the user may dump the contents of specified records in a format as close as possible to the original source code in which they were defined.

The system is written in Pascal. It consists of a precompiler as well as various coordinate assignment and

plotting routines which provide for selective display of the user's data structures.
The issue of portability of the system is discussed in detail.

[Gibs89] Abstract: An experiment is designed to investigate the relationship between system structure and maintainability. An old, ill-structured system is improved in two sequential stages, yielding three system versions for the study. The primary objectives of the research are to determine how or whether the differences in the systems influence maintenance performance; whether the differences are discernible to programmers; and whether the differences are measurable. Experienced programmers perform a portfolio of maintenance tasks on the systems. Results indicate that system improvements lead to decreased total maintenance time and decreased frequency of ripple effect errors. This suggests that improving old systems may be worthwhile and may yield benefits over the remaining life of the system. System differences are not discernible to programmers; apparently programmers are unable to separate the complexity of the systems from the complexity of the maintenance tasks. This finding suggests a need for further research on the efficacy of subjectively based software metrics. Finally, results indicate that a selected set of automatable, objective complexity metrics reflected both the improvements in the system and programmer maintenance performance. These metrics appear to offer potential as project management tools.

[Gilb79] Abstract: David Gelperin (SEN 4 2) tries to define maintainability in "TRW" terms, using words such as changability and testability which are defined [for the author] in vague and narrow ways. If we are going to engineer the software, then [the author] suggests that our definitions should (1) encompass a broader scope of the concept, (2) have operationally useful measuring methods, and (3) relate more closely to already accepted maintainability concepts in the systems engineering literature.

[GilkXX] Abstract: It is shown that a software design methodology based solely on the identification of abstractions is insufficient for the engineering of complex software systems. Performance analysis is then introduced as an important and necessary tool for choosing between alternatives during design. Methods for carrying out the necessary analysis are discussed. These methods are based on a state model of the computation and a probabilistic grammar based model of the input. Finally a brief description of our continuing research in software design is presented.

[Gill88] Abstract: Embedded real time control systems typically require the use of special debugging environments, which consists of a special *debugging processor* that hosts the debugging software and that monitors the execution of the separate *target system* via a special hardware interface. Our focus is on extending the set of *base debugging features* typically found in such an environment to provide better support for real time task debugging and to provide a more visual graphic display of program behavior. We have developed a system that provides such facilities, in the form of a *task condition specification* and *checking* system and a multi-window graphics display. We have implemented a prototype of these novel debugging features that demonstrates how they work and how they assist in the debugging process. We illustrate the features of our system here by providing a "tour" through an example debugging session. We also comment on various constraints that directed the prototype development process.

[Ginz65] Abstract Procedures for program testing associated with implementation of a large complex real-time system are discussed step by step. The discussion includes testing both in a simulated environment and in real time. Final testing and monitoring of the system performance are also briefly considered.

[Girg85] Abstract: The idea of weak mutation testing is to construct test data which would force program components such an expressions and variable references to produce a wrong 'result' if they were to contain certain types of error, for example, off-by-a-constant or wrong-variable. The idea of data flow driven testing is to construct test data which forces the execution of different interactions between variable definitions and references in a program.
This paper describes a tool for FORTRAN 77 programs which has been developed to help the user apply

the weak mutation and data flow testing techniques. The tool instruments a given source program and collects a program execution history. It is then able to report on the completeness of the test data with respect to weak mutation and a family of data flow path selection criteria. Some preliminary experiments with use of the tool are described.

[Girg86a] Abstract: A system called FORTEST has been developed which helps a user apply weak mutation testing, data flow testing and control flow testing for FORTRAN 77 programs. This paper concentrates on experiments which have been performed to compare the ability of test coverage criteria, monitored by the FORTEST system, to aid discovery of a large number of errors seeded into sample programs. Although overall the control flow strategy was the most effective method in discovering errors, it does not provide such specific guidance in the construction of test data as the other strategies. What is more, some errors were exposed only by the data flow method. Hence it is argued that the diverse strategies are best seen as complementary rather than competing methods.

[Glas79] Abbreviated Preface: Software reliability has been a neglected field. Some emphasis has been placed in recent years on mangement to achieve reliability, and the measurement of reliability, but technology to achieve reliability has progressed little in the same time period.

That situation is changing. Software implementors and purchasers of software, particularly the Department of Defense, are beginning to insist on reliability as a requirement of delivered software. This guidebook is a survey of technological and management techniques, written as a menu. Each item in the menu is evaluated, examples of use are given, and references are provided for further study. Recommendations for achievement of software reliability are also provided.

The guidebook is intended to be useful for all application areas and sizes of software projects; special emphasis is placed on the problems of large projects, such as those of military/space applications and massive interrelated data bases.

The reader is expected to be a software manager or technologist or student who has a basic understanding of what software is, but whose knowledge of reliability concepts is either rudimentary or has not been updated to include recent developments.

[Glas80] Abbreviated Introduction: The literature rarely provides value judgments about which methodologies might prove most effective in identifying and correcting the kinds of errors which practicing professional programmers commonly make. An exception is the work of Howden. Here, a set of known software errors is analyzed against a set of methodologies to determine which methodologies might have detected which errors. Out of this analysis comes a set of "enlightened" advocacies of particular techniques which could have found a large number of the known errors.

This paper reports on an extension of Howden's work. Whereas Howden limited his review to highly mathematical scientific library programs, this review examines real-time software with a considerably more varied set of logical requirements. Whereas Howden limited his review to an intense understanding of small number of errors, this review took a less time-consuming look at a larger number of errors. Whereas Howden considered a small set of reliability methodologies, this review considered a somewhat larger sample of the methodologies defined in [an earlier paper by the author].

The broad conclusions of this paper are similar to those of Howden. A set of reliability methodologies is needed; no one or two techniques are sufficient to come close to guaranteeing software reliability. That set should include some sort of functional testing, some sort of structured testing, and some sort of static analysis. However, the specific methodologies recommended as a result of this study differ somewhat from Howden's recommendations.

[Glas81] Abstract: Persistent software errors - those which are not discovered until late in development, such as when the software becomes operational - are by far the most expensive kind of error. Via analysis of software problem reports, it is discovered the predominant number of persistent errors in large-scale software efforts are errors of omitted logic, that is, the code is not as complex as required by the problem to be solved. Peer design

and code review, desk checking, and ultra-rigorous testing may be the most helpful of the currently available technologies in attacking this problem. New and better methodologies are needed.

[Glig87] Abstract: A new security testing method is proposed that combines the advantages of both traditional "black box" (monolithic functional) testing and "white box" (functional-synthesis-based) testing. The new method allows significant coverage both for security model-based tests and for individual kernel-call tests. It eliminates redundant kernel test cases 1) by using a variant of control synthesis graphs, 2) by analyzing dependencies between descriptive kernel-call specifications, and 3) by exploiting access check separability. A higher degree of test assurance is achieved than that of other security testing methods because the new method helps eliminate cyclic dependencies among test programs for different kernel calls. The application of this method to the testing of the Secure Xenix kernel is illustrated.

[Goel79] Abstract: This paper presents a stochastic model for the software failure phenomenon based on a nonhomogeneous poisson process (NHPP). The failure process is analyzed to develop a suitable mean-value function for the NHPP; expressions are given for several performance measures. Actual software failure data are analyzed and compared with a previous analysis.

[Goel80c] Abstract: In March 1978, Schick and Wolverton published a paper [Schi78] in the IEEE Transactions on Software Engineering. Moranda criticized several aspects of this paper. His critique was reviewed by Littlewood and rebutted by Schick and Wolverton. The purpose of this note is to summarize and comment on the main points raised in these discussions.

[Goel81] Abbreviated Introduction: During the last decade, numerous studies have been undertaken to quantify the failure process of large scale software systems. An important objective of these studies is to predict software performance and use the information for decision making. An important decision of practical concern is the determination of the amount of time that should be spent in testing. This decision of course will depend on the model used for describing the failure phenomenon and the criterion used for determining system readiness.

In this paper we present a cost model based on the time dependent fault detection rate model of Goel and Okumoto and describe a policy that yields the optimal value of test time T.

A brief overview of the failure model is given in Section 2. The cost model and the optimal policies are described in Section 3. The results are illustrated via numerical examples in Section 4.

[Goel83] Abstract: The purpose of this guidebook is to provide state-of-the-art information about the selection and use of existing software reliability models. Towards this objective, we have presented a brief summary of the available models backed by a detailed discussion of most of the models in the appendixes. One of the difficulties in choosing a model is to find a match between the testing environment and a class of models. To help a user in this process, we have presented a detailed discussion of most of the assumptions that characterize the various software reliability models. The process of developing a model has been explained in detail and illustrated via numerical examples.

[Goel85] Abstract: A number of analytical models have been proposed during the past 15 years for assessing the reliability of a software system. In this paper we present an overview of the key modeling approaches, provide a critical analysis of the underlying assumptions, and assess the limitations and applicability of these models during the software development cycle. We also propose a step-by-step procedure for fitting a model and illustrate it via an analysis of failure data from a medium-sized real-time command and control software system.

[Goel88] Abstract: This report presents the results of an experiment investigating the effect of Fortran and Ada languages on program reliability. The experimental design employed was a 2^2 full factorial design, i.e., a design in two variables, each of two levels. The problem used in the experiment was the Launch Interceptor Program (LIP), a simple but realistic anti-missiles system. Reliability comparisons between Ada and Fortran programs were based on the total number of errors as well as on errors found during various testing phases. Some

comparisons were also based on error density, the number of errors per 100 non-comment lines of code. It was found that on the average, the Ada programs had about 70 percent less errors. Similar differences were found for data based on error causes and error types.

[Gogu80] Abbreviated Introduction: This note describes a certain general approach to system design and verification based on the use of a high level executable specification language. We begin with a discussion of the nature of specification languages, and give a rather long list of desirable features. We then discuss the two SRI design and specification projects. SPECIAL with HDM, and the combination of CLEAR, OBJ and CAT in the light of these requirements. We conclude by indicating some possible new directions.

[Good70] Abstract: A definition is given of computer interval arithmetic suitable for implementation on a digital computer. Some computational properties and simplifications are derived. An ALGOL code segment is proved to be a correct implementation of the definition on a specified machine environment.

[Good75a] Abstract: This paper examines the theoretical and practical role of testing in software development. We prove a fundamental theorem showing that properly structured tests are capable of demonstrating the absence of errors in a program. The theorem's proof hinges on our definition of test reliability and validity, but its practical utility hinges on being able to show when a test is actually reliable. We explain what makes tests unreliable (for example, we show by example why testing all program statements, predicates, or paths is not usually sufficient to insure test reliability), and we outline a possible approach to developing reliable tests. We also show how the analysis required to define reliable tests can help in checking a program's design and specifications as well as in preventing and detecting implementation errors.

[Good75c] Abstract: This paper is an initial progress report on the development of an interactive system for verifying that computer programs meet given formal specifications. The system is based on the conventional inductive assertion method: given a program and its specifications, the object is to generate the verification conditions, simplify them, and prove what remains. The important feature of the system is that the human user has the opportunity and obligation to help actively in the simplifying and proving. The user, for example, is the primary source of problem domain facts and properties needed in the proofs. A general description is given of the overall design philosophy, structure, and functional components of the system, and a simple sorting program is used to illustrate both the behavior of major system components and the type of user interaction the system provides.

[Good75d] Abstract: This paper defines exception conditions, discusses the requirements exception handling language features must satisfy, and proposes some new language features for dealing with exceptions in an orderly and reliable way. The proposed language features serve to highlight exception handling issues by showing how deficiencies in current approaches can be remedied.

[Good75e] Abstract: Techniques are presented for the design of computer programs that are proved to meet stated specifications. The design strategy is the simultaneous step-wise refinement of both the program and its proof so that at each step the program constructed so far is proved. At each step, the specifications for a single program unit are given, the unit is designed, and then proved, by automatically supportable methods, before going on to successive steps. The proof i) shows that the program unit meets its specifications, ii) exhibits any assumptions the unit makes about the problem domain, and iii) defines the specifications for units to be designed in later steps. The design process is based on the refinement of operational and data abstractions in both the program and its specifications. These abstractions are what allow the proof at each step to be supported by automatic, or interactive, program proving systems. The abstractions also keep the proofs of the individual units at an appropriate level of abstraction and also largely independent, thus significantly reducing the size of the complete proof of the entire program. These techniques of provable programming are illustrated by two examples.

[Good79a] Abbreviated Introduction: Testing is the principal method of deciding whether a program is *ready for*

operational use. In this paper, [the author] will examine various testing approaches. The purpose behind this examination is

1. to summarize what is known today about testing principles and practices, alerting software developers to shortcomings and advantages of some methods under development;
2. to stimulate productive research on testing methodology by identifying areas where further work is needed; and
3. to provide a framework within which testing techniques can be identified, evaluated, and improved.

[Gord76] Abstract: Recent discoveries in the area of Algorithm Structure or Software Physics have produced a number of hypotheses. One of these relates the number of elementary mental discriminations required to implement an algorithm to measurable properties of that algorithm, and the results of one set of experiments confirming this relationship have been published. That publication, while significant, made no claim to finality, suggesting instead that further experiments were warranted. This paper will present the results of a second set of experiments, having the advantages of being conducted in a single implementation language, Fortran, from problem specifications readily available in computer textbooks.

The first section of this paper presents the timing hypothesis, and the elementary equations upon which it rests. The second section presents the details of the experiment and the results which were obtained, and the third section contains an analysis of the data.

[Gord79a] Abstract: The sharply rising cost incurred during the production of quality software has brought with it the need for the development of new techniques of software measurement. In particular, the ability to objectively assess the clarity of a program is essential in order to rationally develop useful engineering guidelines for efficient software production and language development.

A functional relation between the clarity of a program and the number and frequency of operators and operands which occur in the program is presented. This measure of program clarity provides an estimate of the amount of mental effort required to understand the program, assuming that the reader is fluent in the programming language employed.

This measure is tested by applying it to several published examples which demonstrate improvements in program clarity. The objective assessment which is provided using this measure is found to agree with the experimental data gathered.

[Gord79b] Abstract: Several measures of program clarity have been proposed which attempt to assess the clarity of a program as a function of easily measured properties of the code. Such measures include the number of variables or statements, or the density of go to's.

The measure of program clarity, developed in the field of software science, equates the amount of mental effort required to understand a program with the ratio of program volume to implementation level. To be effective, a measure such as this should reflect the improvement in clarity which occurs when program transformations which make software easier to understand are applied.

The removal of each of six impurity classes from poorly written programs is studied. For a wide class of programs, purification reduces the amount of effort required for comprehension as predicted by the measure.

[Gord85a] Abstract: The purpose of this technical note is to formulate a framework for the evaluation of software metrics and to present preliminary results toward that formulation. This framework is in support of DOD's Software Technology for Adaptable Reliable Systems (STARS) program whose principle software-related goals are:

1. Improve productivity (up to tenfold).
2. Improve quality (maintainability, enhancability, correctness, efficiency) and reliability.
3. Increase portability.
4. Promote development and application of reusable software.
5. Reduce time and cost to develop defense software.

The STARS program will achieve these goals by conducting research and development on integrated

Software Engineering Environments (SEEs) and then utilize them. An integrated SEE will provide automated tools for carrying out the task of various software life cycle phases such as requirements analysis, design, coding, module testing, integration testing, and maintenance.

MITRE's role is that of system research and analysis in determining and developing techniques to measure and evaluate the progress made in achieving the aforementioned goals. The remainder of this technical note presents the initial results of MITRE's review of software metrics and suggestions for future activities.

[Gord86] Abstract: In a distributed environment events occur concurrently on different processors. The order in which events occur cannot be easily determined; a program that works correctly one time may fail subsequently if the timing between processors changes. For this research, we have investigated distributed programming bugs that depend on the relative order between events. We describe a tool (called TAP) to aid the programmer in discovering the causes of timing errors in running programs, describe experiments using TAP, and report the impact TAP's history-keeping mechanism has on the running time of various distributed programs. We also show that TAP is useful in finding other types of distributed program bugs.

[Gord88] Abstract: In a distributed environment, events occur concurrently on different processors. The order in which events occur cannot be easily determined; a program that works correctly one time may fail subsequently if the timing between processors changes. For this research, we have investigated distributed program bugs that depend on the relative order between events. We describe a tool (called TAP) to aid the programmer in discovering the causes of timing errors in running programs. TAP, a tool similar to a postmortem debugger, uses the history of interprocess communication to construct a timing graph, a directed graph where an edge joins node x to node y if event x directly precedes event y in time. The programmer can then use TAP to look at the graph to find the events that occurred in an unacceptable order.

Because of the nondeterministic nature of distributed programs, we feel a history-keeping mechanism must always be active so that bugs can be dealt with as they occur. Our goal is to collect enough information at run time to construct the timing graph if needed. Since it is always active, this mechanism must be efficient.

We also describe experiments run using TAP and report the impact that TAP's history-keeping mechanism has on the running time of various distributed programs.

[Gorl87] Abstract: Mockingbird is a testing methodology founded on a formal specification of the test space. The specifications are executable and bidirectional. When run in one direction they act as generators, producing tests whose properties conform to the specification. When run in the opposite direction they act as acceptors, validating tests against the specification. The specification language is a combination of context-free grammars and constraint systems. The semantics of the specification are based on Constraint Logic Programming. This paper describes the philosophy, design and implementation of Mockingbird and its use in testing a large, complex system.

[Gors80] Abbreviated Abstract: In this investigation of the factors that contribute to program simplicity and understandability (and thus modifiability), we are considering a program to be structured if the control structure can be expressed via a Nassi-Shneiderman Diagram which is equivalent to being a Structured Program in the scheme of Linger, Mills and Witt. We assert that it is insufficient to merely identify these factors. We further assert that it also is insufficient to present subjective measures of these factors in that subjective measurements reduce to opinions and are not measurements at all. If we are satisfied with subjective measurements of complexity, then we are satisfied with opinions of complexity and are also complacently satisfied with programming as an "art." A science of programming necessarily implies the ability to objectively measure on a quantitative scale the important characteristics of a program or of a system of programs.

A system of programs considered at the source statement level possesses a structure directly derived from the top-down analysis and module definition. Although this macro-structure--the inter-module structure--may be relatively simple and hierarchical, it may be complex. What does the word "complex" mean? Can we objectively measure the simplicity/complexity factor in a quantitative sense at the inter-module level?

[Goul74] Abstract: This experiment represents a new approach to the study of the psychology of programming, and demonstrates the feasibility of studying an isolated part of the programming process in the laboratory. Thirty experienced FORTRAN programmers debugged 12 one-page FORTRAN listings, each of which was syntactically correct but contained one non-syntactic error (bug). Three classes of bugs (Array bugs, Iteration bugs, and bugs in Assignment Statements) in each of four different programs were debugged. The programmers were divided into five groups, based upon the information or debugging "aids," given them. Key results were that debug times were short (median 6 min.). The aids groups did not debug faster than the control group; programmers adopted their debugging strategies based upon the information available to them. The results suggested that programmers often identify the intended state of a program before they find the bug. Assignment bugs were more difficult to find than Array and Iteration bugs, probably because the latter could be detected from a high-level understanding of the programming language itself. Debugging was at least twice as efficient the second time programmers debugged a program (though with a different bug in it). A simple hierarchical description of debugging was suggested, and some possible "principles" of debugging were identified.

[Gour81] Abbreviated Introduction: It is the purpose of this thesis to describe a new theoretical framework for testing that [provides a more useful criterion for judging testing than whether or not it is capable of verification, provides a way of comparing methods of testing with each other, addresses program reliability, and generalizes previous work on the subject]. Chapter II develops the framework, including definitions that relate verification and testing and the relative powers of methods of testing. Chapter III shows how previous work, both theoretical and applied fits into this framework. Important theoretical works are shown to be special cases of the new framework and the framework is shown to satisfy some previously articulated needs. Many of the common conceptions about the power of various practical testing methods are confirmed, but one, the implication of mutation testing's "competent programmer hypothesis," is shown to be false. One of the conceptions confirmed by the framework is the importance of finding greater use for specifications of programs in the generation of test data. Prior work on this subject is outlined and then Chapter IV presents in detail a new method for generating test data from formal specifications.

[Gour83] Abstract: Testing has long been in need of mathematical underpinnings to explain its value as well as its limitations. This paper develops and applies a mathematical framework that 1) unifies previous work on the subject, 2) provides a mechanism for comparing the power of methods of testing programs based on the degree to which the methods approximate program verification, and 3) provides a reasonable and useful interpretation of the notion that successful tests increase one's confidence in the program's correctness.

Applications of the framework include confirmation of a number of common assumptions about practical testing methods. Among the assumptions confirmed is the need for generating tests from specifications as well as programs. On the other hand, a careful formal analysis of the usual assumptions surrounding mutation analysis shows that the "competent programmer hypothesis" does not suffice to ensure the claimed high reliability of mutation testing. Hardware testing is shown to fit into the framework as well, and a brief consideration of its shows how the practical differences between it and software testing arise.

[Grad87a] Introduction: Many organizations responsible for the evolution of software systems seem to operate constantly in a reactive mode, fighting the flames of the most recent fire. Behind the visible sense of urgency, though, three primary strategic elements appear to control the actions of managers:

- minimizing defects,
- minimizing engineering effort and schedule, and
- maximizing customer satisfaction.

In a broad sense, the ultimate objective of all three approaches is customer satisfaction. This article specifically discusses their relationships to the maintenance of delivered software.

[Grem84] Abstract: Considerable resources are devoted to the maintenance of programs including that required to correct errors not discovered until after the programs are delivered to the user. A number of factors are believed to affect the occurrence of these errors, e.g., the complexity of the programs, the intensity with which

programs are used, and the programming style. Several hundred programs making up a manufacturing support system are analyzed to study the relationships between the number of delivered errors and measures of the programs' size and complexity (particularly as measured by software science metrics), frequency of use, and age. Not surprisingly, program size is found to be the best predictor of repair maintenance requirements. Repair maintenance is more highly correlated with the number of lines of source code in the program than it is to software science metrics, which is surprising in light of previously reported results. Actual error rate is found to be much higher than that which would be predicted from program characteristics.

[Grie76] Abstract: The ideas behind proofs for programs are outlined, and conventional definitions of assignment, etc., are given. The main part of this paper is the idealized development of nontrivial program in a disciplined fashion. The use of Dijkstra's "calculus" for formal development of programs as a guide to structured program development is discussed in relation to the example presented.

[Grie77] Abstract: A parallel program, Dijkstra's on-the-fly garbage collector, is proved correct using a proof method developed by Owicki. The fine degree of interleaving in this program makes it especially difficult to understand, and complicates the proof greatly. Difficulties with proving such parallel programs correct are discussed.

[Grie79] Abbreviated Abstract: The most prevalent approach to proving that a program satisfies a given property has been the invariant-assertion method. Invariant assertions are supplied to express relationships between the different program variables and are attached to specific program points with the understanding that the assertion is to hold every time control passes through the points. Assuming that the assertion attached to the program entrance (input specification) holds, partial correctness is established if we can prove that the assertion attached to the program exit (output specification) holds whenever control reaches the exit. A completely different method, typically the well-founded set method, is applied to prove program termination, i.e., to prove that if the input specification holds, then control will eventually reach the exit. The two proofs establish the total correctness of the program. The intermittent-assertion method, originally introduced by R.M. Burstall, allows one to establish total correctness by a single proof. This method again involves affixing assertions to points in the program with the intention that, at least once, control will pass through the point with the assertion satisfied by the current variable values.

The authors first present and illustrate the intermittent-assertion method by a variety of examples selected to illustrate different aspects of total correctness are markedly simpler than any known conventional counterparts. Then the authors show how proofs by conventional methods may be translated into intermittent-assertion proofs. They effectively show that the translation process is purely mathematical and does not increase the complexity of the proof. Finally, they present two applications of the intermittent-assertion method. The intermittent-assertion method is employed to establish the validity of the transformation of a recursive program into an equivalent iterative one. The second application is concerned with the correctness proof of continuously operating programs.

[Grna80a] Abstract: In the paper a comparison of processing time and reliability performance for the Recovery Blocks scheme and N-Version Programming technique is presented. Derived queuing models can be useful in deciding which of the strategies should be used, depending on system parameters.

[Grov80] Abstract: An implementation of Ada should be based on a machine-independent translator generating code for a Virtual Machine, which can be realized on a variety of machines. This approach, which leads to a high degree of compiler portability, has been very successful in a number of recent language implementation projects and is the approach which has been specified by the U.S. Army and Air Force in their requirements for Ada implementations.

This paper discusses the rationale, requirements and design of such a Virtual Machine for Ada. The discussion concentrates on a number of fundamental areas in which problems arise: basic Virtual Machine structure, including storage structure and addressing; data storage and manipulation; flow of control; subprograms,

blocks and exceptions; and task handling.

[Guin87] Abstract: Program Mutation is a testing methodology that provides quantitative information on the status of software development. Mothra is a testing environment that uses Program Mutation as its underlying methodology. It consists of an integrated set of tools and interfaces that allow the user to interactively test a software system written in Fortran-77 throughout the software development cycle. It is currently being run under UNIX 4.X BSD and Ultrix V1.2.

This document is primarily a users manual for the first time users of Mothra, although it also intends to serve as a reference manual for the more experienced user. The first section gives introductory background and tries to explain the functionality given by the Mothra testing environment. It should only be read by those with little knowledge of mutation testing, and those wishing more detailed information should consult the bibliography. In the second section the different user interfaces to Mothra are explored and examples of software testing are developed. A user wanting questions answered about the specifics of an interface should consult the section relating to that specific interface.

[Gutt77] Abstract: Abstract data types can play a significant role in the development of software that is reliable, efficient, and flexible. This paper presents and discusses the application of an algebraic technique for the specification of abstract data types. Among the examples presented is a top-down development of a symbol table for a block structured language; a discussion of the proof of its correctness is given. The paper also contains a brief discussion of the problems involved in constructing algebraic specifications that are both consistent and complete.

[Gutt78a] Abstract: A data abstraction can be naturally specified using algebraic axioms. The virtue of these axioms is that they permit a representation-independent formal specification of a data type. An example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is twofold. First, is is shown how the use of algebraic axiomatizations can simplity the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of programs at design time, before a conventional implementation is accomplished.

[Gutt78b] Summary: There have been many recent proposals for embedding abstract data types in programming languages. In order to reason about programs using abstract data types, it is desirable to specify their properties at an abstract level, independent of any particular implementation. This paper presents an algebraic technique for such specifications, develops some of the formal properties of the technique, and show that these provide useful guidelines for the construction of adequate specifications.

[Gutt80] Abstract: The formulation and analysis of a design specification is almost always of more utility than the verification of the consistency of a program with its specification. Good specification tools can assist in the process, but have generally not been proposed and evaluated in this light. In this paper we outline a specification language combining algebraic axioms and predicate transformers, present part of a non-trivial example (the specification of a high-level interface to a display), and finally discuss the analysis of this specification.

[Hall80] Abstract: This is a highly non-technical discussion of nine concepts basic to data processing security. The concepts are: DP RESOURCES, THREAT, VULNERABILITY, EXPOSURE, ADVERSE EVENT, LIKELIHOOD, RISK, CONTROL, and RISK MANAGEMENT. A good grasp of these concepts and their inter-relationships is key to understanding this relatively new and still decidely undisciplined discipline.

[Hall86] Abbreviated Abstract: Cloze tests (i.e., fill-in-missing-parts tests) have been a long-standing measure of prose comprehension. They seem to offer software engineers several theoretical and practical advantages over multiple-choice comprehension quizzes, the most common software comprehension measurement tool. Through

human-subject experimentation, evidence was gathered to support the practical advantages of using the cloze procedure for measuring software comprehension. Cloze tests were found to be easy to construct, administer, and score and capable of discriminating between programs of varying comprehensibility. However, discrepancies between multiple-choice comprehension quiz results and some cloze tests results for the same software suggested that certain forms of software cloze tests may not be valid. A model of software cloze tests was developed to identify a software cloze test characteristic that may produce invalid results. The test characteristic was concerned with the relative proportion of "program-dependent" and "program-independent" cloze items within a test. The developed model was shown to be consistent with software cloze test results of another researcher and led to suggestions for improving software cloze testing.

[Halp87] Abstract: Muse is a verification system which extends the collection of tools developed by SRI International for their Hierarchical Development Methodology (HDM). It enhances the SRI system by providing a capability for proving invariants and constraints for the state machine described by a specification written in SPECIAL (the specification language of HDM). In particular, it enables one to use the HDM system to meet the requirements for formal verification in a National Computer Security Center A1 evaluation of a secure operating system. In addition to the tools provided by SRI, Muse has a parser, a facility to handle multiple modules, a formula generator, and a theorem prover. The theorem prover has a number of interesting features designed to facilitate human direction of the proving process. In concept, it is open-ended. We introduce the notion of a theorem prover kernel as a device for ensuring the logical soundness of the prover in the face of continual improvements to its functionality.

[Hals73b] Abstract: A technique for measuring simple structural properties of algorithms is described. Using these measures, it is found that for a nontrivial class of algorithms there is a quantitative relationship between operators and operands and their usage. Properities of "Full" and "Reduced" algorithms are then explored, and shown to predict the quantitative relationship observed.

[Hals77a] Abstract: This book contains the first systematic summarization of a branch of experimental and theoretical science dealing with the human preparation of computer programs and other types of written material. Application of the classical methods of the natural sciences demonstrates that even such relatively intangible objects as written abstracts and computer programs are governed by natural laws, both in their preparation and in their ultimate form.

The work underlying each chapter of this monograph is firmly based on the methods and principles of classical experimental science. Even so, the results in this area, or more specifically, the concept that significant quantitative results are attainable in such an area, are sufficiently counterintuitive as to appear almost weird.

Intuition, however, is far from trustworthy, as demonstrated when that ancient scientist dropped the wood and lead balls from the tower of Pisa. As he held the balls over the edge of the tower, surely the much greater pull on the hand holding the lead ball should have convinced him that the experiment was unnecessary; that no two bodies would behave the same. Even today, watching a feather and a lead shot fall through a vacuum is fascinating, because it is still "unexpected" or counterintuitive.

Perhaps it is this same sense of the unexpected that has fascinated those of use who are working in this new area now called software science. The first experimental results were obtained nearly five years ago; since that time the methods have been refined and extended in many unanticipated directions, but in each case further investigation has increased rather than limited confidence in the results.

[Hame82] Abstract: Karl Popper has described the scientific method as "the method of bold conjectures and ingenious and severe attempts to refute them." Software Science has made bold conjectures in postulating specific relationships between various 'metrics' of software code and in ascribing psychological interpretations to some of these metrics.

[Haml77a] Abstract: If finite input-output specifications are added to the syntax of programs, these specifications can be verified at compile time. Programs which carry adequate tests with them in this way should be

resistant to maintenance errors. If the specifications are independent of program details they are easy to give, and unlikely to contain errors in common with the program. Furthermore, certain finite specifications are maximal in that they exercise the control and expression structure of a program as well as any tests can.

A testing system based on a compiler is described, in which compiled code is utilized under interactive control, but "semantic" errors are reported in the style of conventional syntax errors. The implementation is entirely in the high-level language on which the system is based, using some novel ideas for improving documentation without sacrificing efficiency.

[Haml77b] Abstract: The techniques of compiler optimisation can be applied to aid a programmer in writing a program which cannot be improved by these techniques. A finite, representative set of test data can be useful in this process. This paper presents the theoretical basis for the (nonconstructive) existence of test sets which serve as maximally effective standins for a unlimited number of input possibilities. It is argued that although the time required by a compiler to fully exercise a program on a set of data may be large, The corresponding improvement in the reliability of the program may also be large if the set meets the given theoretical requirements.

[Haml78a] Abstract: A theory of program testing is presented, based on the idea of "reliable test set." Intuitively, a test is reliable if it exposes all errors that any test could find. To obtain a practical theory, two alterations of this idea are suggested: (1) strengthen the form of specification in the test, (2) restrict the kind of errors that the test must expose. Both of these changes have a natural application to program maintenance.

[Haml78c] Abstract: This paper investigates the application of the execution time theory of software reliability to operational computation center software. A brief review of software reliability concepts is provided. Studies of individual operating system components are discussed, as well as a functional subsystem. This work is based on data taken at a large operating computation center over a period of 15 months.

[Haml86] Abstract: Program testing for confidence requires a probabilistic method, because it is impossible for finite tests to guarantee correctness except under very unrealistic restrictions. Existing sampling theory has not been successfully applied to software because of two peculiar problems: (1) an "operational distribution" of input data is seldom an appropriate description of program use, and (2) sample independence has a difficult meaning for programs. Both problems arise because faults reside in the textual space of a program, but tests probe this space only through the input domain.

A theory is presented in which tests establish a probability of correctness, as opposed to predicting future behavior from past samples. The success of the theory depends on dividing the textual and input spaces into units for which uniform sampling is appropriate. Preliminary work shows that far more test points are needed to gain confidence in a program than predicted by the usual sampling theory.

[Haml87] Abstract: A theory of 'probable correctness' is proposed to assess the reliability of software through testing. Current research in testing is not adequate for this assessment. Most testing methods are intended for debugging, to find failures and connect them to program faults for repair. When these methods no longer expose errors, no analysis has been done to find the confidence that may be placed in the software. (Preliminary results here are that this confidence should be low.) Other work applies conventional decision theory to inputs as samples of a program's use. The application is suspect because the necessary independence and distribution assumptions may be violated; in any case, the results are intuitively incorrect. The proposed theory relies on a uniform distribution of test samples, but relates these to textually occurring faults. Preliminary results include an analysis of partition testing, and suggestions for textual sampling. It is crucial that any such confidence theory be plausible, so the foundations of program sampling are examined in detail.

[Haml88] Abstract: Partition testing, in which a program's input domain is divided according to some rule and tests conducted within the subdomains, enjoys a good reputation. However, comparison between testing that observes partition boundaries and random sampling that ignores the partitions gives the counterintuitive result that partitions are of little value. In this paper we improve the negative results published about partition testing,

and try to reconcile them with its intuitive value. Partition testing is shown to be more valuable than random testing only when the partitions are narrowly based on expected faults and there is a good chance of failure. For gaining confidence from successful tests, partition testing as usually practiced has little value.

[Hane72] Abbreviated Introduction: The largest challenge facing software engineers today is to find ways to deliver large systems on schedule. How can we peer into the hazy contingency portion of a schedule and predict in greater detail where bugs will occur, who will be needed to fix them, elapsed time between internal releases, etc.? Belady and Lehman suggest the need for a "micro-model" for system activities, i.e., a model based on internal, structural aspects of a system. This is essentially the objective of this paper. In the following sections, we will develop a very simple, but useful, technique for modeling the "stabilization" of a large system as a function of its internal structure.

The concrete result described in this paper is a simple matrix formula which serves as a useful *model* for the "rippling" effect of changes in a system. The real emphasis is on the use of the formula as a model; i.e., as an aid to understanding. The formula can certainly be used to obtain numeric estimates for specific systems, but its greater value is that it helps to *explain*, in terms of system structure and complexity, why the process of changing a system is generally more involved than our intuition lead us to believe.

[Hanf70] Introduction: The "syntax machine" discussed here automatically generates random test cases for any suitably defined programming language. The test cases it produces are syntactically valid programs. But they are not "meaningful," and if an attempt is made to execute them, the results are unpredictable and uncheckable. For this reason, they are less valuable than handwritten test cases. However, as an inexhaustible source of new test material, the syntax machine has shown itself to be a valuable tool.

In the following sections, we characterize the use of this tool in testing different types of language processors, introduce the concept of "dynamic grammar" of a programming language, outline the structure of the system, and show what the syntax machine does by means of some examples.

[Hans73] Summary: A central problem in program design is to structure a large program such that it can be tested systematically by the simplest possible techniques. This paper describes the method used to test the RC 4000 multiprogramming system. During testing, the system records all transitions of processes and messages between various queues. The test mechanism consists of fifty machine instructions centralized in two procedures. By using this mechanism in a series of carefully selected test cases, the system was made virtually error free within a few weeks. The test procedure is illustrated by examples.

[Hans78] Abbreviated Introduction: In a recent paper, McCabe introduced the cyclomatic number of a program's flow graph as a measure of its complexity. Myers proposed an improved measure consisting of an interval with the original measure as its upperbound. [The author] will argue that if two values are to be presented as a measure it is preferable to couple a variation of the cyclomatic number with a measure of the program's expression complexity.

[Hans84] Abbreviated Preface: Software designers are dissatisfied with the present status of quality assurance and control. Methods and tools are being developed that attempt to locate errors in systems or to demonstrate the absence of such errors. Although many of these tools are still experimental and difficult to use, they have been used successfully in a number of applications. Although additional research on validation is necessary, it is likely that developers of systems could make good use of some of these tools provided they are robust and stable.

From the point of assuring quality throughout the entire life cycle, most of the existing methods and tools are only suitable for specific phases and error classes. By suitable combinations of methods and tools, quality assurance and control should become more effective. The required combination depends on the specific quality requirements, on the current situation of a project, and last but not least, on the available resources.

The goals of this [book are] to review the current status of software validation technology, to provide an in-depth look at the issues, and to project future developments, all in the light of the overall aim of achieving an integrated framework for software validation.

[Hant76] Abstract: This paper explains, in an introductory fashion, the method of specifying the correct behavior of a program by use of input/output assertions and describes one method of showing that the program is correct with respect to those assertions. An initial assertion characterizes conditions expected to be true upon entry to the program and a final assertion characterizes conditions expected to be true upon exit from the program. When a program contains no branches, a technique known as symbolic executions can be used to show that the truth of the initial assertion upon entry guarantees the truth of the final assertion upon exit. More generally, for a program with branches one can define a symbolic execution tree. If there is an upper bound on the number of times each loop in such programs can be executed, a proof of correctness can be given by a simple traversal of the (infinite) symbolic execution tree.

However, for most programs, no fixed bound of the number of times each loop is executed exists and the corresponding symbolic execution trees are infinite. In order to prove the correctness of such programs, a more general assertion structure must be provided. The symbolic execution tree of such programs must be traversed inductively rather than explicitly. This leads naturally to the use of additional assertions which are called "inductive assertions."

[Harr81a] Abbreviated Introduction: The calculation of the cyclomatic number proves to be an effective complexity measure. However, because the cyclomatic measure only counts the number of basic paths, it is incapable of recognizing the effects of two major complexity factors which can be intuitively seen to increase program complexity. These two items are the complexity of the individual blocks within the program - which we shall refer to as "program magnitude," and the program.

[Harr81b] Conclusion: Two programs may be of the same length and possess equivalent properties in all respects except for the control structure configuration. We have illustrated the variations in complexity which may arise from such situations by using two topological measures, viz., McCabe's Cyclomatic Complexity Number and the Scope Complexity Ratio. The Scope Measure is able to distinguish among programs which the cyclomatic number measure considers to be equally complex.

[Harr82] Abbreviated Introduction: Over the past several years, computer scientists have devoted a great deal of effort to measuring computer program "complexity," since many large software systems can be used for 10, 15, or even 20 years. A large part of that time involves maintenance activities, which include all changes made to a piece of software after it has been delivered to and accepted by the final user. Consequently, maintenance is most affected by program complexity.

Recent estimates suggest that about 40 to 70 percent of annual software expenditures involve maintenance of existing systems. Clearly, if complexities could somehow be identified, then programmers could adjust maintenance procedures accordingly. What is needed is some method of pinpointing the characteristics of a computer program that are difficult to maintain and measuring the degree of their presence (or lack of it). Various approaches may be taken in measuring complexity characteristics, such as Baird and Noma's approach, in which scales of measurement are divided into the following four types:

1. Nominal scales.
2. Ordinal scales.
3. Interval scales.
4. Ratio scales.

We believe the ordinal scale to be the best choice for examining complexity metrics, and all measures discussed in this article are in that framework.

[Harr85] Summary: We have developed a Reduced Form which allows software complexity data to be shared among researchers, and at the same time prevents the reconstruction of the actual source code, thus preserving the confidentiality of the software. This is a major concern of many organizations considering participating in metric research.

Some current metrics can be obtained from the Reduced Form. Additional work must be done to include information needed for other metrics before this tool can be finalized. It is hoped that this paper will spark an

interest in other complexity metric researchers who can contribute to the development of the Reduced Form. Those interested should see the companion paper in this issue.

[Harr88b] Abbreviated Introduction: Over the past decade, numerous attempts have been made to develop software complexity measurements. Formulating such measures would allow us to compare two programs and see which was more complex. This article describes one very popular approach to measuring software complexity: software science.

[Harr88c] Abstract: There have been several efforts to use symbolic execution to test and analyze concurrent programs. Recently proof systems have also emerged for concurrent programs and for the Ada language in particular. This paper focuses on using symbolic properties of Ada programs. It expands upon past efforts by incorporating tasking proof rules into the symbolic executor allowing Ada programs with tasking to be formally verified.

[Hart71] Abstract: The purpose of this paper is to outline the theory of computational complexity which has emerged as a comprehensive theory during the last decade. This theory is concerned with the quantitative aspects of computations and its central theme is the measuring of the difficulty of computing functions. The paper concentrates on the study of computational complexity measures defined for all computable functions and makes no attempt to survey the whole field exhaustively nor to present the material in historical order. Rather it presents the basic concepts, results, and techniques of computational complexity from a new point of view from which the ideas are more easily understood and fit together as a coherent whole.

[Hart79] Abstract: The Advanced Interactive Debugging System (AIDS) is described. It is a powerful high-level symbolic interactive debugging aid. AIDS is intended to be available in a program's environment without requiring debugging statements in the program's source code or inclusion of AIDS in the program's executable module.

[Hass80] Abstract: White and Cohen have proposed the domain testing method, which attempts to uncover errors in a path domain by selecting test data on and near the boundary of the path domain. The goal of domain testing is to demonstrate that the boundary is correct within an acceptable error bound. Domain testing is intuitively appealing in that it provides a method for satisfying the often suggested guideline that boundary conditions should be tested.

In addition to proposing the domain testing method, White and Cohen have developed a test data selection strategy, which attempts to satisfy this method. Further, they have described two error measures for evaluating domain testing strategies. This paper takes a close look at their strategy and their proposed error measures. It is shown that inordinately large domain errors may remain undetected by the White and Cohen strategy. Two alternative domain testing strategies, which improve on the error bound, are then proposed and the complexity of each of the three strategies is analyzed. Finally, several other issues that must be addressed by domain testing are presented and the general applicability of this method is discussed.

[Hech75] Abstract: A simple, iterative bit propagation algorithm for solving global data flow analysis problems such as "available expressions" and "live variables" is presented and shown to be quite comparable in speed to the corresponding interval analysis algorithm. This comparison is facilitated by a result relating two parameters of a reducible flow graph (rfg). Namely, if G is an rfg, d is the largest number of back edges found in any cycle-free path in G, and k is the length of the interval derived sequence of G, then $k>=d$. (Intuitively, k is the maximum nesting depth of loops in a computer program, while d is a measure of the maximum loop-interconnectedness.) The node ordering employed by the simple algorithm is the reverse of the order in which a node is last visited while growing any depth-first spanning tree of the flow graph. In addition, a dominator algorithm for an rfg is presented which takes O(edges) bit vector steps.

[Hech77a] Abbreviated Preface: This book presents a theoretical foundation for the pre-execution analysis of

computer programs that is usually referred to as control flow analysis and data flow analysis. Flow analysis is a fundamental prerequisite for many important types of code improvement. In general, control flow analysis precedes data flow analysis. Control flow analysis is the encoding of pertinent, possible program control flow structure or flow of control, usually in the form of one or more graphs. Data flow analysis is the process of ascertaining and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or various attributes of variables) in a computer program.

The primary goal of this book is to teach people algorithms to incorporate in code improvers. However, these algorithms do not perform various code improvements per se, but instead gather information prerequisite to many code improvements. Thus, the subject of this book is not code improvement, but only one constituent process used in many code improvers. The reader will be introduced to typical problems requiring flow analysis algorithms and the theoretical foundation for these algorithms.

[Hech79] Abstract: Limitations in the current capabilities for verifying programs by formal proof or by exhaustive testing have led to the investigation of fault-tolerance techniques for applications where the consequence of failure is particularly severe. Two current approaches, N-version programming and the recovery block, are described. A critical feature in the latter is the acceptance test, and a number of useful techniques for constructing these are presented. A system reliability model for the recovery block is introduced, and conclusions derived from this model that affect the design of fault-tolerant software are discussed.

[Hech80] Abstract: Many new uses of computers require extremely high reliability of the computing function as a whole, and the software involved must conform to these requirements. In more conventional applications, the attainment of higher software reliability will be of great economic benefit. A study of one scientific computing center, in which a good deal of effort had been devoted to providing a highly dependable facility, showed 386 service interruptions in one year. Of these, 227 (almost 60%) were due to software problems. In software generated for the military services, quality assurance provisions are now being invoked which are motivated by the need for higher reliability as well as for greater ease of maintenance.

[Hell72] Abstract: The computational work of a process is measured in terms of the information in a memory for its table-lookup implementation. This measure is applied first to simple logical and arithmetic processes, and then more complicated processes comprising organizations (called synergisms) of several subprocesses. The computational advantages of Cartesian, compositional, and sequential synergisms are investigated and illustrated by means of the work measure. The relation between the work of a process and the work capacity of a facility on which it is implemented is examined, and a concept of efficiency of implementations is formulated. A few areas for further investigation are outlined.

[Hell87] Abstract: This document presents procedures to be followed by flight dynamics software development projects that are monitored by the Software Engineering Laboratory (SEL) for collecting data in support of software engineering research activities. An overview of data collection during the life cycle of a development project is presented. This overview is followed by a discussion of the manner in which the SEL measures the structure and growth of the software product. Finally, detailed instructions for the completion and sub-mission of SEL data collection forms are presented.

[Helm83] Abstract: A runtime monitoring system for detecting and describing tasking errors in Ada programs is presented.

Basic concepts for classifying tasking errors, called deadness errors, are defined. These concepts indicate which aspects of an Ada computation must be monitored in order to detect deadness errors resulting from attempts to rendezvous or terminate. They also provide a basis for the definition and proof of correct detection. Descriptions of deadness errors are given in terms of the basic concepts.

The monitoring systems has two parts: (1) a separately compiled runtime monitor that is added to any Ada source text to be monitored, and (2) a pre-processor that transforms the Ada source text so that necessary descriptive data is communicated to the monitor at runtime. Some basic preprocessing transformations and an

abstract monitoring for a limited class of errors were previously presented. Here an Ada implementation of a monitor and a more extensive set of pre-processing transformations are described. This system provides an experimental automated tool for detecting deadness errors in Ada83 tasking and supplies useful diagnostics. The use of the runtime monitor for debugging and programming evasive actions to avoid imminent errors is described and examples of experiments are given.

[Helm84a] Abstract: A new class of errors, not found in sequential languages, can result when the tasking constructs of Ada are used. These errors are called deadness errors and arise when task communication fails. Since deadness errors often occur intermittently, they are particularly hard to detect and diagnose. Previous papers describe the theory and implementation of runtime monitors to detect deadness errors in tasking programs. The problems of detection and description of errors are different. Even when a dead state is detected, giving adequate diagnostics that enable the programmer to locate its cause in the Ada text is difficult. This paper discusses the use of simple diagnostic descriptions based on Ada tasking concepts. These diagnostics are implemented in an experimental runtime monitor. Similar facilities could be implemented in task debuggers in forthcoming Ada support environments. Their usefulness and shortcomings are illustrated in an example experiment with the runtime monitor. Possible future directions in task error monitoring and diagnosis based on formal specifications are discussed.

[Helm85] Abstract: TSL is a language for specifying sequences of tasking events in Ada programs. TSL specifications are submitted with an Ada program and are monitored at runtime for consistency with the actual tasking events as they occur. This paper presents a preliminary design for TSL, an informal overview of its capabilities, and an operational semantics.

[Hend75] Abstract: A technique is presented whereby a significant amount of program validation can be done simply by exercising the program components in a model environment provided by a finite state machine, specially built to characterise the real environment of that component. The tools necessary to support such a technique are characterised and the merits and demerits of the technique are discussed.

[Henn76b] Abbreviated Introduction: In this note we introduce the concept of a Linear Code Sequence and Jump (LCSAJ) triple. With the aid of these LCSAJs it is possible to analyze both the static and dynamic characteristics of computer programs. In particular they have been extensively used in the analysis of numerical algorithms in both FORTRAN IV and ALGOL 68.

[Henn78] Abstract: This paper describes an experimental testbed facility designed to examine some of the problems which arise in the implementation of high quality numerical software libraries.

The testbed is used to measure the effectiveness of test programs. Effectiveness here is used in the sense that these test programs should ensure that the routine implementation is error free rather than to examine the numerical properties of the algorithm.

The testbed has been used in extensive investigations of the stringent test programs of the NAG numerical algorithms library and continuation of this work is seen as a major application for the testbed.

[Henn84] Abstract: The roles and capabilities of LDRA software Testbeds and their appropriate environments have been described in a number of papers. The way in which management uses the tools as elements of a controlled software development environment is described. The principal benefits of such use are that management has the assurance that software development standards are enforced, and has reliable information concerning project status. The explicit standards enforced by use of these tools are described in detail [elsewhere]. One class of these standards is that of test effectiveness, which is measured primarily through three test effectiveness metrics reinforced by a code auditing capability.

This paper attempts to quantify the benefits of using such a software Testbed in providing assurance of the absence of program errors. The attempt is made from two viewpoints, the theoretical and the experimental.

The theoretical aspect is important because the practical use of a tool may fail to demonstrate that the tool

can be a powerful detector of a class of errors simply because no errors of that type were present in the software sample validated.

Finally the paper attempts to summarize some of the experiences gained through the use of the tools over a twelve year period.

[HennXX] Abbreviated Introduction: The principal objective in functional testing is to verify that a software system satisfies the requirements. This is achieved by constructing test data which in some way explores each of the possibly many functions which the system is required to perform.

When the requirements are expressed in terms of functions, then it is possible to expand the detail until the requirements are expressed in terms of Basic User Perceived functions. Termination criteria for this process are difficult to specify–the point is that there should be no further fine structures as perceived by the users. One task of functional testing is to take each of these Basic User Perceived functions and supply test data to exercise them. In general, it is not sensible or even possible to test them individually, so they must be tested in various combinations.

There is now a considerable body of functional tests which have been investigated by using structural testing metrics. That is, after the functional tests have been completed, the software and test data are examined by structural testing techniques to obtained the coverage metrics. The overall results of traditional functional tests are not impressive in terms of exercising the software. However, it has not been clear what further improvements should be made. It is the objective of this chapter to improve this position.

[Henr81a] Abstract: Automatable metrics of software quality appear to have numerous advantages in the design, construction and maintenance of software systems. While numerous such metrics have been defined, and several of them have been validated on actual systems, significant work remains to be done to establish the relationships among these metrics. This paper reports the results of correlation studies made among three complexity metrics which were applied to the same software system. The three complexity metrics used were Halstead's effort, McCabe's cyclomatic complexity and Henry and Kafura's information flow complexity. The common software system was the UNIX operating system. The primary result of this study is that Halstead's and McCabe's metrics are highly correlated while the information flow metric appears to be an independent measure of complexity.

[Henr81b] Abstract: Structured design methodologies provide a disciplined and organized guide to the construction of software systems. However, while the methodology structures and documents the points at which design decisions are made, it does not provide a specific, quantitative basis for making these decisions. Typically, the designers' only guidelines are qualitative, perhaps even vague, principles such as "functionality," "data transparency," or "clarity." This paper, like several recent publications, defines and validates a set of software metrics which are appropriate for evaluating the structure of large-scale systems. These metrics are based on the measurement of information flow between system components. Specific metrics are defined for procedure complexity, module complexity, and module coupling. The validation, using the source code for the UNIX operating system, shows that the complexity measures are strongly correlated with the occurrence of changes. Further, the metrics for procedures and modules can be interpreted to reveal various types of structural flaws in the design and implementation.

[Henr85] Abstract: This paper describes the development of a procedure for evaluating software engineering methodologies. In formulating this evaluation procedure, the first question addressed is-What constitutes a *methodology*? Using this discussion as a basis, we then establish a linkage of objectives, principles, and attributes that are intrinsic to an "ideal" methodology and which reflects an assessment structured by the needs, process, and product sequence for system development. This linkage is based on universally accepted software engineering goals, and provides a *comparative* scale for assessing the relative "goodness" of a given methodology. The final section of this paper discusses the application of this procedural evaluation approach to the Software Cost Reduction (SCR) methodology currently used by the United States Navy. This example reveals the inherent power of the procedural approach in evaluating software development methodologies.

[Henr88a] Abstract: Maintenance of software makes up a large fraction of the time and money spent in the software life cycle. By reducing the need for maintenance these costs can also be reduced. Predicting where maintenance is likely to occur can help to reduce maintenance by prevention. This paper details a study of the use of software quality metrics to determine high complexity components in a software system. By the use of a history of maintenance done on a particular system, it is shown that a predictor equation can also be developed to identify components which needed maintenance activities. This same equation can be used to determine which components are likely to need maintenance in the future. Through the use of these predictions and software metric complexities it should be possible to reduce the complexity of that component through further decomposition. Even though this is only one study, this methodology of developing maintenance predictors could be applied in any environment.

[Henr88b] Abstract: In this paper we describe our initial work on a long-term project to develop and validate a reliability model and a new class of software complexity metrics which are related to this model. In contrast to previous "black box" approaches, the reliability model is novel because it incorporates knowledge about the system in the form of quantitative software complexity metrics. While the initial model uses existing software metrics a parallel effort in this project is investigating new classes of metrics, **interface** and **dynamic** metrics, which are useful in their own right but are also of particular relevance to the reliability model. The initial definition of both the model and the metrics are given along with a description of the next research milestones.

[HenrXX] Abstract: For many years the software engineering community has been attacking the software reliability problem on two fronts. First via design methodologies, languages and tools as a precheck on quality and second by measuring the quality of produced software as a postcheck. This research attempts to unify the approach to creating reliable software by providing the ability to measure the quality of a design prior to its implementation. A comparison of a graphical and a textual design language is presented in an effort to support research findings that the human brain works more effectively in images than in text.

[Hetz84] Abbreviated Preface: The quality of systems developed and maintained in most organizations is poorly understood and below standard. This book explains how software can be tested effectively and how to manage that effort within a project or organization. It demonstrates that good testing practices are the key to controlling and improving software quality and explains how to develop and implement a balanced testing program to achieve significant quality improvements.

The book covers the discipline of software testing: what testing means, how to define it, how to measure it, and how to ensure its effectiveness. The term *software testing* is used broadly to include the full scope of what is sometimes referred to as test and evaluation or verification and validation activities. Software testing is viewed as the continuous task of planning, designing, and constructing tests, and of using those tests to assess and evaluate the quality of work performed at each step of the system development process. Both the **why** and *how* are considered. The *why* addresses the underlying principles, where the concepts came from, and why it is important. The **how** is practical and explains the method and management practices so that they may be easily understood and put into use.

[Hibb82] Abstract: This paper reports on the status of a research project to develop compiler techniques to optimize programs for execution on an asynchronous multiprocessor. We adopt a simplified model of a multiprocessor, consisting of several identical processors, all sharing access to a common memory. Synchronization must be done explicitly, using two special operations that take a period of time comparable to the cost of data operations. Our treatment differs from other attempts to generate code for such machines because we treat the necessary synchronization overhead as an integral part of the cost of a parallel code sequence. We are particularly interested in heuristics that can be used to generate good code sequences, and local optimizations that can then be applied to improve them. Our current efforts are concentrated on generating straight-line code for high-level, algebraic languages.

We compare the code generated by two heuristics, and observe how local optimization schemes can gradually improve its quality. We are implementing our techniques in an experimental compiler that will generate code

for Cm*, a real multiprocessor, having several characteristics of our model computer.

[Hill83] Abstract: This note describes RED, a remotely executed debugger capable of generating a real-time source level trace history of a high level language program executing on a microprocessor. The trace history consists of a display of the source statements of each basic block executed, annotated by the time at which execution of that block began. Basic blocks are traced rather than statements to reduce sampling bandwidth requirements while still retaining the ability to record the essential logical flow of programs. RED is intended to assist in debugging stand-alone high level language process control programs with real-time constraints.

We outline two possible implementation schemes for generating the real-time trace history. In both, a "debugging co-processor" collects in a history buffer the values of the program counter (PC) and the corresponding value of a clock as each basic block begins execution. The debugger, which runs on the processor hosting the compiler and has access to the co-processor over a fast link, reconstructs a source level trace from the PC-time pairs in the history buffer. In one scheme, the language compiler emits an extra instruction at the beginning of each basic block in the program to output the value of the program counter to a parallel port connected to the debug processor. The second method makes use of an extended target memory space to provide tag bits denoting basic blocks. When an instruction is fetched, the debug processor detects the presence of the tag bits and buffers up the value of the corresponding program counter and time. The first method is simpler to implement, requiring only conventional, usually straightforward hardware additions to the target, but requires the execution overhead of the extra instructions. In both cases the debugger itself runs on the host processor and has access to tables generated during compile time of the source program.

[Hite88] Abstract: Testing programs with tractable algorithms is one area in which software engineers have made numerous advances over the past few decades. Testing rule-based expert systems, however, is a new area in software engineering which requires new techniques.

For the most part, traditional software engineering testing strategies assume modular program development. This assumption is impractical to make for expert system development, for the knowledge base of an expert system is quite simply a huge non-modular program. It consists almost entirely of non-ordered, multi-branching decision statements. In traditional programming, the module interfaces are limited and well defined. For rule-based expert systems, the interaction among rules is combinatoric and highly data-driven. Thus the testing of a completed expert system via traditional path analysis is impractical.

The design of a testing strategy for expert systems focuses on the generic phases of expert system development. Briefly, these phases include system definition, incremental system implementation, and system maintenance. Using the simplified breakdown of the expert system development process as a guide, certain testing techniques can be generalized enough to work for any expert system application.

[Hoar69] Abstract: In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from pursuance of these topics.

[Hoar71b] Abstract: A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A systematic technique is described for constructing the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

[Hoar72] Introduction: In the development of programs by stepwise refinement, the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some

convenient and efficient concrete representation in the store of a computer; and finally programs the primitive operations required by his abstract program in terms of this concrete representation. This paper suggests an automatic method of accomplishing the transition between an abstract and a concrete program, and also a method of proving its correctness; that is, of proving that the concrete representation exhibits all the properties expected of it by the "abstract" program. A similar suggestion [has" been] made more formally in algebraic terms, which gives a general definition of simulation. However, a more restricted definition may prove to be more useful in practical program proofs.

If the data representation is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program. Since abstract programs are usually very much shorter and easier to prove correct, the total task of proof has been considerably lightened by factorising it in this way. Furthermore, the two parts of the proof correspond to the successive stages in program development, thereby contributing to a constructive approach to the correctness of programs. Finally, it must be recalled that in the case of larger and more complex programs the description given above in terms of two stages readily generalizes to multiple stages.

[Hoar74] Abstract: This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.

[Hoar75] Abbreviated Abstract: This paper distinguishes a number of ways of using parallelism, including disjoint processes, competition, cooperation, and communication. In each case an axiomatic proof rule is given.

[Hoar78] Abstract: This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

[Hoar81] Abstract: A process communicates with its environment and with other processes by synchronized output and input on named channels. The current state of a process is defined by the sequences of messages which have passed along each of the channels, and by the sets of messages that may next be passed on each channel. A process satisfies an assertion if the assertion is at all times true of all possible states of the process. The author presents a calculus for proving that a process satisfies the assertion describing its intended behaviour. The following constructs are axiomatised: output; input; simple recursion; disjoint parallelism; channel renaming, connection and hiding; process chaining; nondeterminism; conditional; alternation; and mutual recursion. The calculus is illustrated by proof of a number of simple buffering protocols.

[Hoar87] Introduction: The code of a computer program is a formal text, describing precisely the actions of a computer executing that program. As in other branches of engineering, the progress of its implementation as well as its eventual quality can be promoted by additional design documents, formalized before starting to write the final code. These preliminary documents may be expressed in a variety of notations suitable for different purposes at different stages of a project, from capture of requirements through design and implementation, to delivery and long-term maintenance. These notations are derived from mathematics, and include algebra, logic, functions, and procedures. The connection between the notations is provided by mathematical calculation and proof.

This article introduces and illustrates a selection of formal methods by means of a single recurring example, the design of a program to compute the greatest common divisor of two positive numbers. It is hoped that some of the conclusions drawn from analysis of this simple example will apply with even greater force to software engineering projects on a more realistic scale.

[Hodg76] Abbreviated Introduction: The production of consistently executable and dependable software demands a thoughtful systematic implementation – with clear documentation at each production stage. Recognizing this, the Data Systems Laboratory, at Marshall Space Flight Center, NASA, began a research effort to help discover and institute sound engineering principles into a methodology for the production of software.

Achieving this end demanded, among other things, the development of a formal specifications language that could traceably embody requirements, a high level programming language that could be generated easily and faithfully from specifications and could promote a logical error-free code implementation, a language preprocessor to allow compatibility of the methodology with existing compilers and finally, automatic code analysis tools to attain our original objective--reducing software test and verification effort.

Such a methodology, an integrated Software Specification and Evaluation System (SSES), is being developed for NASA/MSFC. [This paper presents] the technical highlights and unification of the system.

[Holz82] Abstract: This paper introduces a simple algebra for the validation of communication protocols in message passing systems. The behavior of each process participating in a communication is first modeled in a finite state machine. The symbol sequences that can be accepted by these machines are then expressed in "protocol expressions," which are defined as regular expressions extended with two new operators: division and multiplication. The interactions of the machines can be analyzed by combining protocol expressions via multiplication and algebraically manipulating the terms.

The method allows for an arbitrary number of processes to participate in an interaction. In many cases an analysis can be performed manually, in other cases the analysis can be automated. The method has been applied to a number of realistic protocols with up to seven interacting processes.

An automated analyzer was written in the language C. The execution time of the automated analysis is in most cases limited to a few minutes of CPU time on a PDP 11/70 computer.

[Hous77] Abstract: A tool for the systematic production of test cases for a compiler is first presented. The input of the generator are formal grammers, derived from the definition of the reference language. This tool has been applied to the generation of test programs for Algol 68. For each construction which the language possesses, the syntactic structure of the corresponding test and the semantic verifications it contains are given. The test set has begun to be employed on a specific implementation. Discovered errors related to Algol 68 constructions are analyzed.

[Howd75a] Abstract: A methodology for generating program test data is described. The methodology is a model of the test data generation process and can be used to characterize the basic problems of test data generation. It is well defined and can be used to build an automatic test data generation system.

The methodology decomposes a program into a finite set of classes of paths in such a way that an intuitively complete set of test cases would cause the execution of one path in each class. The test data generation problem is theoretically unsolvable: there is no algorithm which, given any class of paths, will either generate a test case that causes some path in that class to be followed or determine that no such data exist. The methodology attempts to generate test data for as many of the classes of paths as possible. It operates by constructing descriptions of the input data subsets which cause the classes of paths to be followed. It transforms these descriptions into systems of predicates which it attempts to solve.

[Howd76c] Abstract: A set of test data T for a program P is *reliable* if it reveals that P contains an error whenever P is incorrect. If a set of tests T is reliable and P produces correct output for each element of T then P is a correct program. Test data generation strategies are procedures for generating sets of test data. A testing strategy is reliable for a program P if it produces a reliable set of test data for P. It is proved that an effective testing strategy which is reliable for all programs cannot be constructed. A description of the path analysis testing strategy is presented. In the path analysis strategy data are generated which cause different paths in a program to be executed. A method for analyzing the reliability of path testing is introduced. The method is used to characterize certain classes of programs and program errors for which the path analysis strategy is reliable. Examples of published incorrect programs are included.

[Howd76e] Abstract: Symbolic evaluation techniques can be used to determine the cumulative effects of a program's calculations on the branching predicates and output variables in the program. If the evaluation techniques are carefully and selectively applied, they can be used to generate revealing symbolic representations of the computations carried out by the paths in a program, and of the systems of predicates that describe the input data that causes program paths to be executed. A symbolic evaluation system called DISSECT is described which can be used to analyze FORTRAN programs. The system includes a sophisticated command language that allows the user to selectively apply symbolic evaluation techniques to different program paths and subpaths. The command language allows the user to carry out different levels of symbolic testing of a program and to construct systems of predicates that can be used to automate the generation of numeric test data. Experiments with the system which illustrate its advantages and limitations are included. DISSECT can be used to carry out a systematic, documented reliability analysis of a program. The paper concludes with a discussion of the potential use of systems like DISSECT as the basic software certification tool in the software development process.

[Howd77a] Abstract: The report is divided into two parts. The first part contains a study of the design of symbolic evaluation systems. It also contains an estimate of the costs of using such a system to carry out symbolic program testing. The second part contains a study of the effectiveness of symbolic testing. It contains an analysis of the circumstances under which symbolic testing is reliable for discovering program bugs. The effectiveness of symbolic testing is compared with other reliability analysis techniques. The analysis of the effectiveness of symbolic testing which is contained in Part 2 is based on the study of six programs. Descriptions of the programs and the details on the analyses are continued in the six Appendices.

[Howd77b] Abstract: Symbolic testing and a symbolic evaluation system called DISSECT are described. The principle features of DISSECT are outlined. The results of two classes of experiments in the use of symbolic evaluation are summarized. Several classes of program errors are defined and the reliability of symbolic testing in finding bugs is related to the classes of errors. The relationship of symbolic evaluation systems like DISSECT to classes of program errors and to other kinds of program testing and program analysis tools is also discussed. Desirable improvements in DISSECT, whose importance was revealed by the experiments, are mentioned.

[Howd77c] Summary: The effectiveness in discovering errors of symbolic evaluation and of testing and static program analysis are studied. The three techniques are applied to a diverse collection of programs and the results compared. Symbolic evaluation is used to carry out symbolic testing and to generate symbolic systems of path predicates. The use of the predicates for automated test data selection is analyzed. Several conventional types of program testing strategies are evaluated. The strategies include branch testing, structured testing and testing on input values having special properties. The static source analysis techniques that are studied include anomaly analysis and interface analysis.

Examples are included which describe typical situations in which one technique is reliable but another unreliable. The effectiveness of symbolic testing is compared with testing on actual data and with the use of an integrated methodology that includes both testing and static source analysis. Situations in which symbolic testing is difficult to apply or not effective are discussed. Different ways in which symbolic evaluation can be used for generating test data are described. Those ways for which it is most effective are isolated. The paper concludes with a discussion of the most effective uses to which symbolic evaluation can be put in an integrated system which contains all three of the validation techniques that are studied.

[Howd78a] Abstract: Two approaches to the study of program testing are described. One approach is theoretical and the other empirical. In the theoretical approach situations are characterized in which it is possible to use testing to formally prove the correctness of programs or the correctness of properties of programs. In the empirical approach statistics are collected which record the frequency with which different testing strategies reveal the errors in a collection of programs. A summary of the results of two research projects which investigated these approaches are presented. The difference between the two approaches are discussed and their relative advantages and disadvantages are compared.

[Howd78b] Summary: An approach to the study of program testing is introduced in which program testing is treated as a special kind of equivalence problem. In this approach, classes of programs P^* and associated classes of test sets T^* are defined which have the property that if two programs P and Q in P^* agree on a set of tests from T^*, then P and Q are computationally equivalent. The properties of a class P^* and the associated class T^* can be thought of as defining a set of assumptions about a hypothetical correct version Q of a program P in P^*. If the assumptions are valid then it is possible to prove the correctness of P by testing. The main result of the paper is an equivalence theorem for classes of programs which carry out sequences of computations involving the elements of arrays.

[Howd78c] Abstract: The use of traces proving properties of programs is investigated. Two kinds of traces are studied. The first, called "value traces" contain intermediate values of program variables. A theorem is presented which can be used to verify the computations which generate the values appearing in a value trace. The second kind of trace, called a "symbolic trace" contains the unevaluated sequence of assignment statements and branch predicates that occur along a program path. A special class of symbolic traces called "elementary traces" is defined. A theorem is presented which proves that if the elementary symbolic traces of a program are correct then all of its symbolic traces are correct. The correctness of the set of symbolic traces for a program implies the correctness of the program.

[Howd78d] Abstract: This short paper summarizes the different approaches to a theory of program testing. The goals of a theory of testing and several of the results which have been achieved are described.

[Howd78f] Abstract: In recent years a number of research projects have been completed which have attempted to assess the effectiveness of different software validation methods. The results of those projects are summarized and the effectiveness of the different methods compared.

[Howd80a] Abstract: An approach to functional testing is described in which the design of a program is used to generate functional test data. The approach depends on the use of design methods that model the abstract functional structure of a program as well as the abstract structure of the data on which the program operates. An example of the use of the method is given and a discussion of its effectiveness.

[Howd80b] Abstract: Error analysis involves the examination of a collection of programs whose errors are known. Each error is analyzed and validation techniques which would discover the error are identified. The errors that were present in version five of a package of Fortran scientific subroutines and then later corrected in version six were analyzed. An integrated collection of static and dynamic analysis methods would have discovered the errors in version five before its release. An integrated approach to validation and the effectiveness of individual methods are discussed.

[Howd80c] Abstract: An approach to functional testing is described in which the design of a program is viewed as an integrated collection of functions. The selection of test data depends on the functions used in the design and on the value spaces over which the functions are defined. The basic ideas in the method were developed during the study of a collection of scientific programs containing errors. The method was the most reliable testing technique for discovering the errors. It was found to be significantly more reliable than structural testing. The two techniques are compared and their relative advantages and limitations are discussed.

[Howd80d] Abstract: Program testing metrics are based on criteria for measuring the completeness of a set of program tests. *Branch testing* measures the percentage of program branches that are traversed during a set of tests. *Mutation testing* measures the ability of a set of tests to distinguish a program from similar programs. A criterion for test completeness is introduced in this paper which measures the ability of a set of tests to distinguish between functions which are implemented by parts of programs. The criterion is applied to functions which are implemented by different kinds of programming language statements. It is more effective than branch testing and incorporates some of the advantages of mutation testing. Its effectiveness can be discussed formally and it can be

described as part of an integrated approach to testing. A tool can be used to implement the method.

[Howd81b] Abstract: The term "static analysis" has traditionally been used to refer to program analysis methods that assist the user in verifying his program, but which do not require its execution. Static analysis includes techniques which produce general information about a program, such as cross-reference tables, as well as techniques which search for particular types of errors, such as uninitialized variables. This survey describes both traditional static analysis methods as well as other validation methods that do not require program execution. It includes techniques that involve the analysis of system documents other than the program code, such as requirements and design analysis. It also includes code analysis techniques such as symbolic evaluation.

[Howd81c] Abstract: A scheme for classifying program testing methods is introduced. Program testing methods are classified according to whether they involve the generation of test data which is based on the requirements specifications, the design specifications or the source code for a program. Detailed descriptions of functional requirements and functional design based testing are included. Source code methods which are described include branch testing, path testing, file testing and expression testing. Other dynamic analysis techniques, such as dynamic assertions and recovery control blocks are also described.

[Howd82a] Abstract: Different approaches to the generation of test data are described. *Error-based* approaches depend on the definition of classes of commonly occurring program errors. They generate tests which are specifically designed to determine if particular classes of errors occur in a program. An error-based method called *weak mutation testing* is described. In this method, tests are constructed which are guaranteed to force program statements which contain certain classes of errors to act incorrectly during the execution of the program over those tests. The method is systematic, and a tool can be built to help the user apply the method. It is extensible in the sense that it can be extended to cover additional classes of errors. Its relationship to other software testing methods is discussed. Examples are included.

Different approaches to testing involve different concepts of the adequacy or completeness of a set of tests. A formalism for characterizing the *completeness* of test sets that are generated by error-based methods such as weak mutation testing as well as the test sets generated by other testing methods is introduced. Error-based, functional, and structural testing emphasize different approaches to the test data generation problem. The formalism which is introduced in the paper can be used to describe their common basis and their differences.

[Howd82b] Introduction: The software life cycle can be divided into requirements, design, programming, and maintenance. Validation has also been considered a phase of the life cycle and is sometimes inserted between programming and maintenance. Recent experience, however, indicates that validation should be integrated into all phases rather than isolated in a separate stage that takes place long after requirements and design have been completed. Studies show that the later validation is carried out, the more expensive it becomes to find errors made early in the development process.

In the integrated approach described in this article, validation is a part of each phase of the life cycle. Two validation activities–analysis and test data generation–take place during each phase. The programming and maintenance phases also include actual examination of program tests. Analysis involves the direct examination of specification and code for errors or erroneous properties. Test data generation involves the construction of test sets that are based on the important functional properties of specification and code.

[Howd85] Introduction: Program testing consists of a scattered collection of rules of thumb, coverage measures and testing philosophies. Several attempts have been made to construct theories to explain why testing works and to isolate classes of faults that can be consistently remedied by certain methods.

In one approach to testing, called functional testing, a collection of methods are integrated that can be described from the same point of view and whose effectiveness can be analyzed using a common theory.

The functional testing methods described here are suitable for module and integration testing at the development stage. The focus is on functional faults – errors caused by a program that computes the wrong function – rather than timing or performance problems. Most of the ideas are not original, but show how it all fits

together. The discussion is informal and no attempt is made to present the theory in formal definitions and theorems.

[Howd86] Abstract: An integrated approach to testing is described which includes both static and dynamic analysis methods and which is based on theoretical results that prove both its effectiveness and efficiency. Programs are viewed as consisting of collections of functions that are joined together using elementary functional forms or complex functional structures.

Functional testing is identified as the input-output analysis of functional forms. Classes of faults are defined for these forms and results presented which prove the fault revealing effectiveness of well defined sets of tests.

Functional analysis is identified as the analysis of the sequences of operators, functions, and data type transformations which occur in functional structures. Functional trace analysis involves the examination of the sequences of function calls which occur in a program path; operator sequence analysis the examination of the sequences of operators on variables, data structures, and devices; and data type transformation analysis the examination of the sequences of transformations on data types. Theoretical results are presented which prove that it is only necessary to look at interfaces between pairs of operators and data type transformations in order to detect the presence of operator or data type sequencing errors. The results depend on the definition of normal forms for operator and data type sequencing diagrams.

[Howd87] Abbreviated Preface: This book presents an integrated approach to program testing and analysis which has a sound mathematical basis. It describes both previous techniques, and how they fit together, as well as new methods. It provides a general approach to testing and validation that incorporates all important software life cycle products, including requirements and general and detailed designs. The results can be used to prove that well-defined classes of faults and failures will b liscovered by specific techniques. Functional testing and analysis is a general approach to verification and validation and not only integrates current techniques, but indicates fruitful directions for continued research and development.

[Howd89a] Overview: This paper presents a brief overview of the author's recent and current work. The main topics address the development of a general model of how software is constructed and of the reasoning errors that humans make during this process, and flavor analysis.

[Howe84] Abstract: In the areas of software development, data processing management often focuses more on coding techniques and system architecture than on how to manage the development. In recent years, "structured programming" and "structured analysis" have received more attention than the techniques software managers employ to manage. Moreover, these coding and architecture considerations are often advanced as the key to a smooth running, well managed project.

This paper documents a philosophy for software development and the tools used to support it. Those management techniques deal with quantifying such abstract terms as "productivity," "performance," and "progress," and with measuring these quantities and applying management controls to maximize them. The paper also documents the applications of these techniques on a major software development effort.

[Hsie89] Abstract: An approach to timing analysis of cyclic concurrent programs is presented. GR_0 path-expressions are used to describe synchronization and concurrency of atomic operations in cyclic concurrent programs. The behavior of a cyclic concurrent program is represented as a partial order of atomic operations, and a technique to derive this partial order from a GR_0 program is developed. Given the execution times of the individual atomic operations of a GR_0 program and a set of timing constraints, our timing analysis technique uses the partial order to determine whether the concurrent program, when executed, will satisfy the set of timing constraints. The timing analysis technique can be completely automated.

[Huan75] Abstract: One of the practical methods commonly used to detect the presence of errors in a computer program is to test it for a set of test cases. The probability of discovering errors through testing can be increased

by selecting test cases in such a way that each and every branch in the flowchart will be traversed at least once during the test. This tutorial describes the problems involved and the methods that can be used to satisfy the test requirement.

[Huan78] Introduction: It is well known that one cannot find all the errors in a program simply by testing it for a set of input data. Nevertheless, program testing is the most commonly used technique for error detection in today's software industry. Consequently, the problem of finding a program test method with increased error-detection capability has received considerable attention in the field of software research.

There appear to be two major approaches to this problem. One is to find better criteria for test-case selection. The other is to find a way to obtain additional information (i.e., information other than that provided by the output of the program) that can be used to detect errors.

The technique of program instrumentation discussed in this article can be regarded as a major outgrowth of the second approach. The main idea is to insert additional statements (instruments) into the program to be tested for the purpose of computing certain program attributes. By testing (executing) the instrumented program for a properly chosen set of test cases, we will be able to obtain the values of the program attributes automatically. The attribute values provide us with additional information for error detection. The following pages illustrate the utility of this technique and explore its potential as a tool for program validation.

[Huan79] Abstract: A data flow anomaly in a program is an indication that a programming error might have been committed. This paper describes a method for detecting such an anomaly by means of program instrumentation. The method is conceptually simple, easy to use, easy to implement on a computer, and can be applied in conjunction with a conventional program test to achieve increased error-detection capability.

[Hump88] Abbreviated Introduction: One SEI project is to provide the Defense Department with some way to characterize the capabilities of software-development organizations. The result is this software-process maturity framework, which can be used by an software organization to assess its own capabilities and identify the most important areas for improvement.

This software-development process-maturity model reasonably represents the actual ways in which software-development organizations improve. It provides a framework for assessing these organizations and identifying the priority areas for immediate improvement. It also helps identify those places where advanced technology can be most valuable in improving the software-development process.

The SEI is using this model as a foundation for a continuing program of assessments and software process development. These assessment methods have been made public, and preliminary data is now available

[Hutc83] Abstract: This paper examines the use of cluster analysis as a tool for system modularization. Several clustering techniques are discussed and used on two medium-size systems and a group of small projects. The small projects are presented because they provide examples (that will fit into a paper) of certain types of phenomena. Data bindings between the routines of the system provide the basis for the bindings. It appears that the clustering of data bindings provides a meaningful view of system modularization.

[IEEE83a] Forward: Software engineering is an emerging field. New terms are continually being generated, and new meanings are being adopted for existing terms. The Glossary of Software Engineering Terminology was undertaken to document this vocabulary. Its purpose is to identify terms currently used in software engineering and to present the current meanings of these terms. It is intended to serve as a useful reference for software engineers and for those in related fields and to promote clarity and consistency in the vocabulary of software engineering. It is recognized that software engineering is a dynamic area; thus the standard will be subject to appropriate change as becomes necessary.

[IEEE83b] Purpose: The purpose of this standard is to describe a set of basic software test documents. A standardized test document can facilitate communication by providing a common *frame of reference* (for example, a customer and a supplier have the same definition for a test plan). The content definition of a standardized test

document can serve as a completeness checklist for the associated testing process. A standardized set can also provide a baseline for the evaluation of current test documentation practices. In many organizations, the use of these documents significantly increases the manageability of testing. Increased manageability results from the greatly increased visibility of each phase of the testing process.

This standard specifies the form and content of individual test documents. It does not specify the required set of test documents. It is assumed that the required set of test documents will be specified when the standard is applied. Appendix B contains an example of such a set specification.

[IEEE83c] Scope: This standard provides minimum requirements for preparation and content of Software Configuration Management (SCM) Plans. SCM Plans document the methods to be used for identifying software product items, controlling and implementing changes, and recording and reporting change implementation status.

This standard applies to the entire life cycle of critical software; for example, where failure could impact safety or cause large financial or social losses. For noncritical software, or for software already developed, a subset of the requirements may be applied.

This standard identifies those essential items that shall appear in all Software Configuration Management Plans. In addition to those items, the users of this standard are encouraged to incorporate additional items into the plan, as appropriate, to satisfy unique configuration management needs, or to modify the contents of specific sections to fully describe the scope and magnitude of the software configuration management effort. Where this standard is invoked for a project engaged in producing several software items, the applicability of the standard shall be specified for each of the software product items encompassed by the project.

Examples are incorporated into the text of this standard to enhance clarity and to promote understanding. Examples are either explicitly identified as such, or can be recognized by the use of the verb *may*. Examples shall not be construed as mandatory implementations.

[IEEE84] The purpose of this standard is to provide uniform, minimum acceptable requirements for preparation and content of Software Quality Assurance Plans (SQAP).

In considering adoption of this standard, regulatory bodies should be aware that specific application of this standard may already be covered by one or more IEEE or ANSI standards documents relating to quality assurance, definitions, or other matters. It is not the purpose of IEEE Std 730 to supersede, revise or amend existing standards directed to specific industries or applications.

This standard applies to the development and maintenance of critical software; for example, where failure could impact safety or cause large financial or social losses. For non-critical software, or for software already developed, a subset of the requirements of this standard may be applied.

The existence of this standard should not be construed to prohibit additional content in a Software Quality Assurance Plan. An assessment should be made for the specific software product item to assure adequacy of coverage. Where this standard is invoked for a project engaged in producing several software items, the applicability of the standard should be specified for each of the software product items encompassed by the project.

[IEEE88] Scope: This standard provides a methodology for establishing quality requirements and identifying, implementing, analyzing and validating software quality metrics. This methodology applies to all software at all phases of the software life cycle. As a standard, this methodology is mandatory, though not exhaustive in its implementation details.

Software quality is the degree to which software possesses a desired combination of attributes. By definition, software quality is relative; it varies from system to system as requirements vary. Likewise, the set of applicable metrics used to measure software quality varies from system to system. For this reason, this standard does not prescribe specific metrics. However, the appendices include examples of metrics together with a complete example of the use of this standard.

A software quality metric is a function whose inputs are software data and whose output is a single (numerical) value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.

[Iann84] Abstract: A set of criteria is proposed for the comparison of software reliability models. The intention is to provide a logically organized basis for determining the superior models and for the presentation of model characteristics. It is hoped that in the future, a software manager will be able to easily select the model most suitable for his requirements from among the preferred ones.

[Ibar82] Abstract: We consider a simple class of loop-free programs whose instruction repertoire consists of $x \leftarrow 0$, $x \leftarrow c$, $x \leftarrow cx$, $x \leftarrow x/c$, $x \leftarrow x + y$, $x \leftarrow x - y$, **skip** l, **if** $p(x,y)$ **then skip** l, and **halt**. (x and y are integer variables, c is a positive integer, x/c is integer division, l is a nonnegative integer, and $p(x,y)$ is a predicate of the form $x > y$, $x >= y$, $x = y$, $x != y$, $x <= y$, or $x < y$; **skip** l causes the $(l+1)$st instruction following the current instruction to be executed next.) We show that the equivalence problem for this class is decidable in 2an2 time (N = sum of the sizes of the programs and – is a fixed positive constant). The bound cannot be reduced to a polynomial in N unless P=NP. In fact, we have the following rather surprising result: The equivalence problem for programs with one input variable (which also serves as the output variable) and one auxiliary variable using only instructions $x \leftarrow 2x$, $x \leftarrow x/2$, and $x \leftarrow x + y$ is NP-hard.

[Ingl86] Abbreviated Introduction: Standard measures of software quality have been set up for AT&T Bell Laboratories. These metrics allow a software project to be followed through its development, controlled introduction, and release to customers. The metrics serve both project and corporate management needs. For project management, they allow more effective management of development effort, and they help ensure a fast and effective solution to problems that arise at any stage. For corporate management, they provide a vehicle for quantifying the overall quality of software development, for setting quality improvement objectives, and for tracking results. In particular, the metrics provide quantitative information on the number of faults, normalized so that corporate results can be summarized and projects of different size can be compared; the responsiveness of support organizations in resolving problems; and the impact of fixes on customers.

[Isod87] Introduction: Debugging programs involves repeating several steps: executing test programs, detecting errors, investigating their causes, correcting them, and recompiling. The most time-consuming, difficult step is the detection and investigation of errors.

Several researchers have tried to find an effective way to represent execution behavior for program debugging. Cargill's Blit debugger uses multiple windows that simultaneously show program execution output, interaction with the debugger, and program text. Myers' Incense debugger displays the variables' data structures, but does not address the dynamic features of program execution. Reiss' Pecan debugger does propose dynamic presentation of the control flow on a program flowchart, but it displays character-based data just like traditional debugging tools. The Balsa and program visualization systems can represent data in figures that are appropriate to their meaning. However, they do not have facilities to represent control flow of blocks or dataflow among variables.

Therefore, these systems are insufficient for debugging programs because they represent only a part of program execution behavior.

We have developed a visual debugger for Ada programs, the Visual and Interactive Programming Support. VIPS uses graphics to show the static and dynamic behavior of program execution. It greatly reduces the time and effort in program debugging because it helps a programmer detect and localize program errors more easily than with traditional tools.

[Itak82] Abstract: Estimation of the size and time required for software development is probably the most difficult aspect of any project. Up to now, most estimates have been done subjectively by experts. These estimates are often inaccurate. In the midst of development, faulty estimates may contribute to delays and/or excess expenses.

In the last several years, several estimation models have been proposed, most of which were models to estimate software development cost (manpower). These models used program size as a variable. However at the beginning of development, when estimations are made, program sizes are usually uncertain and costs (manpower) are equally uncertain.

The authors developed a program-size estimation model for batch programs in a banking system, and used the model in an actual project. Using the adapted model, estimation errors amounted to only 7 percent. This is much better than the accuracy of estimations made by experts in the field (usually about 10 percent accuracy), and indicates that objective estimation methods can be derived for program-size.

In this paper, we introduce our estimation model and discuss the adaptation of that model for a specific project.

[Ives83] Abstract: This paper critically reviews measures of user information satisfaction and selects one for replication and extension. A survey of production managers is used to provide additional support for the instrument, eliminate scales that are psychometrically unsound, and develop a standard short form for use when only an overall assessment of information satisfaction is required and survey time is limited.

[Jach84] Abstract: The occurrence of a data flow anomaly is often an indication of the existence of a programming error. The detection of such anomalies can be used for detecting errors and to upgrade software quality. This paper introduces a new, efficient algorithm capable of detecting anomalous data flow patterns in a program represented by a graph. The algorithm based on static analysis scans the paths entering and leaving each node of the graph to reveal anomalous data action combinations. An algorithm implementing this type of approach was proposed by Fosdick and Osterweil [Fosd76a]. Our approach presents a general framework which not only fills a gap in the previous algorithm, but also provides time and space improvements.

[Jack71] Introduction. In April 1966 work was initiated by Martin Marietta Corporation (MMC), Denver Division, to extend the role of an airborne computer to include flight controls as well as guidance and navigation computations. This project is one of several significant improvements for the Titan IIIC space booster which was funded by the Space and Missile Systems Organization of the Air Force. The new Digital Flight Control System (DFCS) has been successfully tested in four (4) Titan IIIC missions.

The purpose of this paper is to describe how a large hybrid computer simulation was used as an aid to design and develop the DFCS and then used to validate the resulting DFCS airborne software.

The simulation was programmed in six degrees-of-freedom and included an airborne Univac 1824M Missile Guidance Computer (MGC) in the closed loop. Additional computing equipment used in the simulation included three (3) EAI 8800 analog computers, an EAI 8400 digital computer and an SDS 930 digital computer. Flight control hardware components such as rate gyros, body mounted accelerometers, and hydraulic actuators were also used in the simulation.

[Jaha86] Abstract: Two important characteristics of time-critical systems are: the requirement to satisfy stringent timing constraints, and the need to guard against an imperfect execution environment. In this paper, we formalize the safety analysis of timing properties in real-time systems. Our analysis is based on a formal logic: RTL (Real-Time Logic) which is especially suitable for reasoning about the timing behavior of systems. Given the formal specification of a system and a safety assertion to be analyzed, our goal is to relate the safety assertion to the systems specification. There are three distinct cases: 1) the safety assertion is a theorem derivable from the systems specification, 2) the safety assertion is unsatisfiable with respect to the systems specification, or 3) the negation of the safety assertion is satisfiable under certain conditions. A systematic method for performing safety analysis will be presented.

[Jalo89] Abstract: Specifications are means to define formally the behavior of a system or a system component. Completeness is a desirable property for specifications. In this paper, we describe a system that *tests* for the completeness of axiomatic specifications of abstract data types. For testing, the system generates a set of test cases and an implementation of the data type from the specifications. The generated implementation is such that if the specifications are not complete, the implementation is not complete, and the behavior of all of the sequences of valid operations on the data type is not defined. This implementation is tested with the generated test cases to detect the incompleteness of specifications. The system is implemented on a VAX system running Unix.

[Jard83] Abstract: An approach to testing the consistency of specifications is explored, which is applicable to the design validations of communication protocols and other cases of step-wise refinement. In this approach, a testing module compares a trace of interactions obtained from an execution of the refined specification (e.g. the protocol specification) with the reference specification (e.g. the communication service specification).

Non-determinism in reference specifications presents certain problems. Using an extended finite state transition model for the specifications, a strategy for limiting the amount of non-determinacy is presented.

An automated method for constructing a testing module for a given reference specification is discussed. Experience with the application of this testing approach to the design of a Transport protocol and a distributed mutual exclusion algorithm is described.

[Jeff85] Abstract: This paper reviews a study on programming productivity carried out to (i) confirm previous published results, (ii) explore the impact of the changing programming environment on productivity, and (iii) examine the influence on productivity of organizational factors. Programming productivity data were collected from 17 organizations and analyzed in the light of data collected some 4 years earlier. While significant technological changes were observed to have occurred in the programming environment, the results of the later study were in many respects almost identical to those obtained earlier, thus validating the previous study. The organizational variables collected revealed a very strong relationship between a programmer's attitude to his supervisor and programming productivity.

[Jeli72] Introduction and Summary: Software reliability study was initiated by Advanced Information Systems subdivision of McDonnell Douglas Astronautics Company, Huntington Beach, California, to conduct research into the nature of the software reliability problem including definitions, contributing factors and means for control.

Discrepancy reports which originated during the development of two large-scale real-time systems form two separate primary data sources for the reliability study. A mathematical model, descriptively entitled the De-Eutrophication Process, was developed to describe the time pattern of the occurrence of discrepancies (errors). This model has been employed to estimate the initial (or residual) error content in a software package as well as to estimate the time between discrepancies at any phase of its development. A means of predicting mission success on the basis of errors which occur during testing are described.

Problems in categorizing software anomalies are described and the special area of the genesis of discrepancies during the integration of modules is discussed. Management techniques which should reduce the number of software anomalies are described.

[John75] Abstract: The report contains plans for a complete software reliability measurement program using both manual and automatic data entry. The program is to be run in conjunction with SAMTEC at Vandenburg AFB in an effort to establish measurement and evaluation criteria for the advanced systematic techniques for reliable operational software (ASTROS) project. An integral part of that project is the implementation and evaluation of structured programming techniques.

Included in the report are all forms necessary to describe the software development environment, the hierarchy and size of programming modules, and to capture any significant events that will affect programming and test while they are in progress. Forms and instructions for their use for manual data collection are included, as are descriptions of items that could be collected automatically.

[John79] Summary: Designers and implementors of high-level language translators can, with relatively little extra effort, greatly facilitate run-time symbolic debugging. Practical suggestions are presented, based on experiences gained from interfacing several compilers with a run-time debugging system.

[John82b] Abstract: This glossary contains 291 definitions of terms dealing with the debugging of computer software. The list includes numerous synonyms, as well as the proper names of debugging systems described in the open literature. Terms and definitions have been obtained from various sources: the software-engineering literature, other software-engineering glossaries, and individual contributions.

[John83] Abstract: This paper deals with issues that have emerged as a result of a successful implementation of a source level symbolic debugger for HP-1000 computer systems. By analyzing a user's thought processes during a debugging session we created a powerful and easy to use tool for program analysis.

[John84] Abstract: This paper describes a program called PROUST which does online analysis and understanding of Pascal programs written by novice programmers. PROUST takes as input a program and a non-algorithmic description of the program requirements, and finds the most likely mapping between the requirements and the code. This mapping is in essence a reconstruction of the design and implementation steps that the programmer went through in writing the program. A knowledge base of programming plans and strategies, together with common bugs associated with them, is used in constructing this mapping. Bugs are discovered in the process of relating plans to the code; PROUST can therefore give deep explanations of program bugs by relating the buggy code to its underlying intentions.

[Jone78] Overview: Discussed is the unit-of-measure situation in programming. An analysis of common units of measure for assessing program quality and programmer productivity reveals that some standard measures are intrinsically paradoxical. Lines of code per programmer-month and cost per defect are in this category. Presented here are attempts to go beyond such paradoxical units as these. Also discussed is the usefulness of separating quality measurements into measures of defect removal efficiency and defect prevention, and the usefulness of separating productivity measurements into work units and cost units.

[Jone81] Abstract: Programming productivity has become a significant topic for a number of the world's industrial, commercial, governmental, and university communities. The decade from 1970 to 1980 witnessed an unprecedented growth in computers and programming, that was accompanied by unprecedented problems with costs, quality, schedules, and low productivity. Current research indicates that the greatest barrier to improved productivity lies in the enormous costs which are associated with programming defect removal and with paperwork. Therefore the most direct strategy for improving productivity is to concentrate on methods that simplify complexity, improve requirements and design, minimize paperwork, and reduce errors. However, attempts to move toward these goals reveal underlying problems whose solutions will require the application of concepts from disciplines outside of programming, such as linguistics and perceptual psychology. Programming is becoming a catalyst that has the potential of forming new and synergistic combinations of ideas.

By the mid 1980's, software is no longer an "outcast" technology regarded as inferior by older sciences. Accurate metrics of software projects and the new engineering principles supporting reusable designs and reusable code are giving software a new professionalism. The advent of new human interface techniques derived from object-oriented programming methods may be leading to a new plateau, in which human control and usage of complex devices is more natural and intuitive than at any time in history.

This tutorial volume on productivity issues for the eighties attempts to place programming in context with other disciplines, and addresses five major topics: (1) Programming measurements, (2) programming life-cycle analysis, (3) programming requirements and design methods, (4) programming environments, and (5) the new science of software.

[Joyc87a] Abstract: A computerized theraputic radiation machine has been blamed in incidents that have led to the deaths of two patients and serious injuries to several others. The deadly medical mystery used by the machine was finally traced to a software bug, "Malfunction 54," named after the message displayed on the operator console. The affair is seen as epitomizing the software reliability crisis at its worst, and raises the thorny legal issue of liability for personal injuries caused by defective programs. The pending lawsuits over the malfunctioning machine may set a legal precedent that could affect all computer users and vendors. Ultimately, such cases call into question our increasing dependence on computers for everything from banking to national defense.

[Joyc87b] Abstract: The monitoring of distributed systems involves the collection, interpretation, and display of information concerning the interactions among concurrently executing processes. This information and its display can support the debugging, testing, performance evaluation, and dynamic documentation of distributed

systems. General problems associated with monitoring are outlined in this paper, and the architecture of a general purpose, extensible, distributed monitoring system is presented. Three approaches to the display of process interactions are described: textual traces, animated graphical traces, and a combination of aspects of the textual and graphical approaches. The roles that each of these approaches fulfill in monitoring and debugging distributed systems are identified and compared. Monitoring tools for collecting communication, statistics, detecting deadlock, controlling the non-deterministic execution of distributed systems, and for using protocol specifications in monitoring are also described.

Our discussion is based on experience in the development and use of a monitoring system within a distributed programming environment called Jade. Jade was developed within the Computer Science Department of the University of Calgary and is now being used to support teaching and research at a number of university and research organizations.

[Kafu81] Overview: We state a set of criteria that has guided the development of a metric system for measuring the quality of a large-scale software product. This metric system uses the flow of information within the system as an index of system interconnectivity. Based on this observed interconnectivity, a variety of software metrics can be defined. The types of software quality features that can be measured by this approach are summarized. The data-flow analysis techniques used to establish the paths of information flow are explained and illustrated. Finally, a means of integrating various metrics and models into a comprehensive software development environment is discussed. This possible integration is explained in terms of the Gandalf system currently under development at Carnegie-Mellon University.

[Kafu85a] Abstract: In this paper are presented the results of a study in which several production software systems are analyzed using ten software metrics. The ten metrics include both measures of code details, measures of structure, and combinations of the two. Historical data recording the number of errors and the coding time of each component are used as objective measures of resource expenditure of each component. The metrics are validated by showing: (1) the metrics singly and in combination are useful indicators of those components which require the most resources, (2) clear patterns between the metrics and the resources expended are visible when both resources are accounted for, (3) measures of structure are as valuable in examining software systems as measures of code details, and (4) the choice of which, or how many, software metrics to employ in practice is suggested by measures of "yield" and "coverage".

[Kafu85b] Abstract: This paper reports on an effort to relate seven different software quality metrics to the experience of maintenance activities performed on a medium size data base system. Three different versions of the data base system that evolved over a period of three years were analyzed in this study. A major revision of the data base system, while still in its design phase, was also analyzed.

The results of this study indicate: (1) that the growth in system complexity as determined by the software metrics agree with the general character of the maintenance tasks performed in successive versions; (2) the metrics were able to identify the improper integration of functional enhancements made to the system; (3) the complexity values of the system components as indicated by the metrics, conform well to an intuitive understanding of the system by people familiar with the system; and (4) an analysis of the redesigned version of the data base system showed the ·· efulness of software metrics in the (re)design phase by revealing a poorly structured component of the syst ..

[Kafu88] Abstract: This paper reports the results of a study which examined the relationship between a collection of software metrics and the development data (such as errors and coding time) of three commercially produced software systems. The software metrics include both measures of system interconnectivity and measures of system code. This study revealed strong relationships between the metrics and the development data when individual components were aggregated by structure (into subsystems) or by similarity (into groups). The subsystem and group results imply that research and application of metrics can guide the effective application of project resources by identifying those groups which, for example, will contain a disproportionately large fraction of errors. Finally, the study showed the overall utility of two interconnectivity metrics: Henry and Kafura's

information flow metric and McClure's invocation metric. This result is significant because interconnectivity metrics can be applied early in the life cycle.

[Kahn77] Abstract: The concept of *coroutine* or *process* is useful in a large class of applications, usually involving incremental generation of transformation of data. We present a language based on a clear semantics of process interaction, which facilitates well-structured programming of dynamically evolving networks of processes. These networks exhibit the same input/output behavior whether they are executed sequentially or in parallel. Sample program proofs are used to illustrate the benefits of the language's simple denotational semantics. The language serves also to clarify the relationships between coroutines, call-by-need, dynamic data structures and parallel computation.

[Kant80] Abstract: In this paper we consider the application of the recovery block concept to parallel programs for ensuring increased reliability despite the presence of software bugs. The basic idea of this technique is to include standby software components in the program which can be "switched on" in case the active component fails. However, before this could be done, the system must be rolled back to a consistent state. One of the goals in this rollback is to avoid undoing a large amount of computation. It is shown that the process interaction must be severely constrained in order to achieve this goal. Sufficient conditions for limiting the rollback in a system of processes communicating via monitors is also presented.

[Kapp88] Abstract: IDA Paper P-2028 documents a tool that can facilitate the description of processes for the Strategic Defense System (SDS) and Battle Management/Command, Control and Communications (BM/C3) architectures. The process descriptions generated by this tool conform to the Strategic Defense Initiative Organization (SDI) Architecture Dataflow Modeling Technique (SADMT).

[Katz87] Abstract: Packages, subprograms, generics, and tasks are the building blocks of Ada systems. They can combine to hide information, group information, isolate dependencies, and create reusable pieces. However, different people view them from different perspectives for different purposes. Therefore, people have different expectations when they discuss modules and modularity. In this paper, we describe four definitions that divide systems into program blocks, or modules, using various structural criteria. We use this common terminology to show how the different views on the system can help us better understand the modularity of the system. Finally, we use programs from studies comparing design techniques and measuring maintainability to show how these ideas may be applied.

[Kear86] Abbreviated Introduction: Inappropriate use of software complexity measures can have large, damaging effects by rewarding poor programming practices and demoralizing good programmers. Software complexity measures must be critically evaluated to determine the ways in which they can best be used.

[Keil87] Abstract: We suggest that users are interested solely in the quality of prediction which can be obtained from software reliability models. Some ways of analyzing the quality of predictions are proposed and several models and inference procedures are compared against software failure data sets. We conclude that some predictions are extremely poor, notably those arising from ML analysis of the Jelinski-Moranda model. Others are quite good. We suggest promising areas for future work.

[Kell76] Abstract: Two formal models for parallel computation are presented: an abstract conceptual model and a parallel-program model. The former model does not distinguish between control and data states. The latter model includes the capability for the representation of an infinite set of control states by allowing there to be arbitrarily many instruction pointers (or processes) executing the program. An induction principle is presented which treats the control and data state sets on the same ground. Through the use of "place variables," it is observed that certain correctness conditions can be expressed without enumeration of the set of all possible control states. Examples are presented in which the induction principle is used to demonstrate proofs of mutual exclusion. It is shown that assertions-oriented proof methods are special cases of the induction principle. A

special case of the assertions method, which is called parallel place assertions, is shown to be incomplete. A formalization of "deadlock" is then presented. The concept of a "norm" is introduced, which yields an extension, to the deadlock problem, of Floyd's technique for proving termination. Also discussed is an extension of the program model which allows each process to have its own local variables and permits shared global variables. Correctness of certain forms of implementation is also discussed. An appendix is included which relates this work to previous work on the satisfiability of certain logical formulas.

[Kell83] Abstract: In this paper the design diversity approach of fault-tolerant multi-version software as a complement to fault avoidance is discussed and an experiment is described in which 32 programmers were employed. It was found that formal specification languages, while showing promise for the future, are presently very difficult to use and understand, and are severely limited in power. Software errors encountered in the experiment were studied and classified. The increase in reliability seen in multi-version software over individual-version software was substantial; it was even possible to combine three faulty versions and produce a combination that was completely fault-tolerant.

[Kell85a] Abstract: ADAMAT, an Ada Measurement and Analysis Tool, provides immediate assistance for 1) improving the quality of Ada software, and 2) training Ada programmers. The underlying metrics framework is hierarchical based on the McCall metrics framework, tailored to the Ada language, and formally defined using Prolog. The automated data collection component is automatically generated using compiler generation techniques, which include a descriptive technique for describing pattern matching in a well-defined language. The quality analysis component, based on the formal definition of the metrics, provides users with interactive analysis of the metric data, and allows users to step through the Ada metrics hierarchy to pinpoint problem areas.

[Kell85b] Abbreviated Introduction: This paper discusses an approach to creating an automated metrics tool for measuring the level of adherence to software quality guidelines in Ada source code. The tool provides managers with the visibility needed to control the application of quality guidelines on software programs.

Software quality management is very difficult, because the quality of software is not transparent; its assessment requires a thorough review of specifications and code. By formalizing software quality principles, we can develop a tool that helps monitor their use.

[Kemm80] Abstract: This paper gives an overview of the Formal Development Methodology (FDM) and the Ina Jo formal specification language. FDM is an integrated methodology for the design, specification, implementation, and verification of software. It enforces rigorous connections between successive stages of development. The components of the FDM are the Ina Jo formal specification language, the specification processor, the interactive theorem prover, and a verification condition generator.

This paper gives an overview of each of the components and discusses how each fits into the overall verification process. Examples of the different constructs in the specification language are presented as well as a sample two-level formal specification.

[Kemm85a] Abstract: Formal specification and verification techniques are now used to increase the reliability of software systems. However, these approaches sometimes result in specifying systems that cannot be realized or that are not usable. This paper demonstrates why it is necessary to test specifications early in the software life cycle to guarantee a system that meets its critical requirements and that also provides the desired functionality. Definitions to provide the framework for classifying the validity of a functional requirement with respect to a formal specification are also introduced. Finally, the design of two tools for testing formal specifications is discussed.

[Kemm86] Abstract: This is the first volume of the final report of a verification assessment study that was begun in November 1984 and lasted for approximately nine months. The final report consists of five volumes. This volume contains an overview of the study, some conclusions that were formulated, and directions for future

research efforts in formal verification.

The main goal of this effort was a technology interchange among the developers of four established verification systems. The systems investigated were i) Affirm (General Electric Company, Schenectady, New York), ii) FDM (System Development Corporation - A Burroughs Company, Santa Monica, California) iii) Gypsy (the University of Texas at Austin, Austin, Texas), and iv) Enhanced HDM (SRI International, Menlo Park, California).

There was some comparative work on examples, but the main idea was for the developers to learn the details of each other's system as a basis for future development. It would have been interesting and informative to look at other systems, but time did not allow for this.

It was **not** the goal of this study to rate the verification systems that were investigated. It was also **not** the intent of the study to justify the need for formal specification and verification systems or to justify the necessity for research in this area.

[Kern74a] Abbreviated Introduction: Good programming cannot be taught by preaching generalities. The way to learn to program well is by seeing, over and over, how real programs can be improved by the application of a few principles of good practice and a little common sense. Practice in critical reading leads to skill in rewriting, which in turn leads to better writing.

This book is a study of a large number of "real" programs, each of which provides one or more lessons in style. We discuss the shortcomings of each example, rewrite it in a better way, then draw a general rule from the specific case. The approach is pragmatic and down-to-earth; we are more interested in improving current programming practice than in setting up an elaborate theory of how programming should be done.

The examples we give are all in Fortran and PL/I, since these languages are widely used and are sufficiently similar that a reading knowledge of one means that the other can also be *read* well enough. (We avoid complicated constructions in either language and explain unavoidable idioms as we encounter them.) *The principles of style, however, are applicable in all languages, including assembly codes.*

[Kern74b] Abstract: Computer programs can be written many different ways and still achieve the same effect. Until recently, programmers have had little reason to favor one method of expressing code over another. We have come to learn, however, that functionally equivalent programs can have extremely important *stylistic* differences.

Good programming style cuts across application areas, technique and language. Programs written with good style are easier to read and understand, and often smaller and more efficient, than those written badly. Yet few programmers have ever been taught what style is, as we can see from even cursory inspection of their code. Even the techniques of structured programming do not ensure that code will be good; "structured" programs can be just as bad as their unstructured counterparts.

This paper is a survey of some aspects of programming style, primarily expression and structure, showing by examples what happens when principles of style are violated, and what can be done to improve programs. To add the ring of truth to our discussion, the examples are all taken verbatim from programming textbooks.

[Kern81] Table of Contents: Filters. Files. Sorting. Text Patterns. Editing. Formatting. Macro Processing.

[Kieb83] Abstract: Abstract data types, and in particular those that are designed to provide resources for use by concurrently executable programs, are often designed to be used only in certain ways. The intended constraints on use of an instance of such a type can be expressed in two principle ways: as assertions on the domain of the values input to each operator, and as constraints on the sequences in which the operators of the type can be called by a customer process. These constraints must be enforced in the environment in which an instance of the type is used. Nevertheless, they are very much a part of the type specification, for its definition is not complete, nor can the consistency of its representation be proved, without them.

A notation is provided in which to express sequential constraints, which are here called *access-right expressions*. It is suggested that these expressions should be declared in a programming language that supports the definition of monitors or resource managers. Implications for the proof rules of monitors are discussed, and suggestions are made for a programming language implementation.

[King75a] Abstract: The current approach for testing a program is, in principle, quite primitive. Some small sample of the data that a program is expected to handle is presented to the program. If the program produces correct results for the sample, it is assumed to be correct. Much current work focuses on the question of how to choose this sample. We propose that a program can be more effectively tested by executing it "symbolically." Instead of supplying specific constants as input values to a program being tested, one supplies symbols. The normal computational definitions for the basic operations performed by a program can be expanded to accept symbolic inputs and produce symbolic formulae as output.

If the flow of control in the program is completely independent of its input parameters, then all output values can be symbolically computed as formulae over the symbolic inputs and examined for correctness. When the control flow of the program is input dependent, a case analysis can be performed producing output formulae for each class of inputs determined by the control flow dependencies. Using these ideas, we have designed and implemented an interactive debugging/testing system called EFFIGY.

[King76] Abstract: This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretatively executes programs written in a simple PL/1 style programming language. It includes many standard debugging features, the ability to manage and prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program proving is also included.

[Kitc81] Abstract: The increasing cost of software development and maintenance has revealed the need to identify methods that encourage the production of high quality software. This in turn has highlighted the need to be able to quantify factors influencing the amount of effort needed to produce such software, such as program complexity.

Two approaches to the problem of identifying complexity metrics have attracted interest in America; the theoretical treatment of software science by Halstead of Purdue University and the graph-theoretical concept developed by McCabe of the US Department of Defense. This paper reports an attempt to assess the ability of the measures of complexity proposed by these authors to provide objective indicators of the effort involved in software production, when applied to selected subsystems of the ICL operating system VME/B. The proposed metrics were computed for each of the modules comprising these subsystems, also counts of the numbers of machine-level instructions (Primitive Level Instructions, 'PLI') and measures of the effort involved in bringing the modules to an acceptable standard for field release. It was found that all the complexity metrics were correlated positively with the measure of effort, those modules which had proved more difficult having large values for all these metrics. However, neither Halstead's nor McCabe's metrics offered any substantial improvement over the simple 'PLI' count as predictors of effort.

[Knig85a] Abstract: This paper describes an experiment in which simple syntactic alterations were introduced into program text in order to evaluate the testing strategy known as *error seeding*. The experiment's goal was to determine if randomly placed syntactic manipulations can produce failure characteristics similar to those of indigenous errors found within unseeded programs. As a result of a separate experiment, several programs were available, all of which were written to the same specifications and thus were intended to be functionally equivalent programs allowed the influence of individual programmer styles to be removed as a variable from the error seeding experiment. Each of six different syntactic manipulations were introduced into each program and the mean times to failure for the seeded errors were observed. The seeded errors were found to have a broad spectrum of mean times to failure independent of the syntactic alteration used. We conclude that it is possible to seed errors using only simple syntactic techniques that are *arbitrarily* difficulty to locate. In addition, several unexpected results indicate that some issues involved in error seeding have not been addressed previously.

[Knig85b] Abstract: Symbolic execution is the execution of a computer program with symbolic rather than actual values. It has been proposed as a method of proving that a program is correct but has only been applied previously to sequential programs. The introduction of concurrency into programming languages provides many new opportunities for programming errors, and, because of the nondeterminism, errors in concurrent programs are often harder to find than errors in sequential programs. In this paper we discuss a system than symbolically executes concurrent programs. Rather than deal with concurrency in general, the system described here deals with the concurrent aspects of a specific programming language; namely Ada. We chose deliberately to investigate all the detail of an actual programming language, and we chose Ada because of its modern design and expected widespread use. Our goal was to attempt the symbolic execution of the concurrent features of Ada to see whether useful diagnostic information about erroneous Ada programs could be generated. The system described is partially implemented and correctly identifies errors that are not caught by compilers.

[Knig86a] Abstract: N-version programming has been proposed as a method of incorporating fault-tolerance into software. Multiple versions of a program (i.e., "*N*") are prepared and executed in parallel. Their outputs are collected and examined by a voter, and, if they are not identical, it assumes that the majority is correct. This method depends for its reliability improvement on the assumption that programs that have been developed independently will fail independently. In this paper, an experiment is described in which the fundamental axiom is tested. A total of 27 versions of a program were prepared independently from the same specification at two universities and then subjected to one million tests. The results of the tests revealed that the programs were individually extremely reliable but that the number of tests in which more than one program failed was substantially more than expected. The results of these tests are presented along with an analysis of some of the faults that were found in the programs. Background information on the programmers used is also summarized. The conclusion from this experience is that N-version programming must be used with care and that analysis of its reliability must include the effect of dependent errors.

[Knut71] Summary: A sample of programs, written in FORTRAN by a wide variety of people for a wide variety of applications, was chosen 'at random' in an attempt to discover quantitatively 'what programmers really do.' Statistical results of this survey are presented here, together with some of their apparent implications for future work in compiler design. The principal conclusion which may be drawn is the importance of a program 'profile,' namely a table of frequency counts which record how often each statement is performed in a typical run; there are strong indications that profile-keeping should become a standard practice in all computer systems, for casual users as well as system programmers. This paper is the report of a three month study undertaken by the author and about a dozen students and representatives of the software industry during the summer of 1970. It is hoped that a reader who studies this report will obtain a fairly clear conception of how FORTRAN is being used, and what compilers can do about it.

[Knut73] Abstract: A procedure recently devised by A. Nahapetian, for reducing the number of measurements needed to determine all the execution frequencies in a computer program, is shown to be optimal, by interpreting the procedure in a new way.

[Kopp76] Abbreviated Introduction and Summary: The Process Design Engineering Program, under the direction of the Ballistic Missile Defense Advanced Technology Center, has as its objective the development of a unified software engineering discipline addressing all software development problems from receipt of software requirements to delivery of the operational software system. During the first two years of this program, initial process design engineering and management procedures were developed which led to the systematic top-down development of real-time software processes. A prototype set of software tools to support these procedures was designed and implemented as Process Design System 1 (PDS 1), and an experimental BMD baseline software process was then designed and implemented using these techniques and tools.

This paper concentrates on some of the features of the Process Design System 2 and the manner in which its components interact. Special emphasis is placed on the error detection capability of the system and the characteristics of the Process Design Language (PDL). The current status of PDS and planned future efforts are

discussed.

[Kore88] Abstract: A recently developed, experimental, integrated System for Testing And Debugging is presented. Its testing part supports three data flow coverage criteria. The debugging part guides the programmer in the localization of faults by generating and interactively verifying hypotheses about their location.

[Krau88] Abstract: This paper describes how software testing using mutation analysis can be performed very efficiently on an SIMD machine. Mutation analysis provides effective means of determining the reliability of large software systems. However, the cost of conducting such a software test can be computationally expensive. Current implementations [of] mutation tools are unacceptably slow and are only suitable for testing relatively small programs.

Our research has shown that most of the general purpose machine architectures available commercially can be utilized efficiently to carry out cost effective mutation analysis software testing. We have shown this to be the case for *vector-multiprocessors*. In this paper, we develop a technique that permits *unified* scheduling of multiple mutant programs on a very large SIMD machine. We believe that, for the first time in the field of software testing, supercomputers with novel architectures can be used to enhance software productivity by employing techniques like the one proposed in this paper.

[Krie80] Abstract: ANNA is a proposal to extend Ada to include facilities for formally specifying the intended behavior of Ada programs (or portions thereof) at all stages of program development. ANNA programs are Ada programs with formal comments. Formal comments in ANNA consist of virtual Ada text and annotations. The syntax and semantics of different kinds of annotations are defined: declarative annotations (for variables, subtypes, subprograms, and packages), statement annotations, exception annotations, and visibility annotations. ANNA includes a small number of predefined attributes which may appear only in annotations, e.g., access type collections.

The lexical structure of ANNA is designed so that the extensions of Ada appear as Ada comments. ANNA programs are therefore acceptable by Ada translators. The semantics of annotations are defined in terms of Ada concepts, in particular many annotations are generalizations of the constraint concept. It is therefore a simple step for the Ada programmer to use ANNA to give formal specifications of programs.

ANNA is intended to provide a formal framework within which different theories of formal specifications may be applied to Ada. Our proposal omits tasking for the time being.

[Krie83] Abstract: One of the major concerns in the design of the Ada programming language was software reliability. Rigid rules are stated in the language definition that allow checking of program properties either statically (i.e., during the compilation) or dynamically (i.e., during the execution). In fact, Ada compilers are required to perform those checks and give error messages during compilation for static errors, and raise predefined exceptions during execution for dynamic errors. If a dynamic error can be anticipated during compilation, a warning may be given, but the respective exception must still be raised in case the program is submitted for execution.

[Krie86] Summary: The PROSPECTRA project aims to provide a rigorous methodology for developing correct software and a comprehensive support system. It is sponsored by the Commission of the European Communities under the ESPRIT Programme, ref. 390.

The *methodology* integrates program construction and verification during the development process. User and implementor start with a formal specification, the interface or "contract". This initial specification is then gradually transformed into an optimized machine-oriented executable program. The final version is obtained by stepwise application of transformation rules. These are carried out by the system, with interactive guidance by the implementor, or automatically by compact transformation tools.

The final version is correct by construction; only the applicability of transformation rules needs to be verified at each step, assisted by the system. Transformation rules are proved correct, analogously to theorems. They form the nucleus of an extendible knowledge base, the method bank, together with pre-fabricated program components, previous program versions, and entire development histories that can be replayed.

The strict methodology of Program Development by Transformation (based on the CIP approach) is completely supported by the system, enabling the construction of "a priori" correct programs from formal specifications. However, the system also allows other program development styles where the user assumes responsibility for unguarded development transitions. Moreover, it will be possible to integrate existing program components based on their specification, and to develop them further.

The *system* comprises basic components for the application of individual transformation rules and of compact development methods described as transformation scripts; these provide its real power. Any kind of system activity is conceptually and technically regarded as a transformation of a "program" at one of the system layers. This provides for a uniform user interface, reduces system complexity, and allows the construction of system components in a highly generative way.

[Krug88] Abbreviated Introduction: Significant improvement in software reliability calls for innovative methods for developing software, determining its readiness for release, and predicting field performance. This paper focuses on three supporting strategies for improving software quality. First, there is a need for a metric or a set of metrics to help make the decision of when to release the product for customer shipments. Second, accurately estimating the duration of system testing, while not directly contributing to reliability, makes for a smoother introduction of the product to the marketplace.

Finally, achieving significant improvement is easier given the ability to predict field failure rates, or perhaps more realistically, to compare successive software products upon release. Although this third strategy will be discussed in this paper, the emphasis will be on choosing the right software reliability metric and confidently managing the testing effort with the aid of software reliability growth models.

[Lam84] Abstract: The method of projections is a new approach to reduce the complexity of analyzing nontrivial communication protocol entities and communication channels. Protocol entities interact by exchanging messages through channels; messages in transit may be lost, duplicated as well as reordered. Our method is intended for protocols with several distinguishable functions. We show how to construct image protocols for each function. An image protocol is specified just like a real protocol. An image protocol system is said to be faithful if it preserves all safety and liveness properties of the original protocol system concerning the projected function. An image protocol is smaller than the original protocol and can typically be more easily analyzed. Two protocol examples are employed herein to illustrate our method. An application of this method to verify a version of the high-level data link control (HDLC) protocol is described in a companion paper.

[Lamb78] Abstract: Increasing the communication between customer and contractor is seen as an effective way of improving software quality. Experience with a variety of software methods has clearly focused on both the need for communication and the means of accomplishing it. A methodology for defining software requirements and design has been developed. It is based, in part, on a synergism of modeling techniques. Experiences with the methodology have resulted in refinements. Elements of the methodology, experience with it, and current applications aimed at automating its use are described.

[Lamp77] Abstract: The inductive assertion method is generalized to permit formal, machine-verifiable proofs of correctness for multiprocess programs. Individual processes are represented by ordinary flowcharts, and no special synchronization mechanisms are assumed, so the method can be applied to a large class of multiprocess programs. A correctness proof can be designed together with the program by hierarchical process of stepwise refinement, making the method practical for larger programs. The resulting proofs tend to be natural formalization of the informal proofs that are now used.

[Lamp78] Abstract: The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

[Lamp79a] Abstract: A nonassertional approach to proving multiprocess correctness is described by proving the correctness of a new algorithm to solve the mutual exclusion problem. The algorithm is an improved version of the bakery exclusion algorithm. It is specified and proved correct without being decomposed into indivisible atomic operations. This allows two different implementations for a conventional, nondistributed system. Moreover, the approach provides a sufficiently general specification of the algorithm to allow nontrivial implementations for a distributed system as well.

[Lamp79b] Abstract: A formal specification is given for a simple calendar program, and the derivation and proof of correctness of the program are sketched. The specification is easy to understand, and its correctness is manifest to humans.

[Lamp80] Abstract: Hoare's logical system for specifying and proving partial correctness properties of sequential programs is generalized to concurrent programs. The basic idea is to define the assertion (P)S(Q) to mean that if execution is begun anywhere in S with P true, then P will remain true until S terminates, and Q will be true if and when S terminates. The predicates P and Q may depend upon program control locations as well as upon the values of variables. A system of inference rules and axiom schemas is given, and a formal correctness proof for a simple program is outlined. We show that by specifying certain requirements for the unimplemented parts, correctness properties can be proved without completely implementing the program. The relation to Pnueli's temporal logic formalism is also discussed.

[Lamp82] Abstract: Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgetable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

[Lamp83] Abstract: A method for specifying program modules in a concurrent program is described. It is based upon temporal logic, but uses new kinds of temporal assertions to make the specifications simpler and easier to understand. The semantics of the specifications is described informally, and a sequence of examples are given culminating in a specification of three modules comprising the alternating-bit communication protocol. A formal semantics is given in the appendix.

[Lamp84] Abstract: Generalized Hoare Logic is a formal logical system for deriving invariance properties of programs. It provides a uniform way to describe a variety of methods for reasoning about concurrent programs, including noninterference, satisfaction, and cooperation proofs. We describe a simple meta-rule of the Generalized Hoare Logic - the Decomposition Principle - and show how all these methods can be derived using it.

[Land77] Abstract: Modeling of systems featuring hardware and software faults is studied as a means of evaluating the availability and reliability characteristics. The case of a non-redundant computer is studied and it is shown that the unavailability presents an overshoot with respect to its asymptotic value whose height and length are functions of the failure rates associated with the different design errors. Also, a fault-tolerant system is studied that includes protective redundancies at the hardware and software levels.

[Land86] Abbreviated Introduction: The Naval Research Laboratory sponsored this workshop to invigorate research in both program verification and program testing through cross-fertilization, to document the state of the art and practice in both areas, and to identify current assurance requirements and techniques for meeting them. Tutorials characterizing the current state of testing and proving techniques and identifying industry and

government assurance requirements provided a common basis for five discussion groups. These groups addressed (1) the role of specifications in testing and proving, (2) hybrid approaches of testing and proving, (3) levels of assurance, (4) interactions between testing/proving and software engineering, and (5) cost effectiveness.

[Lapr84] Abstract: This paper deals with the evaluation of the dependability (considered as a generic term, whose main measures are reliability, availability, and maintainability) of software systems during their operational life, in contrast to most of the work performed up to now, devoted mainly to development and validation phases.

The failure process due to design faults, and the behavior of a software system up to the first failure and during its life cycle are successively examined. An approximate model is derived which enables one to account for the failures due to the design faults in a simple way when evaluating a system's dependability. This model is then used for evaluating the dependability of 1) a software system tolerating design faults, and 2) a computing system with respect to physical and design faults.

[Lask79] Summary: A real-environment interactive testing procedure is presented which is based upon a hierarchical decomposition of a program into levels of abstraction. Such a decomposition is defined in terms of a program model which involves both control and data flow. The testing strategy adopted is supposed to follow a typical progress of a programmer which carries out a series of experiments with his program. Several semantical and structural issues involved are discussed.

[Lask82] Abstract: A structural approach to testing employing properties of data flow in a program is proposed. The basic notion introduced is that of data context of a program block. It represents the set of all tuples of definitions of the block arguments that are simultaneously live when the control reaches the block. Two testing strategies have been proposed: block testing, exercising every block of all its elementary contexts and d-tree testing exercising the definition tree rooted at an elementary contest of the *stop/exit instruction*.

[Lask83] Abstract: Some properties of a program's data flow can be used to guide program testing. The presented approach aims to exercise use-definition chains that appear in the program. Two such data oriented testing strategies are proposed; the first involves checking liveness of every definition of a variable at the point(s) of its possible use; the second deals with liveness of vectors of variables treated as arguments to an instruction or program block. Reliability of these strategies is discussed with respect to a program containing an error.

[Lask86] Abstract: A codefinition is a set of definitions in the program that simultaneously reach an instruction in it. Codefinitions are used in data-based program testing. An intraprocedural iterative algorithm for the derivation of codefinitions is presented. It has been applied in a data-based testing tool recently implemented.

[Lask88a] Abstract: A program design methodology is presented that advocates the synthesis of tests hand-in-hand with the design at every stage of program development and uses them for early detection of design flaws. It involves formal specifications of abstract programs and abstract data refinement that appear in the design. Main findings: 1) Formalization facilitates black-box and design-based functional testing, 2) Abstract data testing allows a more natural selection of tests than concrete data testing, 3) Black-box testing leads to significant structural coverage, 4) The method can be combined with formal verification.

[Lass79] Abstract: Several examples of simple program schemes are used to study the influence of basic constructs on measures of Software Science. A minimal low level language is used in order to build examples which contain large numbers of the constructs under study. The measures are expressed as functions depending on the number of conceptually unique input-output operands. They may therefore be evaluated and compared to their estimators analytically rather than statistically.

[Lass81] Abstract: The claims that software science could provide an empirical basis for the rationalization of all forms of algorithm description are shown to be invalid from a formal point of view. In particular, the

conjectured dichotomy between operators and operands is shown not to hold over a wide class of languages. An experiment that investigated discrepancies between the level measure and its estimator is described to show that its failure was due to shortcomings in the theory. One cannot obtain reliable results without tampering with both measure and estimator definitions.

[Laue79] **Summary:** Debugging is efficient if it detects all program errors in a short time. This paper discusses several techniques for improving debugging efficiency. Attention is given both to the initial debugging and to acceptance testing in the maintenance stage. A main decision is whether to use top-down or bottom-up debugging, and it is suggested that top-down debugging is more efficient if combined with some of the other techniques. All the techniques shown are independent of any particular language or debug software.

[Lave88] **Abstract:** This paper is a discussion of issues related to the thesis entitled "The Explication of Process-Product Relationships in DoD-STD-2167 and DoD-STD-2168 via an Augmented Data Flow Diagram Model."

In particular, the major results of the above thesis are viewed in light of the draft standards DoD-STD-2167A and DoD-STD-2168 (both dated 1 April 1987), and the issue of *development objectives* is explored.

The ideas presented in this paper represent the author's opinion and are speculative in nature due to the fact that, at present, the revised DoD standards are in draft form, and the issue of development objectives has not yet been thoroughly investigated.

[Lawr81] **Abstract:** Programming data involving 278 commercial-type programs were collected from 23 medium-to-large-scale organizations in order to explore the relationships among variables measuring program type, the testing interface, programming technique, programmer experience, and productivity. Programming technique and programmer experience after 1 year were found to have no impact on productivity, whereas on-line testing was found to reduce productivity. A number of analyses of the data are presented, and their relationship to other studies is discussed.

[LeDo85] **Abstract:** A trace database model for debugging concurrent Ada programs is presented. In this approach, trace information is captured in an historical database and queried using Prolog. This model was used to build a prototype debugger, called Your Own Ada Debugger (YODA). The design of YODA is described and a trace analysis of a sample program exhibiting misuse of shared data is presented. Because the trace database model is flexible and general, it can aid diagnosis of a variety of runtime errors.

[Leac87] **Abstract:** A major goal of software engineering research is the development of metrics which measure the complexity and maintainability of programs, with a small portion of this effort directed specifically towards programs written in Ada. This paper will focus on two main themes. The first theme will be the development of metrics that specifically reflect the complexity of programs in Ada. The second theme will be an investigation of the theoretical limits of metrics as measures of program complexity in general.

[Lee88] **Abstract:** Software creation requires not only testing during the development cycle by the development staff, but also independent testing following the completion of the implementation. However in the latter case, the amount of testing that can be carried out is often limited by time and resources. At the very most, independent testing can be expected to provide 100% test coverage of the test requirements (or specifications) associated with the software element with the minimum of effort. This paper describes a methodology employing *Integer Programming* by which the amount of testing required to provide the maximum possible test coverage of the requirements (for the given test set) is assured while at the same time minimizing the total number of tests to be included in a test suite. A collateral procedure provides recommendations on which tests might be eliminated if less than 100% test coverage of the test requirements is permitted. This latter procedure will be useful in determining the risk of not running the minimum set of tests for 100% test coverage. A third process selects from the test matrix the set of tests to be applied to the system following maintenance modifications of any test requirements – that is, to provide a submatrix for regression testing. The potential benefits for applying the integer programming technique in test data selection is also discussed.

[Less81] Abstract: A new approach for structuring distributed processing systems, called functionally accurate, cooperative (FA/C), is proposed. The approach differs from conventional ones in its emphasis on handling distribution-caused uncertainty and errors as an integral part of the network problem-solving process. In this approach nodes cooperatively problem-solve by exchanging partial tentative results (at various levels of abstraction) within the context of common goals. The approach is especially suited to applications in which the data necessary to achieve a solution cannot be partitioned in such a way that a node can complete a task without seeing the intermediate state of task processing at other nodes. Much of the inspiration for the FA/C approach comes from the mechanisms used in knowledge-based artificial intelligence (AI) systems for resolving uncertainty caused by noisy input data and the use of approximate knowledge. The appropriateness of the FA/C approach is explored in three application domains: distributed interpretation, distributed network traffic-light control, and distributed planning. Additionally, the relationship between the approach and the structure of mangement organizations is developed. Finally, a number of current research directions necessary to more fully develop the FA/C approach are outlined. These research directions include distributed search, the integration of implicit and explicit forms of control, and distributed planning and organizational self-design.

[Leun88] Abstract: Regression testing is a significant, but largely unexplored topic. In this report, the problem of regression testing is analysed, and several important notions are introduced: the types of regression testing, the test case classification according to changes and the *regression number*. Regression testing can be grouped into *corrective regression testing* and *progressive regression testing*, depending on the stability of the specification. The test cases can be grouped into five classes: *reusable, testable, obsolete, changed* and *new* test cases. A problem facing all retesters is the proper identification of test classes. The notion of *regression number* is introduced as a measure of the number of test cases affected by a single instruction change. A program component called a *retestable unit* is proposed to encapsulate the effect of changes on a program. The use of retestable unit may reduce the effort in test selection for regression testing. An algorithm for computing a retestable unit is given, and a preliminary experiment on retestable unit is reported. The regression testing problem can be decomposed into two subproblems: the *test selection problem* and the *test plan update problem*. This report presents a solution to the test plan update problem, which involves the use of a unique data structure for storing program information during testing. The data structure allows an easy manipulation of these information for the purpose of classifying the test cases.

[Leve83b] Abstract: With the increased use of software controls in critical real-time applications, a new dimension has been introduced into software reliability - the "cost" of errors. The problems of safety have become critical as these applications have increasingly included areas where the consequences of failure are serious and may involve grave dangers to human life and property. This paper defines software safety and describes a technique called software fault tree analysis which can be used to analyze a design as to its safety. The technique has been applied to a program which controls the flight and telemetry for a University of California spacecraft. A critical failure scenario was detected by the technique which had not been revealed during substantial testing of the program. Parts of this analysis are presented as an example of the use of the technique and the results are discussed.

[Leve83c] Abstract: Software is increasingly being used in the control of potentially hazardous systems. Software fault-tree analysis is a technique for analyzing the logic of software for any potential contribution to system mishaps. The technique is described using Ada as an example real-time language. Special consideration is given to the problems of concurrency and real-time constraints which are common in these types of applications.

[Leve86b] Abstract: Software safety issues become important when computers are used to control real-time, safety-critical processes. This survey attempts to explain why there is a problem, what the problem is, and what is known about how to solve it. Since this is a relatively new software research area, emphasis is placed on delineating the outstanding issues and research topics.

[Leve87] Abstract: The application of Time Petri net modeling and analysis techniques to safety-critical real-time systems is explored and procedures described which allow analysis of safety, recoverability, and

fault-tolerance.

[Levi78] Panel Overview: Formal methods, i.e., use of mathematical rigor, have been employed by research computer scientists in their attempt to develop general results for many aspects of computer science, e.g., computational complexity, undecidability, numerical analysis, programming language semantics. Much of this work has had little impact on those charged with producing working software systems. However, in recent years numerous researchers have suggested that by applying formal methods to the realization of systems, the quality of such systems could be significantly improved. Such formal methods could be applied in the structuring, specification, verification, and analysis of performance for systems. The position statements below explore the use of these techniques in the production of systems. The general opinion is that formal methods will ultimately assume a vital role, but for the present their use will be restricted to particular systems produced by skilled individuals. The use of the formal methods will gradually increase as the techniques are refined and applied to a larger variety of systems, as tools are developed to support their use, and as the general community becomes better educated in formal methods.

[Levi80] Abstract: This thesis presents proof rules for an extension of Hoar's Communicating Sequential Processes (CSP). CSP is a notation for describing processes that interact through communication, which provides the sole means of synchronizing and passing information between processes. A sending process is delayed until some process is ready to receive the message; a receiving process is delayed until there is a message to be received. It is this delay that provides synchronization.

A proof of a program is with respect to pre- and post-conditions. A proof of weak correctness shows that execution of the program beginning in a state satisfying the pre-condition terminates in a state satisfying the post-condition, providing deadlock does not occur. A proof of strong correctness, in addition, shows that deadlock cannot occur.

A proof of weak correctness has three stages: a sequential proof, a satisfaction proof, and a non-interference proof. A sequential proof reflects the efforts of a process running in isolation. A satisfaction proof combines sequential proofs of processes, reflecting the message passing and synchronization aspects of communication. A non-interference proof shows that no process affects the validity of the proof of another process.

The introduction of the satisfaction proof and our symmetric treatment of send and receive are important aspects of this thesis. By treating send and receive on an equal basis, we simplify our rules and allow the inclusion of send in guards.

A sufficient condition for freedom from deadlock is given that depends on the proof of weak correctness; this is used to prove strong correctness. In general, freedom from deadlock can be very hard to check. Therefore, we derive special cases in which we can reduce the work needed to verify that a program is free from deadlock.

We also present an algorithm for globally synchronizing processes; that is, each process can recognize that all processes are simultaneously in a given state. It works by recognizing a special class of deadlock. Having this algorithm allows us to modify programs that deadlock when the post-condition is established, so that they terminate normally.

[Levi81] Abstract: Proof rules are presented for an extension of Hoare's communicating sequential processes. The rules deal with total correctness; all programs terminate in the absence of deadlock. The commands send and receive are treated symmetrically, simplifying the rules and allowing send to appear in guards. Also given are sufficient conditions for showing that a program is deadlock free. An extended example illustrates the use of the technique.

[Levy84] Abstract: A strategy for performing type checking on programs built out of separately compiled parts is presented. This strategy is used in a programming environment that allows small components of a software system to be reconfigured in different ways. The strategy works by inferring type schemas for all of the undeclared functions used by a component and then unifying each schema with a program library when a configuration is built.

[Lew88] Abstract: The complexity of software often affects its reliability. In order to produce reliable software, its complexity must be controlled by suitably decomposing the software system into smaller subsystems. In this paper, a software complexity metric is developed which includes both the internal and external complexity of a module. This allows analysis of a software system during its development and provides a guide to system decomposition. The basis of this complexity metric is in the development of an external complexity measure which characterizes module interaction.

[Li87] Abstract: Software metrics are computed for the purpose of evaluating certain characteristics of the software developed. A Fortran static source code analyzer, FORTRANL, was developed to study 31 metrics, including a new hybrid metric introduced in this paper, and applied to a database of 255 programs, all of which were student assignments. Comparisons among these metrics are performed. Their cross-correlation confirms the internal consistency of some of these metrics which belong to the same class. To remedy the incompleteness of most of these metrics, the proposed metric incorporates context sensitivity to structural attributes extracted from a flow graph. It is also concluded that many volume metrics have similar performance while some control metrics surprisingly correlate well with typical volume metrics in the test samples used. A flexible class of hybrid metric can incorporate both volume and control attributes in assessing software complexity.

[Lind85] Abbreviated Introduction: This paper describes a specification and validation method that allows validation tests to be generated in a White-Box fashion and administered in a Black-Box fashion. The paper presents a specification technique for CAIS that uses an Ada-based description of CAIS facilities. This Abstract Machine approach to specifying CAIS is summarized. The paper addresses a two-phase approach to developing validation test from The Abstract description of CAIS. In the first phase, existing testing technology is applied to isolate needed test data points in terms of the inputs and expected outputs. The second phase converts the test data points, using further analysis of the specification, into Ada tools that do not rely on data internal to the Abstract description.

[Lind88a] Abstract: This analysis tool produces I/O pairs that represent program execution paths. You can use these pairs as hurdles for program testing and interface validation to overcome.

[Lind88d] Abbreviated Introduction: As the title suggests, this paper is a survey of computer support for formal reasoning, but primarily from the point of view of software engineering applications. It makes no claim to being an objective comparison of theorem proving systems *per se*, nor does it claim to present all the features of the various systems. Instead, it is intended to be an introduction to existing systems and ongoing research, gathering together information which has often only appeared before in narrowly distributed technical reports.

[Lind89] Abstract: In this paper, we report the results of an experimental study of software metrics for a fairly large software system used in a real-time application. We examine a number of issues, including the mutual relationship between various software metrics and, more importantly, the relationship between metrics and the development effort. We report some interesting connections between metrics and the software development effort.

[Ling79] Table of Contents: Precision Programming. Elements of logical expression. Elements of program expression, syntax control structures, syntax data structures, syntax system structures, structured programs, program execution, program functions, program structures. Reading structured programs. The correctness of structured programs, verifying structured programs, correctness of prime programs, techniques for proving program correctness, examples, loop invariants in correctness proofs, formulas for correct structured programs. Writing structured programs.

[Linn88] Abstract: IDA Paper P-2035 presents the SDI Architecture Dataflow Modeling Technique (SADMT), a uniform formal notation for the description of SDI system architectures and the Battle Management and Command, Control, and Communication (BM/C3) architectures. SADMT is a technique for thinking about and

describing architectural processes and structures that use the typing and functional facilities of the Ada programming language. The document defines SADMT and the programming interface to the SADMT Simulation Facility (SADMT/SF). The issues addressed here are those relevant to providing formal descriptions of system, structure and behavior for interface consistency checking, system simulation, and system evaluation.

[Lisk75] Abstract: The main purposes in writing this paper are to discuss the importance of formal specifications and to survey a number of promising specification techniques. The role of formal specifications both in proofs of program correctness, and in programming methodologies leading to programs which are correct by construction, is explained. Some criteria are established for evaluating the practical potential of specification techniques. The importance of providing specifications at the right level of abstraction is discussed, and a particularly interesting class of specification techniques, those used to construct specifications of data abstractions, is identified. A number of specification techniques for describing data abstractions are surveyed and evaluated with respect to the criteria. Finally, directions for future research are indicated.

[Lite76] Abstract: The paper provide data on Cobol error frequency for correction of errors in student-oriented compilers, improvement of teaching, and changes in programming language. Cobol was studied because of economic importance, widespread usage, possible error-inducing design, and lack of research. The types of errors were identified in a pilot study; then, using the 132 error types found, 1,777 errors were classified in 1,400 runs of 73 Cobol students. Error density was high: 20 percent of the types contained 80 percent of the total frequency, which implies high potential effectiveness for software-based correction of Cobol. Surprisingly, only four high-frequency errors were error-prone, which implies minimal error inducing design. 80 percent of Cobol misspellings were classifiable in the four error categories of previous researchers, which implies that Cobol misspellings are correctable by existent algorithms. Reserved word usage was not error-prone, which implies minimal interference with usage of reserved words. Over 80 percent of error diagnosis was found to be inaccurate. Such feedback is not optimal for users, particularly for the learning user of Cobol.

[Litt73] Summary: A Bayesian reliability growth model is presented which includes special features designed to reproduce special properties of the growth in reliability of an item of computer software (program). The model treats the situation where the program is sufficiently complete to work for continuous time periods between failures, and gives a repair rule for the action of the programmer at such failures. Analysis is based entirely upon the length of the periods of working between repairs and failures, and does not attempt to take account of the internal structure of the program. Methods of inference about the parameters of the model are discussed.

[Litt75] Abstract: A system is considered in which switching takes place between sub-systems according to a continuous parameter Markov chain. Failures may occur in Poisson processes in the sub-systems, and in the transitions between sub-systems. All failure processes are independent. The overall failure process is described exactly and asymptotically for highly reliable sub-systems. An application to process-control computer software is suggested.

[Litt78] Abstract: This paper examines critically, with a view to stimulating a discussion, some concepts which have been used in early work on software reliability measurement, and suggests improvements and areas of potentially fruitful future research. It is proposed that hardware-motivated measures such as mttf, mtbf should not be used for software without justification, and it is shown that such justification may be lacking under quite unexceptionable circumstances. Alternative methods of measuring software reliability are proposed. Emphasis is placed upon differentiating between two concepts of software reliability which are often blurred in the work of previous authors. These are, on the one hand, the reliability of the program-as-it-is (the number of bugs it contains), on the other, the reliability of the program-as-it-performs (failure rate, distribution of time to next failure, etc.). It is argued that the latter, here called *operations reliability*, is the one we should use. Measures of operational reliability which avoid use of mttf, etc., are proposed. A case is made for software engineers adopting a Bayesian stand-point: both in the interpretation of probability statements and in inference procedures. It is suggested that reliability modeling solely in terms of failures (or number of bugs) is unnecessarily naive. Interest

really centers upon the *consequences* of failures as much as on their frequency. It is proposed that more effort be devoted to the development of models which incorporate a cost (or utility) structure. Finally, brief consideration is given to the question of program structure. The enormous success of hardware reliability theory, in combining component reliabilities with knowledge of system structure, must be emulated for software. Unfortunately, software structure does not easily lend itself to such an exercise. Some existing models are considered.

[Litt79] Abstract: The paper treats a modular program in which transfers of control between modules follow a semi-Markov process. Each module is failure prone, and the different failure processes are assumed to be Poisson. The transfers of control between modules (interfaces) are themselves subject to failure. The overall failure process of the program is described, and an asymptotic Poisson approximation is given for the case when the individual modules and interfaces are very reliable. A simple formula gives the failure rate of the overall programs (and hence mean time between failures) under this limiting condition. The remainder of the paper treats the consequences of failures. Each failure results in a cost, represented by a random variable with a distribution typical of the type of failure. The quantity of interest is the total cost of running the program for a time t, and a simple approximation distribution is given for large t. The parameters of this limiting distribution are functions only of the means and variances of the underlying distributions, and thus are readily estimable. A calculation of program availability is given as an example of the cost process. There follows a brief discussion of methods of estimating the parameters of the model, with suggestions of areas in which it might be useful.

[Litt80a] Introduction: It is instructive to look at some of the reasons advanced by software developers for their reluctance to use software reliability measurement tools. Here are a few common ones:

1. "Software reliability models are statistical. Programs are deterministic. If certain input conditions cause a malfunction today, then the same conditions are certain to cause a malfunction if they occur tomorrow. Where is the randomness?"
2. "I am paid to write reliable programs. I use the best programming methodology to achieve this. Software reliability estimation procedures would not help me to improve the reliability of my programs."
3. "We verify our software. When it leaves us it is correct."
4. "I ran your software reliability measurement program on some data from a current project of ours. It said there was an infinite number of bugs left in the program. Who are you trying to kid?"
5. (same manager as D, but one week later) "We corrected a couple of bugs and ran the reliability measurement program again. This time it said that there were over 200 bugs left. Infinity minus two equals two hundred? Is this the new math?"
6. "We put a lot of effort into testing. The selection of test data is a systematic process designed to seek out bugs. Reliability estimation based on such test data would be no guide to the performance of the program in a use environment."
7. "We are writing an air traffic control program. Total system crash would be catastrophic. Other failures range from serious to trivial. Reliability models do not distinguish between failures of differing severity."

Although [the author has] been involved in software reliability modeling for the past decade, and [has himself] perpetrated a few models, [he has] a great deal of sympathy with some of the sentiments expressed above. [The author has] a feeling that some of the early models have been oversold, that not enough emphasis has been placed on the underlying modeling assumptions, and that by concentrating on a simple reliability analysis we might be ignoring wider concerns. In this paper [the author] shall be looking at one common deficiency of early models and suggesting a way in which it can be overcome. [The author hopes] that, in passing, some new insight into the wider issues will be gained.

[Litt80b] Abstract: An examination of the assumptions used in early bug counting models of software reliability shows them to be deficient. Suggestions are made to improve modeling assumptions and examples are given of mathematical implementations. Model verification via real-life data is discussed and minimum requirements are presented. An example shows how these requirements may be satisfied in practice. It is suggested that current theories are only the first step along what threatens to be a long road.

[Litt81a] Abstract: An assumption commonly made in early models of software reliability is that the failure rate of a program is a constant multiple of the (unknown) number of faults remaining. This implies that all faults contribute the same amount to the overall failure rate of the program. The assumption is challenged and an alternative proposed. The suggested model results in earlier fault fixes having a greater effect than later ones (the faults which make the greatest contribution to the overall failure rate tend to show themselves earlier, and so are fixed earlier), and the DFR property between fault fixes (assurance about programs increases during periods of failure-free operation, as well as at fault fixes). The model is tractable and allows a variety of reliability measures to be calculated. Predictions of total execution time to achieve a target reliability, and total number of fault fixes to target reliability, are obtained. The model might also apply to hardware reliability growth resulting from the elimination of design errors.

[Lohs84] Overview: The importance of the scientific investigations of software design principles is discussed, and an experimental investigation of the importance of the design principle of module coupling is described. One important dimension of coupling, as promoted by the authors of the structured design methodology, is that of global variable vs. parameterized methods of intermodule communications. It is shown that different proposed software metrics provide conflicting conclusions as to the preferred method of intermodule communication. The three experiments reported herein were performed in university software engineering courses taken by graduate students and upper level undergraduate majors in computer science. They address the effect of global vs. parameterized interfaces on system modifiability. While the type of modification being performed significantly influenced the modifiability of the system, there were no consistent effects due to the type of coupling present in the system.

[Lond75] Abstract: One person's perspective of program verification and its relation to some aspects of reliable software are presented. The main verification method of inductive assertions is illustrated with several variations of one detailed example; a second example shows a surprisingly simple inductive assertion proof of an iterative tree traversal example. Briefly discussed also are the implicit assumptions of most verifications, proving termination, the creating of assertions, and languages in which to write assertions. An abstract overview is given of existing program verification systems together with a sample list of verified programs. A short bibliography is included.

[Lond85] Abbreviated Introduction: The availability of today's powerful personal workstations with high-resolution bit-map displays and pointing devices makes possible the creation and display of drawings containing a wide assortment of characters, fonts, icons, and figures, all of which can be continuously moved for realistic animation. We are currently involved in using such animation to visualize programs and algorithms by creating graphical snapshots and movies correlated with the programs' actions. Such a facility we hope will provide programmers or computer users in general with an understanding of what the programs do, how they work, and why they work. It also will give users visual feedback as a program and its parts are being executed. This animation system will provide pictorial representations of those data structures, at the proper level of abstraction, which are used by a program. Standard representations of internal data structures, such as linked lists or arrays with separate index variables, are often insufficient because the viewer must mentally transcribe such representations to the abstractions involved in the use of those structures. We use the type of diagrams or sketches a programmer draws at a desk or wallboard, or the kinds of schematic figures found in a programming or data structures text; fortunately, we do not need pictures with exquisite shadings that re-create photographs. Such figures change to reflect the changes during the execution of the program. People's apparent tendency to understand by visualizing spatially the abstractions that constitute the intention, or "meaning," of a program is exploited by the system.

[Long77] Abstract: The power industry is becoming increasingly interested in the use of digital computers within nuclear plant protection of systems in order to satisfy increased safety requirements, provide greater operating flexibility, minimize spurious forced outages, and (in conjunction with multiplexing) to meet separation requirements. However, the development and licensing of these digital safety systems has been hindered to date by the difficulty of validing software.

This paper reviews the rationale for safety system software validation requirements. A survey of current methodologies for the development of software for nuclear power plant safety protection system and their associated limitations are provided.

A methodology is then proposed for the development and validation of nuclear power plant safety system software which may permit a quantitative assessment of its correctness. The main features of the methodology are: 1) formal specification and documentation procedures coupled with strict software development restrictions, and 2) comprehensive testing and program analysis used in conjunction with symbolic execution and theorem proving techniques to establish correctness. Adoption of multiple specification and dual programming teams in this methodology introduces a redundancy for easy detection of major design and programming errors. The latter also significantly reduces the amount of testing effort.

[Long88] Abbreviation: [This paper] presents a representation for concurrent systems, called a *task interaction graph*, that facilitates analysis [of the reliability of concurrent systems]. Our representation is an extension of the work of Taylor. We have been developing a model of interacting tasks that may considerably reduce the number of states in concurrency graph representation. We call this representation a *Task Interaction Concurrency Graph* (TICG), since it is derived from a *Task Interaction Graph* (TIG) instead of a control flow representation.

The TICG and TIG models have been designed to capture the rendezvous-like synchronization found in languages like Ada, Distributed Processes, and CSP. Task interaction graphs represent tasks as sets of regions and interactions between regions. To date we have developed rules for translating most of the constructs supported by Ada into the appropriate TIG representation. We have been investigating several kinds of analysis techniques that can be applied to the TIG and TICG models. Deadlock detection and dangerous parallelism are just two examples of the kinds of analysis that can be performed using a TICG representation. We are particularly interested in investigating the extension of error sensitive testing techniques to concurrent, real-time systems.

[Love76] Abstract: Recent work in the field of Software Physics has produced several hypotheses relating the nature of algorithms to measurable properties on computer programs. One hypothesis is that Halstead's measure of E, the number of elementary mental discriminations required to implement an algorithm is strongly related to measurable properties of computer programs. Several experiments have shown a surprising high correlation between E and such measurable properties of programs as number of bugs, coding times, etc. This paper will present the results of an independent study to test this hypothesis.

[Love77b] Abstract: A within-subjects experimental design was used to test the effect of two variables on program understanding. The independent variables were complexity of control flow and paragraphing of the source code. Understanding was measured by having the subjects memorize the code for a fixed time and reconstruct the code verbatim. Also some subjects were asked to describe the function of the program after completing their reconstruction. The two groups of subjects for the experiment were students from an introductory programming class and from a graduate class in programming languages.

The major findings were that paragraphing of the source had no effect for either group of subjects but that programs with simplified control flow were easier for the computer science students to understand as measured by their ability to reconstruct the programs. The dependent variable, rated accuracy of their description of the programs functions, did not differ as a function of either independent variable.

The paper is concluded with a description of the utility of this experimental approach relative to improving the reliability of software and a discussion of the importance of these findings.

[Luck77] Abstract: Emphasis is placed on the practical problems encountered in designing automatic program verifiers and using them as an aid to programming. The paper includes an on-line interactive demonstration of a verifier and a short survey of the kinds of programs that have been verified so far.

[Luck79a] Abstract: A practical method is presented for automating in a uniform way the verification of Pascal programs that operate on the standard Pascal data structures Array, Record, and Pointer. New assertion

language primitives are introduced for describing computational effects of operations on these data structures. Axioms defining the semantics of the new primitives are given. Proof rules for standard Pascal operations on data structures are then defined using the extended assertion language. An axiomatic rule for the Pascal storage allocation operation, NEW, is also given. These rules have been implemented in the Stanford Pascal program verifier. Examples illustrating the verification of programs which operate on list structures implemented with pointers and records are discussed. These include programs with side effects.

[Luck80a] Abstract: We present a method of formal specification of Ada programs containing packages. The method suggests concepts and guidelines useful for giving adequate informal documentation of packages by means of comments.

The method for depends on (1) the standard inductive assertion technique for subprograms, (2) the use of history sequences in assertions specifying the declaration and use of packages, and (3) the addition of three categories of specifications to Ada package declarations: (a) visible specifications, (b) boundary specifications, (c) internal specifications.

Axioms and proof rules for the Ada package constructs (declaration, instantiation, and function and procedure call) are given in terms of history sequence and package specifications. These enable us to construct formal proofs of the correctness of Ada programs with packages. The axioms and proof rules are easy to implement in automated program checking systems. The use of history sequences in both informal documentation and formal specifications and proofs is illustrated by examples.

[Luck80b] Abstract: A method of documenting exception propagation and handling in Ada programs is proposed. Exception propagation declarations are introduced as a new component of Ada specifications, permitting documentation of those exceptions that can be propagated by a subprogram. Exception handlers are documented by entry assertions. Axioms and proof rules for Ada exceptions are given. These rules are simple extensions of previous rules for Pascal and define an axiomatic semantics of Ada exceptions. As a result, Ada programs specified according to the method can be analyzed by formal proof techniques for consistency with their specifications, even if they employ exception propagation and handling to achieve required results (i.e., nonerror situations). Example verifications are given.

[Luck84a] Abstract: A specification language permits information about various aspects of a program to be expressed in a precise machine processable form. This information is not normally part of the program itself.

Specification languages are viewed as evolving from modern high level programming languages. This first step in this evolution is cautious extensions of the programming language. Some of the features of Anna, a specification language extending Ada, are discussed. The extensions include generalizations of constructs (such as type constraints) that are already in Ada, and new constructs for specifying subprograms, packages, exceptions, and contexts.

Anna has been designed in collaboration with B. Krieg-Brueckner and O. Owe.

[Luck85] Abbreviated Introduction: ANNA is a proposal for a specification language, or rather a language in which one might experiment with specification languages. The work was begun by Bernd Krieg-Brueckner and myself, and subsequent collaborators have been O. Owe from Oslo, who worked on the axiomatic semantics, and Friedrich von Henke who worked on the language reference manual and redesigned some of the finer points of the language. S. Sankar, D. Rosenblum, R. Neff, and D. Bryan are currently implementing various prototype tools for experimentation.

ANNA is an syntactic extension of ADA: it takes a subset of ADA productions and adds more. The ANNA specifications appear as formal ADA comments. This means ANNA comments can be processed by a standard ADA tool, which will simply ignore them, and also by special ANNA tools.

All proposed ANNA tools use an extension of DIANA, and therefore can be interfaced easily with other tools in an ADA environment.

ANNA can be used for comparative testing. Comparative testing means comparing the ADA code against its formal specifications for consistency. Self-checking programs are ones which leave the runtime checks

compiled from the formal specifications in the program permanently.

[Luck86a] Abstract: This report gives an overview of the current status and plans to construct a prototype environment of advanced tools for software and hardware development based on the use of wide-spectrum languages. The wide-spectrum languages include Anna (ANNotated Ada), and TSL (Task Sequencing Language). The tools described here provide interactive aid at all stages in the system development process. Special emphasis is placed on distributed computing, both in providing tools for handling parallelism in the subject system, and in designing tools that utilize parallelism in the programming environment. Applications of these tools include requirements analysis, formal specification, rapid prototyping, testing, formal verification and construction of self-testing Ada software for multi-processor systems.

The report describes an existing environment of prototype tools supporting applications of Anna and TSL to formal specification and testing of Ada software. The new environment tools will be based on component tools already developed at Stanford and proven to be portable to various Ada environments. All tools are implemented in Ada and are intended to interface with standard components of Ada programming environments.

[Luck87] Abstract: TSL-1 is a language for specifying sequences of tasking events occurring in the execution of distributed Ada programs. Such specifications are intended primarily for testing and debugging of Ada tasking programs, although they can also be applied in designing programs. TSL-1 specifications are included in an Ada program as formal comments. They express constraints to be satisfied by the sequences of actual tasking events. An Ada program is consistent with its TSL-1 specifications if its runtime behavior always satisfies them. This paper presents an overview of TSL-1. The features of the language are described informally, and examples illustrating the use of TSL-1, both for debugging and specification of tasking programs, are given. A definition of robust TSL-1 specifications that takes into account uncertainty in runtime observation of behavior of distributed systems is given. A runtime monitor for checking consistency of an Ada program with TSL-1 specifications has been implemented. In the future, constructs for defining abstract tasks will be added to TSL-1, forming a new language, TSL-2, for the specification of distributed systems prior to their implementation in any particular programming language.

[MIL85] Scope: This standard prescribes the requirements for the conduct of Technical Reviews and Audits on Systems, Equipments, and Computer Software.

The following technical reviews and audits shall be selected by the program manager at the appropriate phase of program development. Each review/audit is generally described in Section 3, Definitions, and more specifically defined in a separate appendix.

System Requirements Review (SRR)
System Design Review (SDR)
Software Specification Review (SSR)
Preliminary Design Review (PDR)
Critical Design Review (CDR)
Test Readiness Review (TRR)
Functional Configuration Audit (FCA)
Physical Configuration Audit (PCA)
Formal Qualification Review (FQR)
Production Readiness Review (PRR)

Technical Reviews and Audits defined herein shall be conducted in accordance with this standard to the extent specified in the contract clauses, Statement of Work (SOW), and the Contract Data Requirements List. Guidance in applying this standard is provided in Appendix J. The contracting agency shall tailor this standard to require only what is needed for each individual acquisition.

[Majo83] Abstract: In this paper an automated method for testing programs against a formal specification is presented. The method is based on the view of a software system as a network of modules and data capsules which are connected via data flows. Modules are specified in terms of their pre- and postconditions, data

capsules are specified in terms of their usage as input and output. For this purpose a special assertion language is employed. The test procedures derived from the language serve to simulate data capsules and modules when testing, generating arguments and validating results.

[Mand85] Abstract: Multitasking is one of the most novel aspects of Ada. However, the combination of language primitives for concurrent execution of tasks, synchronization, termination, abortion, exception handling, etc. make Ada programs difficult to understand and analyze. This is partly due to the inherent complexity of the language and partly due to the lack of a rigorous definition of its semantics. The Ada Reference Manual describes semantics in informal English prose; as a result, it is often verbose and ambiguous.

The goal of this paper is not to provide a complete formal semantics of Ada multitasking. Rather, we illustrate the use of a semi-formal approach based on (timed) Petri nets which support a rigorous description of the language. The approach is described by stepwise refinements and is used to describe several cases of task interactions, ranging from simple to complex ones. The proposed approach can easily be applied in the description of other multitasking problems not covered in this paper.

[Mann70] Abstract: The problem of convergence, correctness, and equivalence of computer programs can be formulated by means of the satisfiability or validity of certain first-order formulas. An algorithm is presented for constructing such formulas for functional programs, i.e. programs defined by Lisp-like conditional recursive expressions.

[Mann74] Contents: The chapters of this book discuss the following topics. Computability: Finite automata; Turing machines; Turing machines as acceptors; Turing machines as Generators; Turing machines as algorithms. Predicate Calculus: Basic notions; Natural deduction; The resolution method. Verification of Programs: Flowchart programs; Flowchart programs with arrays; Algol-like programs. Flowchart Schemas: Basic notions; Decision problems; Formalization in predicate calculus; Translation problems. The Fixpoint Theory of Programs: Functions and functionals; Recursive programs; Verification methods.

[Mann78] Abstract: This paper explores a technique for proving the correctness and termination of programs simultaneously. This approach, the intermittent assertion method, involves documenting the program with assertions that must be true at some time when control passes through the corresponding point, but that need not be true every time. The method, introduced by Burstall, promises to provide a valuable complement to the more conventional methods.

The intermittent-assertion method is presented with a number of examples of correctness and termination proofs. Some of these proofs are markedly simpler than their conventional counterparts. On the other hand, it is shown that a proof of correctness or termination by any of the conventional techniques can be rephrased directly as a proof using intermittent assertions. Finally, it is shown how the intermittent-assertion method can be applied to prove the validity of program transformations and correctness of continuously operating programs.

[Math87a] Abstract: Several techniques have been developed in the past for improving the *vectorization level* of a program for fast execution on a vector processor like the Cray X/MP, Cyber 205 or Alliant FX/8. Most of these techniques are generally embedded in the language compilers of the vector machine, thereby making it easier for the programmer to benefit from them.

[Math87b] Abstract: In [Math87a] a program transformation technique is presented that aids in inducing vectorization in a given program *P*. This technique has applications in several areas including software testing using *mutation analysis* and in scheduling computations in an arbitrary program on an SIMD machine.

In this paper we provide a formulation of this transformation technique. The technique itself can be used to transform a given program *P*, desired to be executed on *N* data sets, to another program *VP*. Instead of executing *P* sequentially over the *N* data sets, *VP* executes *concurrently* over all the *N* data sets. The transformation rules are such that even though *P* may not yield well to vectorization, *VP* will.

[Math88a] Abstract: In this paper we describe a new methodology to merge a large number of program mutants into a small set of highly vectorizable programs. *Mutants* are generated when using the *mutation analysis* software testing method. Although mutation analysis is a simple and effective software testing methodology, it is computationally intensive. This becomes a major factor to be reckoned with when testing large programs.

The technique described in this paper enables a tester to exploit the architecture of vector processors, like the Cray X-MP or the Alliant FX/8 for efficient execution of all the mutants.

An analysis of the technique is presented elsewhere. The analysis aids a tester is estimating in advance the speed up that can be expected if program unification is employed. The speed up compares the time to execute a unified set of mutants with the time to execute the same set of mutants serially.

[Maug85] Abstract: In this paper, we present the main concepts used in our symbolic debugger for Ada. Described also is a companion tool, the Ada Program VIEWer, which gives users full access to program source while debugging. This debugger is one of the components of the Alsys tool set which aims at providing high-level Ada-oriented tools, incorporating state-of-the-art techniques for software design, documentation, and development.

[Mayf85] Abstract: The Second Workshop identified current issues in Ada Verification and focused on what is needed to build the foundation of an Ada Verification Technology. IDA Workshops will continue to be a meeting place for accessing the current state-of-the-art, identifying promising research areas, monitoring ongoing verification work, promoting the use of the evolving technology, and ensuring that valuable outputs from one area are fed into other areas. The desired product of these workshops will be recommendations to various bodies to coordinate and sponsor certain R&D activities. Working groups on special topics were also established.

[Mayf86] Abstract: The Third IDA Workshop was conducted at the Research Triangle Institute, Research Triangle Park, North Carolina on May 14-15, 1986. The theme of the workshop was "Reaching Verifiable Ada Systems by 1990" and addressed the following:

1. Advances in verification technology
2. Adaptation of current technology in Ada verification systems and methods
3. Broadening the base of support for work in Ada verification
4. Encouraging the participation by larger segments of both the Ada and the verification communities.

A detailed exposition of the Ada formal definition being developed by the European Economic Community was presented. This exposition took the form of a series of tutorial presentations (enclosed in this document) on various aspects of the dynamic and static semantics of the definition and its underlying formalisms. Dr. Harlan Mills from IBM's Federal Systems Division was the keynote speaker.

[McCa76] Abstract: This paper describes a graph-theoretic complexity measure and illustrates how it can be used to manage and control program complexity. The paper first explains how the graph-theory concepts apply and gives an intuitive explanation of the graph concepts in programming terms. The control graphs of several actual Fortran program are then presented to illustrate the correlation between intuitive complexity and the graph-theoretic complexity. Several properties of the graph-theoretic complexity are then proved which show, for example, that complexity is independent of physical size (adding or subtracting functional statements leaves complexity unchanged) and the complexity depends only on the decision structure of a program.

The issue of using nonstructured control graphs is given and a method of measuring the "structuredness" of a program is developed. The relationship between structure and reducibility is illustrated with several examples.

The last section of this paper deal with a testing methodology used in conjunction with the complexity measure; a testing strategy is defined that dictates that a program can either admit of a certain minimal testing level or the program can be structurally reduced.

[McCa77a] Abbreviated Preface: The objective of the study was to establish a concept of software quality and provide an Air Force acquisition manager with a mechanism to quantitatively specify and measure the desired

level of quality in a software product. Software metrics provide the mechanism for the quantitative specification and measurement of quality.

The first volume describes the process of developing our concept of software quality and what the underlying software attributes are that provide the quality, and defines the metrics which provide a measure of the degree to which the attributes exist.

The second volume describes the application of the metrics to software products and the validation of the metrics' relationship to software quality.

The third volume is a preliminary stand-alone reference document to be used by an acquisition manager to implement the techniques established during the study.

[McCa79] Abbreviated Introduction: These high costs [for life cycle management of large-scale software systems] result from characteristics of software that do not necessary relate to the correctness of the implementation of a function or how reliably the function operates, but instead relate to "how well" the software is designed, coded, and documented with respect to maintaining, transferring, modifying, etc., the software. This "how well" is a major aspect of software quality. This situation identifies a weakness in how the requirements of software system developments are defined currently. Emphasis is placed on the functions that must be performed, the schedule in which the system must be produced, and the cost of producing the system. Little or no attention is given to identifying what qualities over the life cycle the software system should exemplify. There are two major reasons for this focus. First, the initial operation of the system, how correctly and reliably the system performs, is always important to the sponsor of a development. It provides the first test of not only how well the developer has done, but also how well the sponsor has done in specifying, monitoring, and controlling the development. (Cost and schedule are obvious concerns since the system usually must be developed in a constrained period of time and within a constrained budget.) Second, no standard definition or identification of what qualities the acquisition manager should consider has been available. No mechanism has existed which would allow an acquisition manager to quantitatively specify the quality desired and then measure how well the development was progressing toward the desired quality. The little consideration given quality to date generally has been very subjective and not followed up by measurement or assurance activities.

The potential life cycle cost savings of standardized concept of software quality and a mechanism for specifying and measuring software quality are substantial considering the large portion of life cycle costs attributed to the qualities mentioned previously. The subject of this chapter and the next is a concept of software quality metrics and their application in a quality management program.

[McCa80a] Abstract: Software metrics (or measurements) which predict software quality have been refined and enhanced. Metrics were classified as anomaly-detecting metrics which identify deficiencies in documentation or source code, predictive metrics which measure the logic of the design and implementation, and acceptance metrics which are applied to the end product to assess compliance with requirements.

A Software Quality Measurement Manual was produced which contained procedures and guidelines for assisting software system developers in setting quality goals, applying metrics and making quality assessments.

[McCa80b] Abstract: Software metrics (or measurements) which predict software quality have been refined and enhanced. Metrics were classified as anomaly-detecting metrics which identify deficiencies in documentation or source code, predictive metrics which measure the logic of the design and implementation, and acceptance metrics which are applied to the end product to assess compliance with requirements.

A Software Quality Measurement Manual was produced which contained procedures and guidelines for assisting software system developers in setting quality goals, applying metrics and making quality assessments.

[McCa82a] Abstract: This Guideline presents the various applications of the Structured Testing methodology. The core of this technique is to avoid programs that are inherently untestable by first measuring and limiting program complexity. The definition and development of a program complexity measure is presented. The complexity measure is then, in the second phase of the methodology, used to quantify and proceduralize the testing process. How to apply the techniques to the maintenance process in order to identify the code that must be re-tested after

making a modification is illustrated.

[McCa82b] Abstract: This paper deals with the use of Structured Analysis just prior to system acceptance testing. Specifically, the drawing of Data Flow Diagrams (DFDs) was done after integration testing. The DFDs provided a picture of the logical flow through the integrated system for thorough system acceptance testing. System test sets were derived from the flows in the DFDs. System test repeatability was enhanced by the matrix which flowed from the test sets.

[McCa87a] Abstract: This report describes the results of a research and development effort to develop a methodology for predicting and estimating software reliability. A Software Reliability Measurement Framework was established which spans the life cycle of a software system and includes the specification, prediction, estimation, and assessment of software reliability. Data from 59 systems, representing over 5 million lines of code, were analyzed and generally applicable observations about software reliability were made. A detailed approach to the collection and analysis of reliability data is also presented.

[McCa87b] Abstract: The Guidebook provides detailed procedures for the preparation of software reliability predictions and estimations on DOD projects. In developing the Guidebook, 59 software systems were examined and 19 key variables were identified that affected the software reliability of those systems. Procedures to measure these variables were developed to account for the type of application, development, environment, various software characteristics (such as modularity and complexity), test technique, test effort and test coverage. A methodology was also provided to use these measures to predict software fault density and software failure rates.

The Guidebook could be applied by an Air Force acquisition office to help plan for adequate software reliability early in a project's life, specify achievable software reliability goals in a RFP, evaluate progress toward those goals at key project milestones and decide when to release the software. The Guidebook could also be used by the technical staff to establish thresholds for critical measures such as complexity. In addition, the Guidebook also contains Quality Review and Standards Review Checklists that can be used in conjunction with the software reliability prediction and estimation methodology. The Quality Review Checklists are used to assess the quality of the requirements and design representation of the software while the Standards Review Checklist would be applied to software code. The checklists provide good guidance for ensuring that quality is built into the software.

[McCl78a] Introduction: A frequently stated objective of structured programming is to control program complexity. However, since the notion of complexity is not well understood and the existing techniques for measuring complexity are crude, it is difficult to determine if indeed structured programming can achieve this objective. The purpose of this paper is two-fold:
1. to discuss the probable sources of complexity in a well-structured program
2. to present a methodology for measuring and controlling complexity in a well-structured program.

[McCl78b] Abbreviated Preface: This book is intended for the programmer in the business community. Its purpose is to present software methodologies and techniques to guide the programmer in developing well-structured programs. In general, the methodologies discussed are applicable to any higher level programming language (e.g., ALGOL 60, PL/1, COBOL) that provides the basic constructs required by structured programming. Explaining how to code well-structured programs is only an ancillary purpose of the book. The primary intent is to extend the structured programming approach to include the programming process as well as the program structure. This is accomplished by:
1. Clarifying the meaning of basic software terms such as structured programming, top-down programming, and bottom-up programming (see chapter 2)
2. Presenting guidelines for selecting and applying an appropriate design methodology for writing a well-structured program (see chapters 3 and 4)
3 Including program complexity analysis as an integral step in the programming process (see chapter 5).

A commonly stated objective of structured programming is to control program complexity-the issue being

that the more complex a program, the more difficult it becomes to understand how the program works. Obviously, program complexity is a function of program size; but also it is a function of the number of possible program execution paths and the control variables that direct path selection. The structured programming approach attempts to control program complexity by restricting module invocation and, in general, by restricting the use of control structures. It does not, however, limit the use of control variables other than to suggest that they be accessed in as few program modules as possible.

In this book, the complexity issue is confronted by providing a means of quantitatively measuring program complexity and by inserting complexity control as a postdesign/preimplementation step in the programming process. In this way, the programmer can more effectively analyze and control the quality of a program as it is being developed.

[McGa85b] Abstract: The availability and quality of computer resources during the software development process has been speculated to have measurable, significant impact on the efficiency of the development process and the quality of the resulting product. Environmental components such as the types of tools, machine responsiveness, and quantity of direct access storage may play a major role in the effort to produce the product and in its subsequent quality as measured by factors such as reliability and ease of maintenance.

During the past six years, the NASA Goddard Space Flight Center has conducted experiments with software projects in an attempt to better understand the impact of software development methodologies, environments, and general technologies on the software process and product. Data has been extracted and examined from nearly 50 software development projects. The data collection and analysis has been performed jointly by NASA, the Computer Science Department at the University of Maryland, and the Computer Science Corp. The projects have varied in size from 3000 up to 130,000 lines of code with an average of 60,000 lines of code. All have been related to the support of satellite flight dynamics ground-based computations. As a result of changing situations and technology, the computer support environment has varied widely. Some projects enjoyed fast response time, archaic tool support, and limited terminal access to the development machine.

This study examined the relationship between computer resources and the software development process and product as exemplified by the subject NASA data. Based upon the results, a number of computer resource-related implications are provided.

[McMu80] Abstract: A compiler-based specification and testing system for defining data types has been developed. The system, DAISTS (data abstraction implementation, specification, and testing system) includes formal algebraic specifications and statement and expression test coverage monitors. This paper describes our initial attempt to evaluate the effectiveness of the system in helping users produce software. In an exploratory study, subjects without prior experience with DAISTS were encouraged by the system to develop effective sets of test cases for their implementations. Furthermore, an analysis of the errors remaining in the implementations provided valuable hints about additional useful testing metrics.

[McMu83] Abstract: This paper describes our experience in specifying, implementing, and validating a record-oriented text editor. Algebraic axioms served as the specification notation; and the implementation was tested with a compiler-based system that uses the axioms to test implementations with a finite collection of test cases. Formal specifications were sometimes difficult to produce, but helped reveal errors during unit testing. Thorough exercising of the implementations by the specifications resulted in few errors persisting until integration.

[Medi81] Abstract: This paper describes an incremental programming environment (IPE) based on compilation technology, but providing facilities traditionally found only in interpretive systems. IPE provides a comfortable environment for a single programmer working on a single program.

In IPE the programmer has a uniform view of the program in terms of the programming language. The program is manipulated through a syntax-directed editor and its execution is controlled by a debugging facility, which is integrated with the editor. Other tools of the traditional tools cycle (translator, linker, loader) are applied automatically and are not visible to the programmer. The only interface to the programmer is the user interface of the editor.

[Mell82] Abstract: This paper describes the formal specification and proof methodology employed to demonstrate that the SIFT computer meets its requirements. The hierarchy of design specifications is shown, from very abstract descriptions of system function down to the implementation. The most abstract design specifications are simple and easy to understand, almost all details of the realization having been abstracted out, and can be used to ensure that the system functions reliably and as intended. A succession of lower level specifications refine these specifications into more detailed and more complex views of the system design, culminating in the Pascal implementation. The paper describes the rigorous mechanical proof that the abstract specifications are satisfied by the actual implementation.

[Meye67] Abbreviated Introduction: Anyone familiar with the theory of computability will be aware that practical conclusions from the theory must be drawn with caution. If a problem can theoretically be solved by computation, this does not mean that it is practical to do so. Conversely, if a problem is formally undecidable, this does not mean that the subcases of primary interest are impervious to solution by algorithmic methods.

The question of detecting improvable programs will appear again later in this paper, but our main concern will be with a related question: can one look at a program and determine an upper bound on its running time? Again, a fundamental theorem in the theory of computability implies that this cannot be done. The Theorem does *not* imply that one cannot bound the running time of broad categories of interesting programs, including programs capable of computing all the arithmetic functions one is likely to encounter outside the theory of computability itself.

In the next section we describe such a class of programs, called "Loop programs." Each Loop program consists only of assignment statements and iteration (loop) statements. Although Loop programs cannot compute all the computable functions, they can compute all the *primitive* recursive functions.

[Miar83] Abstract: The consensus in the programming community is that indentation aids program comprehension, although many studies do not back this up. We tested program comprehension on a Pascal program. Two styles of indentation were used - blocked and nonblocked - in addition to four possible levels of indentation (0,2,4,6 spaces). Both experienced and novice subjects were used. Although the blocking style made no difference, the level of indentation had a significant effect on program comprehension. (2-4 spaces had the highest mean score for program comprehension.) We recommend that a moderate level of indentation be used to increase program comprehension and user satisfaction.

[Mili84] Abstract: Many program verification methods are known nowadays: Inductive Assertion Method, Symbolic Execution Method, Subgoal Induction Method, Computational Induction Method, Structural Induction Method, Fixpoint Theory of Programs. This paper presents a simple classification of them.

[Mill74a] Abstract: A structural basis for the formulation of testcases for given computer programs has been found to be an effective and efficient strategy. An existing automated program validation system employs these techniques with good success in minimizing the number of testcases required; this same system permits automatic identification of testcases in a high proportion of instances. Research aimed at fully automating the testcase generation process continues.

[Mill75a] Abstract: Structured programming has proved to be an important methodology for systematic program design and development. Structured programs are identified as compound function expressions in the algebra of functions. The algebraic properties of these function expressions permit the reformulation (expansion as well as reduction) of a nested subexpression independently of its environment, thus modeling what is known as stepwise program refinement as well as program execution. Finally, structured programming is characterized in terms of the selection and solution of certain elementary equations defined in the algebra of functions. These solutions can be given in general formulas, each involving a single parameter, which display the entire freedom available

[Mill75b] Abstract: Computer software production costs continue to increase–to the point where these costs are overwhelmingly dominant in the majority of computer applications. At the same time, there is an increasing

sense of urgency about software reliability, which must be achieved without significant additional software cost increases. Techniques for attaining suitable levels of implicit software quality and reliability range from programming and managerial disciplines which seek to instill it from the onset of a software system development, to techniques for systematically testing (exercising) software systems. Software testing is a viable technique for completed (or about-to-be-completed) software systems because: (1) it permits full "validation" of the software system, (2) it can approximate a formal program proof of correctness, and (3) it is largely automatable and relatively inexpensive.

For very large software systems, or those in particularly crucial applications, it is possible to reduce the verification, validation, and testing cost by avoiding certain difficult-to-test programming constructs. Some of these potentially troublesome forms are identified, explanations of the way in which they unnecessarily increase software testing costs are given, and engineering solutions which seek to avoid these difficulties are given.

[Mill75c] Abstract: Software quality enhancement can be achieved in the near term through use of a systematic program testing methodology. The methodology attempts to relate functional software testcases with formal software specifications as a means to achieve correspondence between the software and its specification. To do this requires generation of appropriate testcase data.

Automatic testcase generation is based on *a priori* knowledge of two forms of internal structure information: a representation of the tree of subschema automatically identified from within each program text, and a representation of the iteration structure of each subschema. This partition of a large program allows for efficient and effective automatic testcase generation using straightforward backtracking techniques.

During backtracking a number of simplifying, consolidating, and consistency analyses are applied. The result is either (1) early recognition of the impossibility of a particular program flow, or (2) efficient generation of input variable specifications which cause the testcase to traverse each portion of the required program flow.

A number of machine output examples of the backtracking facility are given, and the general effectiveness of the entire process is discussed. in creating correct structured programs. testing cost by avoiding certain difficult-to-test programming constructs.

[Mill75d] Abstract: There is no foolproof way to ever know that you have found the last error in a program. So the best way to acquire confidence that a program has no errors is never to find the first one, no matter how much it is tested and used. It is an old myth that programming must be an error-prone, cut-and-try process of frustration and anxiety. The new reality is that you can learn to consistently write programs which are error free in their debugging and subsequent use. This new reality is founded in the ideas of structured programming and program correctness, which not only provide a systematic approach to programming but also motivate a high degree of concentration and precision in the coding subprocess.

[Mill77a] Abbreviated Introduction: The problems of providing quality assurance for computer software have received a good deal of attention from the computing community. Such areas as program proving, automatic programming, structured programming, and hierarchical design/development methodologies have all experienced significant growth - largely as a result of the increased attention focused on them. Program testing, on the other hand, has not enjoyed the same level of intensive investigation, even though it has a number of technical and intuitive appeals.

Both art and theory operate in program testing today. The "art" of program testing suggests new theoretical routes which drive the development of additional "theory" which, in turn, drives the accumulation of further art.

This paper describes some recent efforts to build a bridge linking the theory of program testing with its practice. Although building that bridge has been a desirable goal, only now has sufficient research insight and actual testing experience been gained to even begin contemplating the form this practically oriented but strongly founded bridge can take.

[Mill77b] Abstract: Automated program testing tools can have significant utility in a formal program testing and quality assurance activity. It is possible to characterize automated tools by the degree to which they require

modification of the source programs, and by the level of automation they achieve. Ten categories of automated testing tools are described functionally and operationally. Commercially available examples of each class of tool are given. When the data is available, indications of the relative effectiveness of the tool are also given.

[Mill79a] Abstract: The current state of the art in program testing technology is identified in terms of the philosophical underpinnings of software, the theoretical foundations of the field, the tools and techniques that can be brought to bear in a testing activity, the methods that exist for planning and measuring the testing activity, and the methods of management and control that exist.

The future needs for program testing technology are identified in three major categories: theoretical foundations, methodology, and automated tools. Over twenty needs for program testing targeted for the 1980s timeframe are identified in detail.

[Mill79b] Abbreviated Introduction: It has been said that one of the biggest problems in the software quality assurance community is that program proving techniques appear not to scale up, while systematic testing methods don't appear to scale down! What's intended here is to observe that systematic testing methods attempt to deal with "large" phenomena, while program proving techniques deal at a very fine level of detail. Naturally enough, the outcomes of the two processes will be different.

The objective of this short piece is to present some statistics derived in a relatively large-scale systematic testing activity and to suggest what some of the implications of those "numbers" might be.

[Mill79c] Abstract: The workshop's general and session chairmen offer their summaries of the challenges to the software testing community identified at the December meeting.

[Mill81a] Abbreviated Preface: This is the second edition of *Software Testing and Validation Techniques*. This edition updates and amends the set of papers included in the prior edition that was first organized in 1978.

Since that time, there have been several advances of significance in the software testing and validation field; a number of new papers have been published, and in many ways the field has become more mature and stable. In addition, the field has become deeper and richer, possibly as the result of increased emphasis on software quality and on quality assurance. Each year, the number of published papers of significance to software testing and validation has increased, as has the number of researchers actively involved in the field.

The papers we have added to this edition fall into the boundaries we have previously used to organize the book: Theoretical Foundations, Static Analysis - Tools and Techniques, Dynamic Analysis - Tools and Techniques, Effectiveness Assessment, Management and Planning, and Research and Development.

[Mill81b] Abstract: Comparing the usefulness of methodologies for software development can be especially difficult when the services offered are based on different philosophies. Two systems, AFFIRM and HDM, were compared for their application to operation system security analysis. The assessment technique was to specify and analyze for security flaws on both systems a miniature example of a security kernel. The specification languages are at the opposite poles of the range from algebraic axioms to transition specifications. The types of security properties that could be verified with the tools available were access policy invariants and information flows. One theorem prover was highly interactive and the other nearly automatic. We found that the example could be specified satisfactorily and recognizably on both systems with a comparable amount of effort. The security analyses, on the other hand, led to very different verification tasks and different results. The two results were complementary rather than contradictory, and some additional experimentation, guided by theoretical suspicions, showed the exact relationship between them.

[Mill84] Abstract: Writing distributed programs is difficult for at least two reasons. The first reason is that distributed computing environments present new problems caused by asynchrony, independent time bases, and communication delays. The second reason is that there is a lack of tools available to help the programmer understand the program he/she has written. The tools we use for single machine environments do not easily generalize to a distributed environment. There has been only limited success with previous systems that have tried to help the

programmer in developing, debugging, and measuring distributed programs.

To better understand distributed programs we have: specified a model for distributed computations, developed a measurement methodology from this model, constructed tools to implement the measurements, and developed data analysis techniques to obtain useful results from the measurements. The most important feature of the models, methodology, and tools is consistency between the programmers view, the computation model, the measurement methodology and the analysis.

This consistency has resulted in several benefits. The first is simplicity of structure throughout the measurement and analysis tools. The second benefit is the ease of obtaining useful information about a programs behavior.

The model of distributed programs defines the two basic actions of a program to be computation and communication. Our research focuses on the communications performed by a program. The measurement model is based on the monitoring of communications between the parts of a program. Given our definition of a program, monitoring communications completely encapsulates the behavior of a computation. From the measurement model, we have constructed tools to measure distributed programs for two working operating systems, UNIX and DEMOS/MP. These measurement tools provide data on the interactions between the parts of a distributed program.

We have developed a number of analysis techniques to provide information from the data collected. We can report communications statistics on message counts, queue lengths, and message waiting times. We can perform more complex analyses such as measuring the amount of parallelism in the execution of a distributed program. The analyses also include detecting paths of causality through the parts of a distributed program. The measurement tools and analyses can be constructed so that the results can be fed back into the operating system to help with scheduling decisions.

[Mill85] **Abstract:** A new method for estimating the present failure rate of a program is presented. A crude nonparametric estimate of the failure rate function is obtained from past failure times. This estimate is then smoothed by fitting a completely monotonic function, which is the solution of a quadratic programming problem. The value of the smoothed function at present time is used as the estimate of present failure rate. A Monte Carlo study gives an indication of how well this method works.

[Mill87a] **Introduction:** Recent experience demonstrates that software can be engineered under statistical quality control and that certified reliability statistics can be provided with delivered software. IBM's Cleanroom process has uncovered a surprising synergy between mathematical verification and statistical testing of software, as well as a major difference between mathematical fallibility and debugging fallibility in people.

With the Cleanroom process, you can engineer software under statistical quality control. As with cleanroom hardware development, the process's first priority is defect prevention rather than defect removal (of course, any defects not prevented should be removed). This first priority is achieved by using human mathematical verification in place of program debugging to prepare software for system test.

Its next priority is to provide valid, statistical certification of the software's quality through representative-user testing at the system level. The measure of quality is the mean time to failure in appropriate units of time (real or processor time) of the delivered product. The certification takes into account the growth of reliability achieved during system testing before delivery.

To gain the benefits of quality control during development, Cleanroom software engineering requires a development cycle of concurrent fabrication and certification of product increments that accumulate into the system to be delivered. This lets the fabrication process be altered on the basis of early certification results to achieve the quality desired.

[Mill87b] **Abstract:** The Interrogator is a Prolog program that searches for security vulnerabilities in network protocols for automatic cryptographic key distribution. Given a formal specification of the protocol, it looks for message modification attacks that defeat the protocol objective. It is still under development, but it has been able to rediscover a known vulnerability in a published protocol. It is implemented in LM-Prolog on a Lisp Machine, with a graphical user interface.

[Misr81] Abstract: We present a proof method for networks of processes in which component processes communicate exclusively through messages. We show how to construct proofs of invariant properties which hold at all times during network computations, and terminal properties which hold upon termination of network computations, if network computation terminates. The proof method is based upon specifying a process by a pair of assertions, analogous to pre- and post-conditions in sequential program proving. The correctness of network specification is proven by applying inference rules to the specifications of component processes. Several examples are proved using this technique.

[Misr83] Abstract: Methods proposed for software reliability prediction are reviewed. A case study is then presented of the analysis of failure data from a space shuttle software project to predict the number of failures likely during a mission, and the subsequent verification of the predictions.

[Miya87] Abstract: The Deviation-value (D-value) is a new measure for software data involved during software development. The D-value provides an alternative to software metrics based upon "per number of lines of code" such as error rate (number of errors per thousand lines of code) and documentation rate (number of pages of module design documentation per thousand lines of code). Using D-value, the data of software modules are much more fairly evaluated than these conventional metrics.

This paper presents the derivation of the D-value using the theoretical background of a control chart called u chart and weighted regression analysis. The advantage of using the D-value rather than metrics based upon "per number of lines of code" is demonstrated through an analysis of the data of four projects. The D-value is used to find the data items which actually relate to software quality, and we find that the quality of each module measured by D-value becomes better as the documentation rate D-value increases. Finally, using the theory behind the D-value, a new software acceptance guideline is discussed.

[MiyaXX] Abstract: Effective software reliability evaluation requires theories of software reliability which define and deal with software reliability quantitatively, technologies for reliability data measurement and data analysis, techniques to estimate or predict software reliability, and practical reliability evaluation methodologies which effectively reflect the characteristics of software. This paper addresses the extents to which these requirements are currently met, and introduces improved approaches for an effective software reliability evaluation. Introduced are the methodologies for software reliability evaluation and the software reliability evaluation-aid tools.

[Mizu83] Overview: In Japan, people are the key to software quality control. At NEC, members of a QC team work together to achieve high standards, competing with other teams for awards.

[Moha79] Abstract: Several software quality assessment methods which span the software life cycle are discussed. The quality of a system design can be estimated by measuring the system entropy function or the system work function. The quality improvement due to reconfiguration can be determined by calculating system entropy loading measures. Software science and Zipf's law are shown to be useful for estimating program length and implementation time. Deterministic and statistical methods are presented for predicting the number of errors. Testing theory is useful in planning the program test process; as discussed in this paper, it includes measurement of program structural characteristics to determine test effectiveness and test planning. Statistical models for estimating software reliability are also discussed.

[Mora75] Abstract: Estimates of future performance of a software package are obtained from debugging data in essentially two ways. In one way the record in time of the occurrence of anomalies is used; in this paper three different mathematical models of failure rates are described, together with illustrative predictions of MTTF and of the total error content using actual trouble report data. A second estimate of performance of a program is by its "operational reliability" which is obtained through variations of input data according to assumed probability laws. With respect to this procedure, an outline is given of the goals of some research currently being done at McDonnell Douglas Astronautics.

[Mora78a] Introduction: [The author] found the review by Dennis Geller of the Glenford J. Myers' book, *Software Reliability: Principles and Practices* (Computer, October 1977, pp. 117-118), provided excellent coverage of the principal theme of the book, that being software, vis-a-vis reliability. While [the author concurs] with Mr. Geller on all of the points which he makes, both favorable and unfavorable, [the authors] own reading of the book focussed on the reliability aspects of the material presented.

From this perspective there are several comments on the book which [the author offers] for consideration. [The author feels] these comments are especially timely, since the Myers' interpretation of reliability, presented in the first book on the subject, reinforces the erroneous concept that reliability "equates to" perfection.

[Mora78c] Introduction to Comment on "A Review and Evaluation of Software Science," by A. Fitzsimmons and T. Love: This article raised two questions in my mind: whether the "effort" measure is a good measure of complexity; and whether the correlation coefficient is a reliable tool for validating the conjectures of software science.

[Mora80] Abstract: Two variations of the Jelinski/Moranda model are described. The first permits estimation of the error content of the completed software package using data which is taken on only portions of the package. That model is applicable when the eventual size of the program is known at the outset.

The second model permits a similar analysis during the development of any software package which is homogeneous with respect to its complexity (error making/finding).

These models should assist analysts in the determination of error content early on. They should also eliminate the present practice of applying models to the wrong regime (decreasing failure rate models applied to growing-in-size software).

[More87] A theory of fault-based program testing is defined and explained. Testing is fault-based when it seeks to demonstrate that prescribed faults are not in a program. It is assumed that a program can only be incorrect in a limited fashion specified by associating alternate expressions with program expressions. Classes of alternate expressions can be infinite. Substitution of an alternate expression for a program expression yields an alternate program that is potentially correct. The goal of fault-based testing is to produce a test set that differentiates the program from each of its alternates. A particular form of fault-based testing based on symbolic execution is presented. In symbolic testing program expressions are replaced by symbolic alternatives that represent classes of alternate expressions. The output from the system is an expression in terms of the input and the symbolic alternative. Equating this with the output from the original program yields a propagation equation whose solutions determine those alternatives which are not differentiated by this test.

[More88] Abstract: Testing is *fault-based* when its goal is to demonstrate the absence of prespecified faults. This paper presents a framework that characterizes fault-based testing schemes based on how many prespecified faults are considered and on the contextual information used to deduce the absence of those faults. Established methods of fault-based testing are placed within this framework. Most methods either are limited to finite fault classes, or focus on local effects of faults rather than global effects. A new method of fault-based testing called *symbolic testing* is presented by which infinitely many prespecified faults can be proven to be absent from a program based upon the global effect the faults would have if they were present. Circumstances are discussed as to when testing with a finite test set is sufficient to prove that infinitely many prespecified faults are not present in a program.

[Morg86] Abstract: This paper focuses on a reachability graph analyzer (RGA), a tool which provides mechanisms for proving general system properties (e.g., deadlock-freeness) as well as system-specific properties. The tool is sufficiently general to allow a user to apply complex user-defined analysis algorithms to reachability graphs. The alternating-bit protocol with a bounded channel is used to demonstrate the power of the tool and to point to future extensions.

[Morg87] The introduction of concurrency into programs has added to the complexity of the software design

process. This is most evident in the design of communications protocols where concurrency is inherent to the behavior of the system. The complexity exhibited by such software systems makes more evidence the need for computer-aided tools for automatically analyzing behavior.

The Distributed Systems project at UCI has been developing techniques and tools, based on Petri nets, which support the design and evaluation of concurrent software systems. Techniques based on constructing reachability graphs that represent projections and selections of complete state-spaces have been developed. This paper focuses attention on the computer-aided analysis of these graphs for the purpose of proving correctness of the modeled system. The application of the analysis technique to evaluating simulation results for correctness is discussed. The tool which supports this analysis (the reachability graph analyzer, RGA) is also described. This tool provides mechanisms for proving general system properties (e.g., deadlock-freeness) as well as system-specific properties. The tool is sufficiently general to allow a user to apply complex user-defined analysis algorithms to reachability graphs. The alternating-bit protocol, with a bounded channel, is used to demonstrate the power of the tool and to point to future extensions.

[Mori83] Abstract: This paper describes techniques for the representation and refinement of visual specifications in the context of PegaSys (Programming Environment of the Graphical Analysis of SYStems), a system that supports a visual paradigm for the development and explanation of interactions among the conceptual entities in a system design. Pictures have a computational meaning that is represented in a formal language, called the *form calculus.* The form calculus is extensible in that it contains a core set of primitives which can be used to build a variety of abstract design models. Complexity is managed by means of picture hierarchies, whose construction is guided by a precise refinement methodology.

The representation and refinement techniques presented here have been implemented and all reasoning is fully automatic and efficient. Determining the validity of a picture refinement, for example, involves either the application of a simple graph algorithm or the proof of a formula whose predicates range over small, finite sets. Excerpts from a sample session with PegaSys are used to illustrate a hierarchy of visual specifications.

[Morr71] Abstract: An inductive method for proving things about recursively defined functions is described. It has shown to be useful for proving partial functions equivalent and thus applicable in proofs about interpreters for programming languages.

[Morr77] Abstract: A new proof method, subgoal induction, is presented as an alternative or supplement to the commonly used inductive assertion method. Its major virtue is that it can often be used to prove a loop's correctness directly from its input-output specification without the use of an invariant. The relation between subgoal induction and other commonly used induction rules is explored and, in particular, it is shown that subgoal induction can be viewed as a specialized form of computation induction. Finally, a set of sufficient conditions are presented which guarantee that an input-output specification is strong enough for the induction step of a proof by subgoal induction to be valid.

[Muno88] Abstract: An approach to software product testing is presented. The approach uses the following techniques: automatic test case generation, self-checking test cases, black box test cases, random test cases, sampling a form of exhaustive testing, correctness measurements, and the correction of defects in the test cases instead of in the product (defect circumvention). The techniques have been cost effective and applied to very large products.

[Muns89] Abstract: Software complexity metrics attempt to define the unique characteristics of computer programs in an analytical way. Many such metrics have been developed to explain various perceived differences among programs. Many studies have been conducted to show the similarity among classes of these metrics. What is lacking in this body of literature is a technique which will aid in the establishment of the true dimensionality of the complexity problem space.

The objective of this paper is to examine some recent investigations in the area of software complexity using factor analysis to begin an exploration of the actual dimensionality of the complexity metrics. This

technique can expose the relationships of these many metrics, one to another. Some correlation coefficients from recent empirical studies on software metrics were factor analyzed, showing the probable existence of five complexity dimensions within thirty different complexity measures.

[Mura89] Abstract: This paper presents a method for detecting deadlocks in Ada tasking programs using structural and dynamic analysis of Petri nets. Algorithmic translation of the Ada programs into Petri nets that preserve control flow and message flow properties is described. Properties of these Petri nets are discussed, and algorithms are given to analyze the nets to obtain information about static deadlocks that can occur in the original programs. Petri net invariants are used by the algorithms to reduce the time and space complexities associated with dynamic Petri net analysis (i.e., reachability graph generation).

[Musa75] Abstract: An approach to a theory of software reliability based on *execution time* is derived. This approach provides a model that is simple, intuitively appealing, and immediately useful.

The theory permits the estimation, in advance of a project, of the amount of testing in terms of execution time required to achieve a specified reliability goal [stated as a mean time to failure (MTTF)]. Execution time can then be related to calendar time, permitting a schedule to be developed. Estimates of execution time and calendar time remaining until the reliability goal is attained can be continually remade as testing proceeds, based only on the length of the execution time intervals between failures. The current MTTF and the number of errors remaining can also be estimated. Maximum likelihood estimation is employed, and confidence intervals are also employed. The foregoing information is obviously very valuable in scheduling and monitoring the progress of program testing. A program has been implemented to compute the foregoing quantities.

The reliability model that has been developed can be used in making system tradeoffs involving software or software and hardware components. It also provides a soundly based unit of measure for the comparative evaluation of various programming techniques that are expected to enhance reliability.

The model has been applied to four medium-sized software development projects, all of which have completed their life cycles. Measurements taken of MTTF during operation agree well with the predictions made at the end of system test. As far as the author can determine, these are the first times that a software reliability model has been used *during* software development projects. The paper reflects and incorporates the practical experience gained.

[Musa79a] Abstract: This paper investigates the validity of the execution-time theory of software reliability. The theory is outlined, along with appropriate background, definitions, assumptions, and mathematical relationships. Both the execution time and calendar time components are described. The important assumptions are discussed. Actual data are used to test the validity of most of the assumptions. Model and actual behavior are compared. The development projects and operational computation center software from which the data have been obtained are characterized to give the reader some basis for judging the breadth of applicability of the concepts.

[Musa79b] Introduction: Boehm, Brown, and Lipow have characterized the multi-dimensional nature of software quality in terms of a hierarchy of attributes. One of the high-level attributes is reliability, which they define qualitatively as the satisfactory performance of intended functions. This definition may be refined to the quantitative statement "probability of failure-free operation in a specified environment for a specified time." A "failure" is an unacceptable departure of program operation from program requirements, where, as in the case of hardware, "unacceptable" must ultimately be defined by the user. The term "fault" will be used to indicate the program defect that causes the failure. Several trends have recently combined to escalate the importance of quantitative software reliability measures:

1. The large and growing number of real-time and interactive systems has increased the operational and cost impact of failure.
2. The increasing number, size, and complexity of computer networks and distributed processing systems have multiplied the risk and effects of failure.
3. The explosive growth of personal computing has created a demand for relatively foolproof software for unsophisticated users.

Measurement is seen to be important as soon as one recognizes that in software as in hardware there can be too much as well as too little reliability. Improvement of reliability, of course, costs money, and usually impacts development schedules and system performance (in the case of software, through increased memory, processing time, and peripherals requirements). The system engineer and the manager have to make design tradeoffs among the foregoing factors and it is best that this be done in quantitative terms. The need for a quantitative reliability measure continues throughout the development process, particularly during test, since reliability is a valuable indicator of system status. Finally, reliability or mean-time-to-failure (MTTF) is a useful metric for characterizing system operation and for controlling change during the maintenance phase. This paper will focus on the system engineering application, but it will also touch on monitoring the system test phase and controlling change during maintenance.

[Musa80b] Abstract: The theme of this paper is the field of software reliability measurement and its application. Needs for and potential uses of software reliability measurement are discussed. Software reliability and hardware reliability are compared, and some basic software reliability concepts are outlined. A brief summary of the major steps in the history and evolution of the field is presented. Two of the leading software reliability models are described in some detail. The topics of combinations of software (and hardware) components and availability are discussed briefly. The paper concludes with an analysis of the current state-of-the-art and a description of further research needs.

[Musa84] Abstract: A new software reliability model is developed that predicts expected failures (and hence related reliability quantities) as well or better than existing software reliability models, and is simpler than any of the models that approach it in predictive validity. The model incorporates both execution time and calendar time components, each of which is derived. The model is evaluated using actual models.

[Musa87] Table of Contents: Introduction to software reliability, selected models, applications. Practical Application, system definition, parameter determination, project-specific techniques, application procedures, implementation planning. Theory, software reliability modeling, markovian models, description of specific models, parameter estimation, comparison of software reliability models, calendar time modeling, failure time adjustment for evolving programs.

[Musa89] Abbreviated Introduction: How do you validate that a piece of software loaded into a processor functions correctly? One traditional answer is that you subject it to a rigorous system test. But there is a fundamental problem: For any but the most trivial application, the number of distinct input combinations you would need to verify is enormous - orders and orders of magnitude larger than any number that can be tested exhaustively.

Furthermore, because of the discrete nature of computer memory and processing, the difference of a single input bit out of thousands may be all that separates an input combination that runs successfully from one that doesn't.

How then do you validate software? In hard engineering terms, the answer is that up to now you really haven't. There is a lot of lore about system testing, but it all boils down to guesswork. That is, it is guesswork unless you can structure the problem and perform the testing so that you can apply mathematical statistics.

If you can do this, you can say something like "No, we cannot be absolutely certain that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95-percent confidence that the probability of 1,000 CPU hours of failure-free operation in a probabilistically defined environment is at least 0.995."

When you do this, you are applying *software-reliability measurement*. In this situation, this is the *best* you can do. For purists, this may not be a satisfactory answer to our initial question. But with software-reliability measurement, you do not deal explicitly with the vastness, discreteness, and discontinuity of a program input space - you sidestep these imponderables by using statistics to provide concrete, quantitative guidance.

In this article, we define the basic concepts of software-reliability measurement and show you how to use them in software validation.

[Muss79] Abstract: This paper describes the data type definition facilities of the AFFIRM system for program specification and verification. Following an overview of the system, we review the rewrite rule concepts that form the theoretical basis for its data type facilities. The main emphasis is on methods of ensuring *convergence* (finite and unique termination) of sets of rewrite rules and on the relation of this property to the equational and inductive proof theories of data types.

[Myer77] Abstract: A recent paper has described a graph-theoretic measure of program complexity, where a program's complexity is assumed to be only a factor of the program's decision structure. However, several anomalies have been found where a higher complexity measure would be calculated for a program of lesser complexity than for a more-complex program. This paper discusses these anomalies, describes a simple extension to the measure to eliminate them, and applies the measure to several programs in the literature.

[Myer78a] Abstract: This paper describes an experiment in program testing, employing 59 highly experienced data processing professionals using seven methods to test a small PL/1 program. The results show that the popular code walkthrough/inspection method was as effective as other computer-based methods in finding errors and that the most effective methods (in terms of errors found and cost) employed pairs of subjects who tested the program independently and then pooled their findings. The study also shows that there is a tremendous amount of variability among subjects and that the ability to detect certain types of errors varies from method to method.

[Myer78b] Introduction: Moranda's remarks in The Open Channel in *Computer*, April 1978, on my book, *Software Reliability: Principles and Practices*, fall into two general areas. First, he feels that the book "is not about software reliability as it has come to be defined." Second, he seems defensive about my "low opinion" (his words) of probability-based models, particularly his model.

[Myer79] Table of Contents: A Self-Assessment Test. The Psychology and Economics of Program Testing. Program Inspections, Walkthroughs, and Reviews. Test Case Design. Module Testing. Higher-Order Testing. Debugging. Test Tools and Other Techniques.

[Myer83] Abstract: Many modern computer languages allow the programmer to define and use a variety of data types. Few programming systems, however, allow the programmer similar flexibility when displaying the data structures for debugging, monitoring and documenting programs. *Incense* is a working prototype system that allows the programmer to interactively investigate data structures in actual programs. The desired displays can be specified by the programmer or a default can be used. The default displays provided by Incense present the standard form for literals of the basic types, the actual names for scalar types, stacked boxes for records and arrays, and curved lines with arrowheads for pointers. In addition to displaying data structures, Incense also allows the user to select, move, erase and redimension the resulting displays. These interactions are provided in a uniform, natural manner using a pointer device (*mouse*) and keyboard.

[Myhr68] Abstract: Some specific comparisons are made in this note between the use of the asymptotic Chi-square distribution of the likelihood ratio and the asymptotic normality of the maximum likelihood estimates to obtain confidence intervals for reliabilities of arbitrary systems when only failure data on the components is known. In all the comparisons made, using moderate samples and systems of average complexity, the asymptotic Chi-square appears to give much more accurate confidence intervals. Although the asymptotic Chi-square method requires more computation for most systems than does the method based on asymptotic normality, these examples indicate the Chi-square method would yield superior results in most practical instances.

[NBS82a] Abstract: Thirty techniques and tools for validation, verification, and testing (V,V&T) are described. Each description includes the basic features of the technique or tool, the input, the output, an example, an assessment of the effectiveness and usability, applicability, an estimate of the learning time and training, an estimate of needed resources, and references.

[NBS82b] Abstract: Thirty techniques and tools for validation, verification, and testing (V,V&T) are described. Each description includes the basic features of the technique or tool, the input, the output, an example, an assessment of the effectiveness and usability, applicability, an estimate of the learning time and training, an estimate of needed resources, and references.

[Nage84] Abbreviated Summary: This report documents the second of two studies performed by Boeing Computer Services on modeling the process of software error detection from the results of experiments specifically designed to complement this activity. The experiments consist of simulations conducted on code prepared under controlled conditions and executed with randomly selected inputs. Six codes were developed in the first study and this study continues the experiment with six more. The code is initialized to an original state and flexed with independently generated random inputs. Errors are corrected as they are encountered until a stopping rule is satisfied. Replication is introduced by repeating the entire process from initialization.

The previous study explored the effects of programmer and problem as experimental design factors on the error structure. The current study enlarges this set of factors by varying the experience level of the programmer and the relative frequency or usage of the program units. The use of FORTRAN is contrasted with the use of a micro-based assembler language as another design factor. All of these factors, not surprisingly, affected performance and some very tentative relational hypotheses are suggested.

An analytic framework for replicated and non-replicated (i.e., traditional) software experiments is initiated in this study in order to present the results in a meaningful context. A method of obtaining an upper bound on the error rate of the next error is proposed. Two other forecasting methods are proposed. One based on a crude approximation to the proportional hazards model. The other subtracted the observed error probability and the program's success rate from one to estimate the remaining error rate.

[Naka89] Abstract: Many simple software errors are found in earlier software test phases. The ratio of complex errors to simple errors gradually increases with continual testing. This paper describes a software reliability model called the Error Complexity Model. In this model, errors are classified by error complexity which is a measure of error detectability. The number of remaining software errors is estimated from the ratio of complex to simple errors and the number of discovered errors. New criteria for error complexity classification are proposed. The model is evaluated and compared with existing models using actual error data.

[Naur69] Abstract: The paper describes a programming discipline, aiming at the systematic construction of programs from given global requirements. The crucial step in the approach is the conversion of the global requirements into sets of action clusters (sequences of program statements), which are then used as building blocks for the final program. The relation of the approach to proof techniques and to programming languages is discussed briefly.

[Nels78] Abstract: Recent work on software reliability associates correct execution of a test case with a statistical inference that the program will execute correctly for a specified subset of inputs. Test cases can be designed so that their associated subsets cover the entire input domain, allowing reliability estimates to be made for expected operational use profiles.

[Ng78] Summary: This paper reports a FORTRAN *post mortem* dump system (PMD) for the ICL *1900* computers. The system, jointly implemented by Birmingham and Liverpool Universities, can perform a core/storage dump in terms of the original FORTRAN source following the segment (subroutines, etc.) history of execution when the program fails to terminate successfully. The compilation overheads of the new system are very low and the execution overheads practically none.

[Nico87] Abstract: In this paper we consider the queueing analysis of a fault-tolerant computer system. The failure/repair behavior of the server is modeled by an irreducible continuous-time Markov chain. Jobs arrive in a Poisson fashion to the system and are serviced according to FCFS discipline. A failure may cause the loss of the work already done on the job in service, if any; in this case the interrupted job is repeated as soon as the server is

ready to deliver service. In addition to the delays due to failures and repairs, jobs suffer delays due to queueing. We present an exact queueing analysis of the system and study the steady state behavior of the number of jobs in the system. As a numerical example, we consider a system with two processors subject to failures and repairs.

[Noon75] Abstract: In the author's view structured programming consists of the use of the following: structure, abstraction, and specification. The purpose of this paper is to develop formal specifications for a nontrivial program in order to facilitate a proof of correctness. It is shown how the specifications serve as an abstraction for the program. A proof of correctness then consists of merely showing that the program at each level meets its formal specifications. Under this methodology lower levels of the program can be changed without affecting higher levels.

[Ntaf79] Abstract: In this paper various path cover problems, arising in program testing, are discussed. Dilworth's theorem for acyclic digraphs is generalized. Two methods for finding a minimum set of paths (minimum path cover) that covers the vertices (or the edges) of a digraph are given. To model interactions among code segments, the notions of required pairs and required paths are introduced. It is shown that finding a minimum path cover for a set of required pairs is NP-hard. An efficient algorithm is given for finding a minimum path cover for a set of required paths. Other constrained path problems are considered and their complexities are discussed.

[Ntaf81a] Abstract: In this paper, we introduce required element testing and report on an experimental comparison of this strategy with branch and random testing. The required element testing strategy studied here uses data flow information to generate a set of required elements for a program. The comparison with branch and random testing is performed using mutation analysis as a measure of test set adequacy.

[Ntaf81b] Abstract: Certain graph theoretic problems dealing with the testing of structured programs are treated. A structured digraph is a digraph that represents a structured program. A labeling procedure which characterizes structured digraphs is described. An efficient algorithm for finding a minimum path cover for the vertices of digraphs that belong to an important family of structured digraphs is given. To model interactions among code segments the notions of "required pairs" and "must pairs" are introduced and the corresponding constrained path cover problems are shown to be NP-complete even for acyclic structured digraphs.

[Ntaf82] Abstract: Two classes of program testing strategies are introduced that consist of specifying a set of required elements for the program and then covering those elements with appropriate test inputs. In general, a required element has a structural and a functional component and is covered by a test case if the test case causes the features specified in the structural component to be executed under the conditions specified in the functional component. Data flow analysis is used to specify the structural component, and data flow interactions are used as a basis for developing the functional component. The strategies are illustrated with examples and some experimental evaluations of their effectiveness are presented.

[Ntaf85] Abstract: In this paper we compare a number of structural testing strategies in terms of their relative coverage of the program's structure and also in terms of the number of test cases needed to satisfy each strategy. We also discuss some of the deficiencies of such comparisons.

[Offu87] Abstract: Mutation analysis is a powerful technique for testing software systems. In the Mothra project, conducted at Georgia Tech's Software Engineering Research Center, mutation analysis is used as a basis for building an integrated software testing environment. Mutation analysis requires the execution of many slightly differing versions of the same program to evaluate the quality of the data used to test the program. In the current version of the Mothra system, a program to be tested is translated to intermediate code, where it and its mutated versions are executed by an interpreter.

In this paper, we discuss some of the unique requirements of an interpreter used in a mutation-based testing environment. We then describe how these requirements affected the design and implementation of the Fortran 77 version of the Mothra interpreter. Other topics covered include the architecture of the interpreter and

many of the design elements that it incorporates. We also describe the intermediate language used by Mothra and the features of the interpreter that are needed for software testing.

[Ohba84] Abstract: This paper discusses improvements to conventional software reliability analysis models by making the assumptions on which they are based more realistic. In an actual project environment, sometimes no more information is available than reliability data obtained from a test report. The models described here are designed to resolve the problems caused by this constraint on the availability of reliability data. By utilizing the technical knowledge about a program, a test, and test data, we can select an appropriate software reliability analysis model for accurate quality assessment. The delayed S-shaped growth model, the inflection S-shaped model, and the hyperexponential model are proposed.

[Ohba89] Abstract: This paper discusses the improvement of conventional software reliability growth models by elimination of the unreasonable assumption that errors or faults in a program can be perfectly removed when they are detected. The results show that exponential-type software reliability growth models that deal with error-counting data could be used even if the perfect debugging assumption were not held, in which case the interpretation of the model parameters should be changed. An analysis of real project data is presented.

[Okad82] Abstract: In order to obtain a usable effort estimation for a small scale project, an earlier phase of CAD/CAM system software development was carefully studied. Upon analysis of data obtained, three additional attributes other than the number of source statements were taken into account as a basis for the effort estimation. They were 1) complexity, 2) personnel skill and 3) specification volatility. Subsequently, a set of models for the small project effort estimation was derived. Program size-effort correlations obtained were higher than 0.972. The models were then applied to the consecutive phases of the same project. The fitness between the estimated effort and the actual effort was satisfactory in a practical sense.

[Olde77] Abstract: Software science techniques have been used to provide a framework for evaluation of problem solving systems. In that effort, two methods for calculating the level of a language (L and L) were used; it was suspected that L, while adequate in that application, might be inferior to L. By using a set of hypothetical languages, each with different intrinsic data structures and operators, it is shown here that when an inappropriate language is applied to some problems, L may reflect an inaccurately large value for language level, and can sometimes be made to yield an arbitrary value. Since L is often as easily applied as L, and does not exhibit this anomalous behavior, it is suggested that its general use is to be preferred.

[Olde83] Abstract: This paper describes a technique for predicting the execution behavior of a source program or a software design specification. As a by-product of syntactic analysis, a program graph is constructed which can subsequently be treated as the graph of a finite automaton. The expression for execution behavior is the regular expression of the graph. Several simplification techniques for these expressions are discussed and exemplified. In particular, the substitution of known values for program segments followed by constant folding cannot be done indiscriminately; the allowable situations are characterized. Applications include the prediction of execution time for a program or a software design, other forms of language analysis, and program restructuring.

[Olen86] Abstract: This paper presents a flexible and general mechanism for specifying problems relating to the sequencing of events and mechanically translating them into dataflow analysis algorithms capable of solving those problems. Dataflow analysis has been used for quite some time in compiler code optimization. It has recently gained increasing attention as a way of statically checking for the presence or absence of errors and as a way of guiding the test selection process. Most static analyzers, however, have been custom-built to search for the fixed, and often quite limited, classes of dataflow conditions. We show that the range of sequences for which it is interesting and worthwhile to search is actually quite broad and diverse. We create a formalism for specifying this diversity of conditions. We then show that these conditions can be modeled essentially as dataflow analysis problems for which effective solution are known and further show how these solution can be exploited to serve as the basis for mechanical creation of analyzers for these conditions.

[Oste76a] Summary: This paper describes DAVE, a system for analyzing Fortran programs. DAVE is capable of detecting the symptoms of a wide variety of errors in programs, as well as assuring the absence of these errors. In addition, DAVE exposes and documents subtle data relations and flows within programs. The central analytic procedure used is a depth first search. DAVE itself is written in Fortran. Its implementation at the University of Colorado and some early experience are described.

[Oste76b] Abstract: This paper describes DAVE, an automatic program testing aid which performs a static analysis of Fortran programs. DAVE analyzes the data flows both within and across subprogram boundaries of Fortran programs, and is able to detect occurrences of uninitialized and dead variables in such programs. The paper shows how this capability facilitates the detection of a wide variety of errors, many of which are often quite subtle. The central analytic mechanism in DAVE is a depth-first search procedure which enables DAVE to execute efficiently. Some experiences with DAVE are described and evaluated and some future work is projected.

[Oste77] Abstract: An unfortunate characteristic of current static analysis algorithms is their apparent inability to distinguish between executable and unexecutable program paths. The definitive determination of executability of a given path has long been known to be unachievable. This paper presents some heuristics for detecting certain classes of unexecutable paths and preliminary findings tending to indicate that the heuristics can be expected to be rather effective. The heuristics are based upon the application of existing static data flow analysis algorithms and hence offer hope of coexisting with and guiding diagnostic and optimization scans which also use data flow analysis.

[Oste80] Abstract: This paper presents an approach to integrating four techniques for testing, analysis and verification into one overall strategy for incrementally raising the confidence in software in a cost-effective way. The paper summarizes the strengths, weaknesses, and operational characteristics of dynamic testing, static analysis, symbolic execution and formal verification. It uses a detailed example as an illustration. Next the integrated strategy is presented. Finally, there is a discussion of how this strategy can be used to raise confidence in software requirements and design specifications as well.

[Oste83] Abstract: This paper discusses the goals and methods of the Toolpack project and in this context discusses the architecture and design of the software system being produced as the focus of the project. Toolpack is presented as an experimental activity in which a large software tool environment is being created for the purpose of general distribution and then careful study and analysis. The paper begins by explaining the motivation for building integrated tool sets. It then proceeds to explain the basic requirements that an integrated system of tools must satisfy in order to be successful and to remain useful both in practice and as an experimental object. The paper then summarizes the tool capabilities that will be incorporated into the environment. It then goes on to present a careful description of the actual architecture of the Toolpack integrated tool system. Finally the Toolpack project experimental plan is presented, and future plans and directions are summarized.

[Oste84] Abstract: This paper presents a view of how various testing, analysis and debugging techniques can be integrated into a tool supported methodology. The paper is composed of two major components. In the first, the techniques are described in detail, compared and contrasted. An integrating methodology is proposed. The second component of the paper deals with Toolpack, a specific ensemble of tools having goals similar to those described in the first component, and IST, the Integrated System of Tools, an integration strategy for these tools. This second part of the paper indicates how Toolpack/IST could be configured into a system capable of implementing the integrated strategy of the first section in an efficient, effective way.

[Oste87] Abbreviated Introduction: In this paper we have suggested that the notion of a "process program" –namely an object which has been created by a development process, and which is itself a software process description–should become a key focus of software engineering research and practice. We believe that the essence of software engineering is the study of effective ways of developing process programs and of maintaining their effectiveness in the face of the need to make changes.

The main suggestions presented here revolve around the notion that process programs must be defined in a precise, powerful and rigorous formalism, and that once this has been done, the key activities of development and evolution of both process programs themselves and applications programs can and should be carried out in a more or less uniform way.

This strongly suggests the importance of devising a process programming language and a software environment capable of compiling and interpreting process programs written in that language. Such an environment would become a vehicle for the organization of tools for facilitating development and maintenance of both the specified process, and the process program itself. It would also provide a much needed mechanism for providing substantive support for software measurement and management.

[Ostr80] Overview: Testing is the most common way of gaining confidence in the correctness of software. Despite a long history of practical testing experience, it is only during the last five years that researchers have attempted to formulate a theoretical foundation for testing.

The initial steps in this direction where taken by Goodenough and Gerhart, who formulated an ambitious theory which described the conditions under which a program can be determined to be correct by testing.

The problems of a theory based on the concept of ideal tests are of three types: formal unsolvability, impracticality, and unrealistic assumptions. As we shall see, theories with less ambitious goals as well as various methodologies used in practice, must also face these problems.

[Ostr84] Abstract: A study has been made of the software errors committed during development of an interactive special-purpose editor system. This product, developed for commercial production use, has been followed during nine months of coding, unit testing, function testing, and system testing. Detected problems and their fixes have been described by testers and debuggers. A new fault categorization scheme was developed from these descriptions and used to classify the 173 faults that resulted from the project's errors. For each error, we asked the programmers to select its most likely cause, report the stages of the software development cycle in which the error was committed and the problem first noted, and the circumstances of the problem's detection and isolation, including time required, techniques tried, and successful techniques. The results collected in this study are compared to results from earlier studies, and similarities and differences are noted.

[Ostr86] Abstract: The overall goal of software testing is to expose errors that exist in program code. The specific goal of specification-based or black-box test case design is to create a series of test cases that fully exercise the functionality of the software. To achieve this goal, it is necessary to insure a systematic and comprehensive treatment of the specification. Such a treatment is particularly difficult when the specification is a large, evolving document, written in a natural language, and test case design is to be performed by a multi-person team. Even if only some of these factors are present, a strategy is needed to assure that the specification has been completely considered. As the size of the specification and the test team grows, the need for a tool to manage the process becomes more pressing. This paper describes a strategy for managing specification-based testing, proposes such a tool, and describes its use.

[Ostr88] Abbreviated Introduction: A method for creating functional test suites has been developed in which a test engineer analyzes the system specification, writes a series of formal test specifications, and then uses a generator tool to produce test descriptions from which test scripts are written. The advantages of this method are that the tester can easily modify the test specification when necessary, and can control the complexity and number of the tests by annotating the test specifications with constraints.

[Otte79] Abstract: A major portion of the problems associated with software development might be blamed on the lack of appropriate tools to aid in the planning and testing phases of software projects. As one step towards solving this problem, this paper presents a model to estimate the number of bugs remaining in the system at the beginning of the testing and integration phases of development. The model, based on software science metrics, was tested using data currently available in the literature. Extensions to the model are also presented which can be used to obtain such estimates as the expected amount of personnel and computer time required for project

validation.

[Otte81] Abstract: An earlier paper presented a model based on software science metrics to give quantitative estimate of the number of bugs in a programming project at the time validation of the project begins. In this paper, we report the results from an attempt to expand the model to estimate the total number of bugs expected during the total project development. This new hypothesis has been tested using the data currently available in the literature along with data from student projects. The model fits the published data reasonably well, however, the results obtained using the student data are not conclusive.

[Owic75] Abstract: A language for parallel programming, with a primitive construct for synchronization and mutual exclusion, is presented. Hoare's deductive system for proving partial correctness of sequential programs is extended to include the parallelism described by the language. The proof method lends insight into how one should understand and present parallel programs. Examples are given using several of the standard problems in the literature. Methods for proving termination and the absence of deadlock are also given.

[Owic76] Abstract: An axiomatic method for proving a number of properties of parallel programs is presented. Hoare has given a set of axioms for partial correctness, but they are not strong enough in most cases. This paper defines a more powerful deductive system which is in some sense complete for partial correctness. A crucial axiom provides for the use of auxiliary variables, which are added to a parallel program as an aid to proving it correct. The information in a partial correctness proof can be used to prove such properties as mutual exclusion, freedom from deadlock, and program termination. Techniques for verifying these properties are presented and illustrated by application to the dining philosophers problem.

[Owic82] Abstract: A liveness property asserts that program execution eventually reaches some desirable state. While termination has been studied extensively, many other liveness properties are important for concurrent programs. A formal proof method, based on temporal logic, for deriving liveness properties is presented. It allows a rigorous formulation of simple informal arguments. How to reason with temporal logic and how to use safety (invariance) properties in proving liveness is shown. The method is illustrated using, first, a simple programming language without synchronization primitives, then one with semaphores. However, it is applicable to any programming language.

[Paig72] Abbreviated Introduction: In this paper, we present a technique for applying some fundamental flow graph concepts to computer programs to yield some quantitative measurement of software complexity. Due to the lack of any complete testing facility, it is important to order or rank the priorities in which subroutines or portions of subroutines should be tested. In this manner, since all subroutines cannot be completely checked out, at least the more critical segments can be flagged for testing.

[Paig75] Abstract: Current interests in software engineering have posed serious questions about the evolution of programs and languages. Computer programs are not simply collections of statements; they involve specific structural relationships between the program elements. Program structure has been discussed as being an important influence on the ease with which programs can be constructed, verified, understood, and changed. The discipline of "structured programming" has been developed because computer scientists have sought to better control and understand the programming process.

Program graphs have been used as a vehicle to focus attention on the structure of a program. In this paper a systematic methodology for partitioning a program graph (digraph) to highlight the relationships between program elements is introduced along with an attendant notation. This notation is described in purely mathematical terms in the first section, and then the programming-related implications of this approach are addressed in the second section.

[Paig77b] Abstract: In recent years, applications of graph theory to computer software have given fruitful results and attracted more and more attention. A program graph is a graph structural model of a program

exhibiting the flow relation of connection among the elements (statements) in the program.

One particular aspect of graph analysis which is extremely useful for software is that of partitioning, since it both reduces the complexity of the system and highlights the actual system composition.

The purpose of this paper is to review and discuss given approaches to partitioning graphs. These techniques are best identified by the names of the units into which the program is grouped: segments, DD-paths, intervals, classes, and level-i paths. The objective here is to review these techniques on a fundamental level without exhausting all the uses and users of each approach.

[Paig78a] Abstract: This paper describes a quantitative software testing methodology for non-structured and structured programs. The paper first treats some of the recent work by McCabe and Paige which has developed the groundwork for a quantitative analysis of software testing. This perspective has set the stage for use of a program-graph basis as the thread for the software testing effort. A basis is a set of paths such that any other path in the graph can be expressed as a combination of paths in the basis. A technique for generating a unique, practical basis for a program-graph is introduced. The strategy for testing programs using this basis is discussed. The final section treats the simplifying effect of structured programs on this testing approach.

[Paig81] Abstract: A complete software testing process must concentrate on examination of the software characteristics as they may impact reliability. Software testing has largely been concerned with structural tests, that is, test of program logic flow. In this paper, a comparison software test technique for the program data called *data space testing* is described.

An approach to data space analysis is introduced with an associated notation. The concept is to identify the sensitivity of the software to a change in a specific data item. The collective information on the sensitivity of the program to all data items is used as a basis for test selection and generation of input values.

[Panz76] Abstract: A test procedure is a formal specification of test cases to be applied to one or more target program modules. Test procedures are executable. A process called the VERIFIER applies a test procedure to its target modules and produces an exception report indicating which test cases, if any, failed.

Test procedures facilitate thorough software testing by allowing individual modules or arbitrary groups of modules to be thoroughly tested outside the environment in which they will eventually reside. Test procedures are complete, self-contained, self-validating and execute automatically. Test procedures are a deliverable product of the software development process and are used for both initial checkout and subsequent regression testing of target program modifications.

Test procedures are coded in a new language called TPL (Test Procedure Language). The paper analyzes current testing practices, describes the structure and design of test procedures and introduces the Fortran Test Procedure Language.

[Panz78a] Abstract: Typical testing activities may involve many hundreds of tests. An automatic software test driver assists the tester by managing all of the test data, and automatically running the tests. Savings during regression testing can be significant.

[Panz78b] Abbreviated Introduction: The execution of software test cases and the verification of test results may be performed automatically by a new type of program called an automatic software test driver. When using an automatic test driver, a formal test procedure is coded in a special test language. The test procedure takes the place of the test data and test setup instructions of conventional testing, and control the automatic test driver. An automatic test driver applies one test procedure to all or part of a target program, executes all of the test cases specified in the test procedure, and verifies that the results of each test case are correct. This paper describes the Fortran Test Procedure Language (TPL/F) which was developed at General Electric and is used for specifying test procedures for Fortran software.

The specific goals of the TPL/F automatic test driver are as follows. The need for writing drivers and stubs for module and subsystem testing is eliminated since the TPL/F system can test any combination of one or more modules independently of the rest of the target program. The TPL/F test language provides a standard format for

specifying software tests and the test procedure processor provides a standard test execution setup. Since the formal test procedures specify the correct outcomes of test cases, the test procedure processor automates the verification of test execution results.

[Panz78c] An automatic software test driver is a new type of software tool which controls and monitors the execution of software tests. An automatic test driver is controlled by a formal test procedure coded in a special software test language. The test procedure replaces the test data and test setup instructions of conventional testing. The specific goals of automatic test drivers are to eliminate the need for writing drivers and stubs for module and subsystem testing, to provide a standard format and language for specifying software tests, to provide a standard execution setup for software tests, and to automate the verification of test execution results.

A test procedure contains input data to be supplied to the program under test and model outputs against which actual outputs of the target program are verified. Typically, ninety percent or more of the text of a test procedure consists of model outputs which must be revised each time the target program is modified. The TPL/2.0 automatic software test driver described in this paper automates both the initial generation and subsequent revision of test procedure model outputs.

[Parn72a] **Introduction:** In two earlier reports, we have suggested some techniques to be used producing software with many programmers. The techniques were especially suitable for software which would exist in many versions due to modifications in methods or applications. These techniques have been taught in an undergraduate course and used in an experimental project in that course. The purpose of this report is to describe the results that have been obtained and to discuss some conclusions which we have reached. The experiment was completely uncontrolled, the programmers generally inexperienced and poor, and the programming system used was not designed for the task. The numerical data presented below have no real value. We include them primarily as an illustration of the type of result that can be obtained by use of the techniques described in the earlier reports. We consider these results a drastic improvement over the state of the art. *Major* changes in a system can be confined to *well-defined, small,* subsystems. No intellectual effort is required in the final assembly or "integration" phase.

[Parn72b] This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

[Parn72c] **Abstract:** This paper presents an approach to writing specifications for parts of software systems. The main goal is to provide specifications sufficiently precise and complete that other pieces of software can be written to interact with the piece specified without additional information. The secondary goal is to include in the specification no more information than necessary to meet the first goal. The technique is illustrated by means of a variety of examples from a tutorial system.

[Parn74] **Abstract:** This paper discusses the use of the term "hierarchically structured" to describe the design of operating systems. Although the various uses of this term are often considered to be closely related, close examination of the use of the term shows that it has a number of quite different meanings. For example, one can find two different senses of "hierarchy" in a single operating system. An understanding of the different meanings of the term is essential, if a designer wishes to apply recent work in Software Engineering and Design Methodology. This paper attempts to provide such an understanding.

[Parn77] This paper discusses the role of formal and precise specifications in the methodical development of

software which we know to be correct. The differences between the general use of the work "specification" and the engineering use of that term are discussed. The software development tasks that we are undertaking require a "divide and conquer" approach that can only succeed if we have a precise way of describing the subproblems. It is shown how predicate transformers and abstract specifications can be used when design decisions are made. Two examples of the use of abstract specifications are described and detailed specifications are included.

[Parn78] Designing software to be extensible and easily contracted is discussed as a special case of design for change. A number of ways that extension and contraction problems manifest themselves in current software are explained. Four steps in the design of software that is more flexible are then discussed. The most critical step is the design of a software structure called the "uses" relation. Some criteria for design decisions are given and illustrated using a small example. It is shown that the identification of **minimal** subsets and **minimal** extensions can lead to software that can be tailored to the needs of a broad variety of users.

[Parn79] [The author has] have been asked to discuss the chapter "An Appraisal of Program Specifications," by Liskov and Berzins. Since it would appear that the authors and [the author] are in fundamental agreement on the purpose of program specifications, [the author] will say little about our common position and focus on the areas where [the authors] perception of the role of specifications seems to differ somewhat from that of the authors. Most of [the authors] comments are based on [his] experience in using both formal and informal program specifications in a variety of programming projects since 1970.

Interest in the topic of program specifications derives from _____ _____ the design of a software system is a large and complex task. It is important that the designers be able to record the intermediate design decisions _____ also useful to be able to evaluate the design decisions using _____ _____ established criteria. All of the concepts mentioned in the Liskov-Berzins _____ are tools for these purposes. My view of the way that formal _____ design decisions can be used during the program development process _____ in [1].

[Parn85] Abbreviated Introduction: This report comprises eight short papers that were completed while [the author] was a member of the Panel on Computing in Support of Battle Management, convened by the Strategic Defense Initiative Organization (SDIO). SDIO is part of the Office of the US Secretary of Defense. The panel was asked to identify the computer science problems that would have to be solved before an effective antiballistic missile (ABM) system could be deployed. It is clear to everyone that computers must play a critical role in the systems that SDIO is considering. The essays that constitute this report were written to organize [the author's] thoughts on these topics and were submitted to SDIO with [the author's] resignation from the panel.

[Parn88] Abbreviated Introduction: Under AECB Projects No. 2.127.1 and No. 2.127.2 members of the Department of Computing and Information Science of Queen's University [the authors] were asked to review the software being prepared to control the two shutdown systems for the nuclear reactors at the Darlington generating station. New Canadian nuclear generating stations have two shutdown systems, each independent of the other and both independent of the reactivity and process control system. Although earlier Ontario Hydro generating stations used computers for reactivity control, the shutdown systems had been kept as simple as possible and were built using hardwired logic. In the Darlington plant both shutdown systems (SDS-1 and SDS-2) will be controlled by computer systems. A significant factor in the reliability and safety of those systems will be the reliability and trustworthiness of the software.

[The authors] were asked to examine the software and software documentation for SDS-1 and SDS-2 to determine whether they meet appropriate standards and whether they could be certified to be sufficiently dependable for such a critical application. (i.e., whether the documentation would enable a detailed safety evaluation of the software to be carried out in a later project phase).

[Parr80] Abstract: A new model of the software development process is presented and used to derive the form of the resource consumption curve of a project over its life cycle. The function obtained differs in detail from the Rayleigh curve previously used in fitting actual project data. The main advantage of the new model is that it

relates the rate of progress which can be achieved in developing software to the structure of the system being developed. This leads to a more testable theory, and it also becomes possible to predict how the use of structured programming methods may alter patterns of life cycle resource consumption.

[Pate89] Abstract: A key factor in the acceptance of high level programming languages has been the development of a comprehensive set of tools to support the user. If formal languages for specification are to achieve the same level of acceptance, they too will require extensive automated support. This paper describes a set of prototype tools which are designed to assist the developer in the use of formal specification techniques.

[Payt82] Abstract: This paper describes a system of automated tools for program generation. These tools translate formal specifications of design into efficient programs to perform the stated task. Compiler generation techniques are applied to create a general system that is applicable to the development of a wide range of software products. Usage of this system formalizes the software development process thus promoting a decrease in software design and development costs and easing the maintenance process. The software process is not bound to a particular implementation language thus software portability is enhanced.

[Pear84] Abbreviated Preface: This book is about *heuristics*, popularly known as rules of thumb, educated guesses, intuitive judgments or simply *common sense*. In more precise terms, heuristics stand for strategies using readily accessible though loosely applicable information to control problem-solving processes in human beings and machine. This book presents an analysis of the nature and the power of typical heuristic methods, primarily those used in artificial intelligence (AI) and operations research (OR) to solve problems of search, reasoning, planning and optimization on digital machines.

The discussions in this book follow a three-phase pattern: Presentation, characterization, and evaluation. We first present a set of general-purpose problem-solving strategies guided by heuristic information, then highlight the general principles and properties that characterize this set and, finally, we present mathematical analyses of the performances of these strategies in several well-structured domains. Some psychological aspects of how people discover and use heuristics are discussed briefly.

[Perk86] Abstract: Metrics researchers are currently in the early stages of validating the relationship between metrics and the quality problems encountered by users and developers of software. In order to establish these relationships, large amounts of data defined for validating specific metrics must be collected. Before performing such costly validation, we believe the metrics should be evaluated with respect to whether they reflect our current understanding of quality principles. Our preliminary attempt at validation focuses on a human vs. automated approach to analyzing an existing Ada program. The program consists of fourteen packages and approximately 150 procedures and functions. Segments of this code were selected and analyzed with respect to the software quality sub-criteria of flow simplicity, limited visibility, and error prevention and detection. The study focuses on disagreements between human and automated analysis, and attempts to explain those discrepancies and suggest possible ways to improve both measurement techniques and the quality of the software program analyzed.

[Perk87] Abstract: Our investigation applies an automated, hierarchical, Ada-specific software metrics framework to Navy-supplied Ada software to determine the effectiveness of such a framework as an aid to improving the quality of Ada software.

The metrics framework measures six software criteria and consists of approximately 150 software metric elements, where each metric element relates a software quality principle to the use of specific features of the Ada language.

The investigation involves: 1) analysis of the metric scores for the Navy-supplied Ada code, 2) modifications of the Ada code to correct the quality problems indicated by the metric scores, resulting in two improved versions of the code (the first incorporates only statement-by-statement changes and the second incorporates changes to the overall organization of the code), and 3) comparison of metric scores for the three versions of the Ada code.

[Perr83] Table of Contents: Establishing a test methodology. Establishing a system test policy, life cycle testing approach. Testing an application system test plan. Developing an application system test plan, testing techniques, testing tools, requirements phase testing, design phase testing, program phase testing, test phase testing, installation phase testing, maintenance phase testing, testing documentation. Assessing test performance. Evaluating the effectiveness of testing. Testing tools. Testing metrics. Bibliography.

[Perr86] Table of Contents: Will the computer do what I want to do? What can go wrong with computerized applications, and what to do about it, testing business fit, testing system fit, testing people fit. Does the software work correctly? Developing a test plan, creating testing conditions, verifying the correctness of the software functions. So now the software is in operation! Validation computer-produced output. Glossary. Golden rules. Index.

[Pesc85] Abstract: For the validation of the kernel system calls of a family of UNIX systems a knowledge based test environment was conceived. A prototype version is currently implemented in Prolog. The knowledge base consists essentially of three parts:

- test case specifications of the various system calls.
- a test suite generator with predicates including information about UNIX system properties and sound test practices, and
- a test protocol archive including utilities to extract and prepare reports about the test results.

All information in the knowledge base is stored as Horn clauses, i.e. facts and rules immediately to be consulted and executed by a Prolog interpreter.

[Pete77] Abstract: Over the last decade, the Petri net has gained increased usage and acceptance as a basic model of systems of asynchronous concurrent computation. This paper surveys the basic concepts and uses of Petri nets. The structure of Petri nets, their markings and execution, several examples of Petri net models of computer hardware and software, and research into the analysis of Petri nets are presented, as are the use of the reachability tree and the decidability and complexity of some Petri net problems. Petri net languages, models of computation related to Petri nets, and some extensions and subclasses of the Petri net model are also briefly discussed.

[Pets85] Introduction: Selecting test cases for system testing of the PICS/DCPR database application poses a fundamental problem. Due to the size of the system, methodologies described in the literature do not apply. Their formulations of "thorough testing" require so many test cases that they are not practical for system testing.

To deal with this problem, we have adopted an approach to test case selection that uses a simple set of priority rules to judge which test cases are more important than others. These priority rules derive from the charter of the system test group in the PICS/DCPR project, observations about developer testing, and the consequences of different types of software defects on users.

These practical priority rules are believed to constitute a more realistic approach to system-testing large database applications than current theory does.

[Pimo75] Abstract: This paper deals with the problem of assessing the reliability of programs written using structured programming techniques and having undergone a certain amount of testing. A program is said to be verified if, for a given set of tests it can be shown that every case of interest has been tested. As this end is, however, unattainable, we will consider, in the following, that a program is verified if one can prove that all the logic paths in the program flow graph have been traversed. Therefore, we will consider that a certain degree of verification is attained with a given set of tests, according to the number of paths actually traversed. This degree of verification, which is a non-decreasing function of the number of tests can be considered as an assessment of program reliability. The degree of verification attained through experiments can then be deduced from the images of experiments in the program flow graph. This paper defines a practical procedure to perform such an evaluation.

[Pipp78] Abbreviated Introduction: Many complex systems such as those found in a computer or a telephone

exchange are constructed by interconnecting a large number of simple components. The complexity of these systems arises from the number of components and the intricacy of their interconnections, rather than from any great complexity of the components themselves. The systems formed in this fashion are somehow much greater than the sum of their parts.

It is natural to assume that every component in a complex system is there for a reason, but although it may be true that the removal of any component would cause the system to malfunction, it is also possible that an overall reorganization would lead to a working system with many fewer components.

Complexity theory seeks to determine the minimum number of components needed for these systems. It pursues this goal in two ways: by finding new designs that call for fewer components and by showing that a certain number of components will be needed no matter what design is followed. Finding new designs for a system has an obvious practical significance: it can increase the efficiency of the system and reduce its cost. The second type of investigation, which sets limits beyond which further attempts at improvement are futile, is equally necessary for a complete understanding of a particular system and is often much harder to accomplish.

[Piwo82] Abstract: Two well-publicized program complexity measures are software science and cyclomatic complexity. Three areas where these measures do not always follow our intuitive notions of complexity are: structured vs unstructured programs, nested vs sequential predicates, and the use of case statements. This paper defines a nesting level complexity measure that punishes unstructuredness, and the nesting of predicates, and rewards the use of case statements. Examples are given where the nesting level complexity agrees with intuitive rankings of program structures where software science, cyclomatic complexity, and their suggested refinements do not.

[Pooc74] Abstract: Decomposition and conversion algorithms for translating decision tables are surveyed and contrasted under two broad categories: the mask rule technique and the network technique. Also, decision table structure is briefly covered, including checks for redundancy, contradiction, and completeness; decision table notation and terminology; and decision table types and applications. Extensive literature citations are provided.

[Popk78] Abstract: This report discusses the use of flowchart graphs, adjacency matrices, and zero-one linear programming to find the minimum number of tests necessary to execute every segment of a computer program at least once. The methods of Lipow are used as the basis for determining the maximum incomparable set, i.e., the largest set of program segments through which one and only one test case should pass. The size of the maximum incomparable set gives the minimum number of tests necessary to execute each segment at least once, while the elements of this set give the paths of each test. The report develops methods for finding the maximum incomparable set for loopless and some elementary looping flowcharts.

[Post87] Introduction: In 1984, L&N authorized a software engineering project to create a real-time, process-monitoring program that would be embedded in a large process-control system. This undertaking was named the Process Information Management Subsystem (PIMS) Trending Project. Before undertaking the Trending Project, L&N was reporting failure-density factors near 1.3. The subsequent drop to 0.072 represented a 95-percent improvement in software quality.

As well as quality improvements, we measured productivity increases on the Trending project compared to earlier L&N projects.

When the Trending Project software was delivered, the engineers had produced 29 lines of source code per staff-day. Even allowing for a substantial error margin in the estimates for productivity factors, the gain was more than 200 percent.

How were these quality and productivity improvements achieved? The PEI Testing Methodology – an integrated set of policies, techniques, metrics, and standards – was added to the L&N quality-assurance program. The methodology concentrates on five improvement techniques: (1) defining requirements for testability, (2) designing software for testability, (3) designing tests for most-probable errors (see Software Standards in May, July, and this issue), (4) designing tests before code is designed, and (5) performing reviews (inspections and walkthroughs).

In recent years, all the techniques included in the methodology have been studied one by one. Based on these studies, software researchers predicted that when all these techniques or modern programming practices were used together, software productivity increases as high as 50 percent were possible. This case study shows that the synergistic effect of combining techniques can be even more beneficial than anticipated.

[Pout87] This paper presents two Ada testing tools that have been developed in Nokia Information Systems, Softplan. Their main properties are described. It will be shown that these tools have a considerable potential in increasing the efficiency of testing (which is separate from debugging). These tools have been written in Ada and they are fully independent of the Ada compilation system and the operatoring system where they are used. This paper reveals also some of the basic idea how this kind of portability has been reached as well as some experiences in this respect.

[Prat80] Abstract: A vigorous approach to evaluating computer models is presented. With a concise, 21 question worksheet as a basis, logical criteria are developed for determining whether a model is: (1) safe for operational use by managers, (2) in need of further validation, or (3) of value only in providing valuable lessons for future modeling work. A numeric figure of merit reflecting significant aspects of the cost/benefit picture of the model is then developed as a guide for determining which models should be further developed or implemented.

[Prat87] Abstract: A new software testing strategy is described. The strategy is "adaptive" in that previous test paths (inputs) are used as a guide in the selection of subsequent paths (inputs). Preliminary implementations have successfully exploited the method's inherent user-interactive capability. The method ensures branch coverage, requires only "order n" tests (n being the number of decision nodes in the program flowgraph), and offers considerable advantages over existing strategies in its computational requirements.

[Press83] Abstract: Software metrics (or measurements) which predict software quality were extended from previous research to include two additional quality factors: interoperability and reusability. Aspects of requirements, design, and source language programs which could affect these two quality factors were identified and metrics to measure them were defined. These aspects were identified by theoretical analysis, literature search, interviews with project managers and software engineers, and personal experience.

A guidebook for software quality measurement was produced to assist in setting quality goals, applying metrics and making quality assessments.

[Prob82c] Abstract: A standard technique for monitoring software testing activities is to instrument the module under test with counters or probes before testing begins; then, during testing, data generated by these probes can be used to identify portions of as yet unexercised code. In this paper the effect of of the disciplined use of language features for explicitly delimiting control flow constructs is investigated with respect to the corresponding ease of software instrumentation. In particular, assuming all control constructs are explicitly delimited, for example, by END IF or equivalent statements, an easily programmed method is given for inserting a minimum number of probes for monitoring statement and branch execution counts without disrupting source code structure or paragraphing. The use of these probes, called statement probes, is contrasted with the use of standard (branch) probes for execution monitoring. It is observed that the results apply to well-delimited modules written in a wide variety of programming languages, in particular, Ada.

[Prob84] Abstract: A testing-based approach for constructing and refining very high-level software functionality representations such as intentions, natural language assertions, and formal specifications is presented and applied to a standard line-editing problem as an illustration. The approach involves the use of specification-based (black-box) testcase generation strategies, high-level specification formalisms, redundant or parallel development and cross validation, and a logic programming support environment. Test-case reference sets are used as software functionality representations for the purposes of cross validating two distinct high-level representations, and identifying ambiguities and omissions in those representations. In fact, we propose the use of successive refinements of such test reference sets as the authoritative specification throughout the software development

process. Potential benefits of the approach include improvements in user/designer communication over all life cycle phases, and an increase in the quality of specifications and designs.

[Prot88] Abstract: This study presents results of a software reliability experiment that investigates the feasibility of a new error detection method. The method can be used as an acceptance test and is solely based on empirical data about the behavior of internal states of a program. The experimental design uses the existing environment of multi-version experiment previously conducted at the NASA Langley Research Center, in which the 'launch interceptor' problem is used as a model problem. This allows the controlled experimental investigation of versions with well-known single and multiple faults, and the availability of an oracle permits the determination of the error detection performance of the test. Fault-interaction phenomena are observed that have an amplifying effect on the number of error occurrences. Preliminary results indicate that all faults examined so far are detected by the acceptance test. This shows promise for further investigations, and for the employment of this test method in other applications.

[Purd72] Abstract: A fast algorithm is given to produce a small set of short sentences from a context free grammar such that each production of the grammar is used at least once. The sentences are useful for testing parsing programs and for debugging grammars (finding errors in a grammar which causes it to specify some language other than the one intended). Some experimental results from using the sentences to test some automatically generated simple LR(1) parsers are also given.

[Putn78] Abstract: Application software development has been an area of organizational effort that has not been amenable to the normal managerial and cost controls. Instances of actual costs of several times the initial budgeted cost, and a time to initial operational capability sometimes twice as long as planned are more often the case than not.

A macromethodology to support management needs has now been developed that will produce accurate estimates of manpower, costs, and times to reach critical milestones of software projects. There are four parameters in the basic system and these are in terms managers are comfortable working with –effort, development time, elapsed time, and a state-of-technology parameter.

The system provides managers sufficient information to assess the financial risk and investment value of a new software development project before it is undertaken and provides techniques to update estimates from the actual data stream once the project is underway. Using the technique developed in the paper, adequate analysis for decisions can be made in an hour or two using only a few quick reference tables and a scientific pocket calculator.

[Putn79] Overview: Few managers are able to predict the time and resources needed to develop large-scale software systems. Progress is often measured by the rate of expenditure of resources rather than by some count of accomplishments. Unrealistic estimates often result in last minute efforts to get code written quickly, resulting in cost overruns and poor quality software.

Software development can be brought under control. It requires an understanding of how application software behaves, what factors management can control and what factors are limited by the process itself.

The basis of effective management is the fact that the software development process exhibits a characteristic behavior, which can be exploited, so that the expensive results of unrealistic approaches can be avoided.

[Quir85] Preface: Real-time software poses serious problems. It fails too often and the failures can be both extremely troublesome and sometimes dangerous.

In this report, the techniques available for validation and verification of real-time systems software are reviewed. Material, which is at present scattered through conference proceedings, research notes and journal papers, is gathered together and presented in the context of practical usefulness. More detailed references are included wherever possible.

[RADC76a] Abstract: A study of software errors is presented. Techniques for categorizing errors according to

type, identifying their source, and detecting then are discussed. Various techniques used in analyzing empirical error data collected from four large software systems are discussed and results of analysis are presented. Use of results to indicate improvements in the error prevention and detection processes through use of tools and techniques is also discussed.

A survey of software reliability models is included, and recent work on TRW's Mathematical Theory of Software Reliability (MTSR) is presented.

Finally, lessons learned in conjunction with collecting software data are outlined, with recommendations for improving the data collection process.

[Rabi77] Abstract: The framework for research in the theory of complexity of computations is described, emphasizing the interrelation between seemingly diverse problems and methods. Illustrative examples of practical and theoretical significance are given. Directions for new research are suggested.

[Rama75a] Abstract: In the past few years, research has been actively carried out in an attempt to improve the quality and reliability of large-scale software systems. Although progress has been made on the formal proof of program correctness, proving large-scale software systems correct by formal proof is still many years away. Automated software tools have been found to be valuable in improving software reliability and attacking the high cost of software systems. This paper attempts to describe some main features of automated software tools and some software evaluation systems that are currently available.

[Rama76] Abstract: Software validation through testing will continue to be a very important tool for ensuring correctness of large scale software systems. Automation of testing tools can greatly enhance their power and reduce testing cost. In this paper, techniques for automated test data generation are discussed. Given a program graph, a set of paths are identified to satisfy some given testing criteria. When a path or program segment is specified, symbolic execution is used for generating input constraints which define a set of inputs for executing this path or segment. Problems encountered in symbolic execution are discussed. A new approach for resolving array references ambiguities and a procedure for generating test inputs satisfying input constraints are proposed. References to arrays are recorded in a table during symbolic execution and ambiguities are resolved when test data are generated to evaluate the subscript expressions. The implementation of a test data generator for Fortran programs incorporating these techniques is also described.

[Rama81] Abstract: This paper discusses the necessity of a good methodology for the development of reliable software, especially with respect to the final software validation and testing activities. A formal specification development and validation methodology is proposed. This methodology has been applied to the development and validation of a pilot software, incorporating typical features of critical software for nuclear power plant safety protection. The main features of the approach include the use of a formal specification language and the independent development of two sets of specifications. Analyses on the specification consists of three parts: validation against the functional requirements, consistency and integrity of the specifications, and dual specification comparison based on a high-level symbolic execution technique. Dual design, implementation, and testing activities are developed to support the methodology. These include the symbolic executor and test data generator/dual program monitor system. The experiences of applying the methodology to the pilot software are discussed, and the impact on the quality of the software is assessed.

[Rama82] Abstract: It is essential to assess the reliability of digital computer systems used for critical real-time control applications (e.g., nuclear power safety control systems). This involves the assessment of the design correctness of the combined hardware/software system as well as the reliability of the hardware. In this paper we survey methods of determining the design correctness of systems as applied to computer programs.

Automated program proving techniques are still not practical for realistic programs. Manual proofs are lengthy, tedious, and error-prone. Software reliability provides a measure of confidence in the operational correctness of the software. Since the early 1970's several software reliability models have been proposed. We classify and discuss these models using the concepts of residual error size and the testing process used. We also

discuss methods of estimating the correctness of the program and adequacy of the set of tests used.

These methods are directly applicable to assessing the design correctness of the total integrated hardware/software system which ultimately could include large complex distributed processing systems.

[Rand75] Abstract: This paper presents and discusses the rationale behind a method for structuring complex computing systems by the use of what we term "recovery blocks," "conversations," and "fault tolerant interfaces." The aim is to facilitate the provision of dependable error detection and recovery facilities which can cope with errors caused by residual design inadequacies, particularly in the system software, rather than merely the occasional malfunctioning of hardware components.

[Rapp80] Abstract: This paper examines a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. It is argued that currently used path selection criteria which examine only the control flow of a program are inadequate. Our procedure associates with each point in a program at which a variable is defined, those points at which the value is used. Several related path criteria, which differ in the number of these associations needed to adequately test the program, are defined and compared.

[Redw83] Abstract: A systematic approach to test data design is presented based on both practical translation of theory and organization of professional [XXX]. The approach is organized around five domains and achieving coverage (exercise) of them by the test data. The domains are processing functions, input, output, interaction among functions, and the code itself. Checklists are used to generate data for processing functions. Separate checklists have been constructed for eight common business data processing functions such as editing, updating, sorting, and reporting. Checklists or specific concrete directions also exist for input, output, interaction, and code coverage. Two global heuristics concerning all test data are also used. A limited discussion on documenting test input data, expected results, and actual results is included.

Use, applicability, and possible expansions are covered briefly. Introduction of the method has similar difficulties to those experienced when introducing any disciplined technique into an area where discipline was previously lacking. The approach is felt to be easily modifiable and usable for types of systems other than the traditional business data processing ones for which it was originally developed.

[Reif75] Abstract: Recent investigations on the use of automation to realize the twin objectives of cost reduction and reliability improvement for computer programs developed for the U.S. Air Force are reported. The concepts of reliability and automation as they pertain to software are explained. Then, over twenty automated tools and techniques (aids) identified in this investigation are described and categorized. Based on the information reviewed, an assessment of the state of the technology is made. Finally, specific recommendations which try to give direction to future efforts are offered.

[Reif79a] Abstract: This concept paper discusses the possible use of failure modes and effects analysis (FMEA) as a means to produce more reliable software. FMEA is a fault avoidance technique whose objective is to identify hazards in requirements that have the potential to either endanger mission success or significantly impact life-cycle costs. FMEA techniques can be profitably applied during the analysis stage to identify potential hazards in requirements and design. As hazards are identified, software defenses can be developed using fault tolerant or self-checking techniques to reduce the probability of their occurrence once the program is implemented. Critical design features can also be demonstrated a priori analytically using proof of correctness techniques prior to their implementation if warranted by cost and criticality.

[Reif79b] Abstract: Software tools can serve as powerful aids in the design, development, test, and maintenance of computer software. So, in light of the recent growth in cost of software relative to total system cost, it should come as no surprise that the subject of software tools has sparked a good deal of interest throughout the computer industry. In response to that interest, this paper provides a comprehensive listing of the software tools and techniques currently available.

We first describe the typical software life cycle and its three major stages: (1) conceptual and requirements; (2) development; and (3) operations and maintenance. Next we describe the six categories of software tools (simulation, development, test and evaluation, operations and maintenance, performance measurement, and programming support). Table 1 relates the life cycle areas to the various categories of software tools.

[Reyn87] Abbreviated Introduction: The Partial Metrics System [to support the metrics-driven] implementation of individual modules in a large-scale programming project design is explained, with an emphasis on the refinement process. A model, with its three phases, shows that the pseudocode refinement process can be monitored in partial metric terms.

[Reyn89] Abstract: This paper describes a software tool, the partial metrics system (PMS), that supports the metrics-driven design of pseudocode program modules. Although this is a generic approach that is language independent, we illustrate its application using Ada as the target language. Each new refinement of a pseudocode program is assessed in terms of a set of partial metrics. These metrics are extensions of Halstead's *Software Science*, McCabe's *Cyclomatic Complexity*, and others.

It is then demonstrated how these metrics can drive the design process for an individual module. Heuristics are suggested that can allow the programmer to make use of these metrics in order to produce improved designs.

[Rich81a] Abstract: A major drawback of most program testing methods is that they ignore program specifications, and instead base their analysis solely on the information provided in the implementation. This paper describes the partition analysis method, which assists in program testing and verification by evaluating information from both a specification and an implementation. This method employs symbolic evaluation techniques to partition the set of input data into procedure subdomains so that the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. The partition divides the procedure domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data and to verify consistency between the specification and the implementation. Moreover, the test data selection process, called partition analysis testing, and the verification process, called partition analysis verification, are used to enhance each other, and thus increase program reliability.

[Rich82] Abstract: The partition analysis method compares a procedure's implementation to its specification. In addition to verifying consistency between the two, this comparison is used to derive test data. Unlike most test data selection strategies, which consider only the implementation, partition analysis selects test data that characterize the procedure in terms of its intended behavior as well as the structure of its implementation. To accomplish this, partition analysis divides or "partitions" the procedure's domain into subdomains in which all elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. Initial experimentation has shown that through the integration of testing and verification, as well as through the use of information derived from both the implementation and the specification, the partition analysis method is effective for determining program reliability. This paper provides an overview of the partition analysis method and reports the results obtained from preliminary evaluation of its effectiveness.

[Rich85a] Abbreviated Introduction: Several of the validation tools being developed employ a method called *symbolic evaluation*, which creates a symbolic representation of the program. This chapter describes symbolic evaluation and surveys some of the testing applications of this method.

Symbolic evaluation monitors the manipulations performed on the input data. Computations and their applicable domain are represented algebraically over the input domain, thereby describing the relationship between the input data and the resulting values. Normal execution computes numerical values but loses information about the way in which these numerical values were derived, whereas symbolic evaluation preserves this information. When further analyzed, this information provides the basis for several testing techniques.

For the most part, current testing research is directed at either the problem of determining the paths (the particular sequences of statements) that must be tested or the problem of selecting revealing test data for the

selected paths. For the path selection problem, techniques such as program coverage, data flow testing, and perturbation testing have been proposed. For the test data selection problem, a number of informal guidelines have been put forth. Recently there has been considerable work on developing more systematic test data selection techniques that can either eliminate certain classes of errors or provide a quantifiable error bound. Many of the current path selection and test data selection techniques base their analyses on the information provided by symbolic evaluation.

The above testing techniques are referred to as structural techniques, since they base their analysis solely on the information provided by a given implementation. There are two drawbacks to such an approach. First, it ignores the information that may be available from a specification. Second, it delays testing until the implementation is complete, thereby not detecting errors in the most timely and cost-effective manner. Research efforts that use symbolic evaluation to assist in solving both these problems are currently underway. Specification-guided program testing techniques use the information provided by symbolic evaluation of a specification to guide in the testing of its implementation, while specification testing techniques employ symbolic evaluation to actually test a specification.

The next section of this chapter provides a brief overview of symbolic evaluation, with an example to demonstrate the method. The third section describes a number of ways in which symbolic evaluation of a program aids the path selection and test data selection aspects of testing. The fourth section describes the use of symbolic evaluation for specification-guided program testing and specification testing.

[Rich85b] Abstract: The partition analysis method compares a procedure's implementation to its specification, both to verify consistency between the two and to derive test data. Unlike most verification methods, partition analysis is applicable to a number of different types of specification languages, including both procedural and nonprocedural languages. It is thus applicable to high-level descriptions as well as to low-level designs. Partition analysis also improves upon existing testing criteria. These criteria usually consider only the implementation, but partition analysis selects test data that exercise both a procedure's intended behavior (as described in the specifications) and the structure of its implementation. To accomplish these goals, partition analysis divides or **partitions** a procedure's domain into subdomains in which all elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. This partition divides the procedure domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data and to verify consistency between the specification and the implementation. Moreover, the testing and verification processes are designed to enhance each other. Initial experimentation has shown that through the integration of testing and verification, as well as through the use of information derived form both the implementation and the specification, the partition analysis method is effective for evaluating program reliability. This paper describes the partition analysis method and reports the results obtained from an evaluation of its effectiveness.

[Rich87a] Abstract: RELAY, a model for error detection, defines *revealing conditions* that guarantee that a fault *originates* an error during execution and that the error *transfers* through computations and data flow until it is *revealed*. This model of error detection provides a fault-based criterion for test data selection. The model is applied by choosing a fault classification, instantiating the conditions for the classes of faults, and applying them to the program being tested. Such an application guarantees the detection of errors caused by any fault of the chosen classes. As a formal model of error detection, RELAY provides the basis for an automated testing tool. This paper presents the concepts behind RELAY, describes why it is better than other fault-based testing criteria, and discusses how RELAY could be used as the foundation for a testing system.

[Rich88a] Abbreviated Introduction: This paper reports on a new model of error detection called RELAY, which provides a fault-based criterion for test data selection. The RELAY model builds upon the testing theory introduced by Morell, where an error is "created" when an correct state is introduced at some fault location, and it is "propagated" if it persists to the output. We refine this theory by more precisely defining the notion of when an error is introduced and by differentiating between the persistence of an error through computations and its persistence through data flow operations. We introduce similar concepts, origination and transfer, as the first erroneous evaluation and the persistence of that erroneous evaluation, respectively.

[Ridd78] Abstract: A modeling scheme is presented which provides a medium for the rigorous, formal, and abstract specification of large-scale software system components. The scheme allows the description of component behavior without revealing or requiring the description of a component's internal operation. Both collections of sequential processes and the data objects which they share may be described. The scheme is of particular value during the early stages of software system design, when the system's modules are being delineated and their interactions designed, and when rigorous, well-defined specification of undesigned components allows formal and informal arguments concerning the design's correctness to be formulated.

[Roac80] Abstract: Software cost estimation techniques reported in recent literature are compared. Six cost estimation techniques are described in a common notation. An example is worked through all the techniques to illustrate similarities and differences.

[Roby85] Abbreviated Forward: These Proceedings of the First Workshop on Formal Specification and Verification of Ada, held at the Institute for Defense Analyses, are composed in part of papers and slides supplied by the speakers, and in part of summaries of the talks and discussions edited from recordings made of the Workshop.

The purpose of this initial two-and-a-half day Workshop was to identify current issues in Ada verification and to decide what could be done to improve current understanding and practice of Ada software verification. Since verification impacts not only coding activities but all development activities, it is desirable that many groups be kept informed about the progress of these Workshops.

[Roe87] Abstract: Several inequalities are derived for use in certifying function subroutines by means of black box testing. It is assumed that a function is approximated by means of a polynomial of limited degree on a closed interval. These inequalities give upper bounds on the error measured over a finite sample and known properties of the function.

[Romb84] Abstract: This paper describes results of a study to develop maintenance metrics based on structural software design characteristics. The intent of the study was to define a characteristic metric set, suited to explain and predict software maintenance behavior. The maintenance aspects investigated in this study are stability and modifiability. While stability addresses the average number of modules affected per change cause, modifiability characterizes the ease with which changes can be made within each of these modules. Additional interest is dedicated to the difference between characteristic design and implementation metric sets, and to the difference between change behavior during development and maintenance. This study examines the development of six software systems and controlled maintenance experiments using these systems.

[Romb87a] Abstract: This paper describes a study on the impact of software structure on maintainability aspects such as comprehensibility, locality, modifiability, and reusability in a distributed system environment. The study was part of a project at the University of Kaiserslautern, West Germany, to design and implement LADY, a LAnguage for Distributed sYstems. The study addressed the impact of software structure from two perspectives. The language designer's perspective was to evaluate the general impact of the set of structural concepts chosen for LADY on the maintainability of software systems implemented in LADY. The language user's perspective was to derive structural criteria (metrics), measurable from LADY systems, that allow the explanation or prediction of the software maintenance behavior. A controlled maintenance experiment was conducted involving twelve medium-size distributed software systems; six of these systems were implemented in LADY, the other six systems in an extended version of sequential Pascal. The benefits of the structural LADY concepts were judged based on a comparison of the average maintenance behavior of the LADY systems and the Pascal systems; the maintenance metrics were derived by analyzing the interdependence between structure and maintenance behavior of each individual LADY system.

[Rose84] Abstract: This paper describes a methodology for the design of a class of Ada software tools which perform source-to-source transformation of Ada programs. The tools perform the transformations on the

DIANA representation of an input source program using a package of templates which are the DIANA representation of source program textual insertions.

Following a brief overview of DIANA, the environment required by these tools is described; a typical environment consists of an implementation of DIANA and a set of utility programs. Next, the paper describes a "skeleton" program which is used to implement a tool; the tool skeleton is a recursive DIANA tree traversal program which is expanded incrementally with code to perform a set of specific transformations. The next portion of the paper gives a detailed description of the design methodology; the methodology provides for mapping a source-level specification of a transformation tool to a DIANA-level specification, which serves as an implementation guide for the tool. Finally, a description is given of an application of the design methodology, a preprocessor for the task monitoring system described by Helmbold and Luckham.

To conclude the paper, a summary of the advantages and suggested applications of the design methodology is presented. The major advantage of the methodology is that it allows the transformations performed by a tool to be implemented and tested incrementally, making debugging less complex and implementation more efficient.

[Rose85a] Abbreviated Introduction: This article describes a methodology for the design of Ada transformation tools using the DIANA representation of Ada source program input. The methodology was tested on the implementation of the task monitor preprocessor of Helmbold and Luckham and proved quite an effective way to implement an Ada tool. A tools designed according to the methodology requires an environment of support programs and packages for maintaining DIANA trees. Each tool is an expanded version of a very simple program called a tool skeleton, which is nothing more than a **case** statement that recursively traverses a DIANA tree.

[Rose85b] Abstract: Computer systems have become an integral part of most organizations. The need to provide continuous, correct service is becoming more critical. However, decentralization of computing, inexperienced users, and larger more complex systems make for operational environments that make it difficult to provide continuous, correct service. This document is intended for the computer system manager (or user) responsible for the specification, measurement, evaluation, selection or management of a computer system.

This report addresses the concepts and concerns associated with computer system reliability. Its main purpose is to assist system managers in acquiring a basic understanding of computer system reliability and to suggest actions and procedures which can help them establish and maintain a reliability program. The report presents discussions on quantifying reliability and assessing the quality of the computer system. Design and implementation techniques that may be used to improve the reliability of the system are also discussed. Emphasis is placed on understanding the need for reliability and the elements and activities that are involved in implementing a reliability program.

[Ross88] Abstract: This report discusses the role of Management Indicators in validating the predictive capability of the bottom-up evaluation process, which is defined by the Procedural Approach to the Evaluation of Software Development Methodologies. The bottom-up evaluation process provides a framework for determining the extent to which software engineering objectives, e.g., reliability and maintainability, are present in a software product from a design perspective of the code and supporting documentation. The bottom-up evaluation process is observed to be a predictor of the extent to which the objectives are realized in the post-developed product.

Employment of the bottom-up evaluation process to determine the extent to which the objectives are present in the product is accomplished by the utilization of Design Indicators. Management Indicators are proposed as a counterpart to Design Indicators and enable one to measure the extent to which the objectives are realized in a developed product. While Design Indicators focus on design structure characteristics of the product, Management Indicators focus on the acquisitional, behavioral, and maintenance characteristics. To accomplish the validation of the predictive capability, the correlation between the values obtained by utilizing Design Indicators and those obtained by utilizing Management Indicators must be investigated. The author has chosen to study and present the software engineering objectives of reliability and maintainability as they related to a future validation effort.

[Rowl81a] Abstract: The element z is called a transcendental for the class F if functions in F can be uniquely

identified by their values at z. Conditions for the existence of transcendentals are discussed for certain classes of polynomials, and rational functions. Of particular interest are those transcendentals having an exact representation in computer arithmetic. Algorithms are presented for reconstruction of the coefficients of a polynomial from its value at a transcendental. The theory is illustrated by application to polynomials, quadratic forms, and quadrature formulas.

[Rowl88] Abstract: This paper describes techniques for the automatic generation of large artificial software systems which can be used for laboratory studies of testing and integration strategies, reliability models, and so forth. A prototype generator is described which produces code for such systems by constructing a large number of nearly identical modules. This generator has been used to construct a family of systems which in theory can be made arbitrarily large. Several experiments were conducted to explore the sensitivity of the Jelinski-Moranda model to violations of the assumption that all defects have equal probability of being discovered.

[Rube75] Abstract: This paper discusses the need for quantitative descriptions of software errors and methods for gathering such data. The software development cycle is reviewed, and the frequency of the errors that are detected during software development and independent validation are compared. Data obtained from validation efforts are presented, indicating the number of errors in ten categories and three severity levels; the inferences that can be drawn from these data are discussed. Data describing the effectiveness of validation tools and techniques as a function of time are presented and discussed. The software validation cost is contrasted with the software development cost. The applications of better quantitative software error data are summarized.

[Rums77] Abstract: The performance measure and analysis of software operating systems which extend basic computing machinery is discussed. The description of an external monitoring technique which facilities the correlation of hardware events with software functions without the need for software monitors is presented. A time related event is defined to provide the basis for the technique used to implement the monitor system. In addition, event analysis methods are introduced which allow a software system execution profile to be constructed.

[Rust71] Abbreviated Introduction: This volume deals with efforts at control and extermination of that notorious form of non-insect life which we in the programming community refer to, somewhat contemptuously, as "bugs." Although as individuals we may in less cautious moments speak of bugs with cavalier disdain, it is always with a latent awareness that such bravado may be the harbinger of a period of intense bug-hunting, relieved only by occasional naps on piles of discarded dumps. To the bug-plagued victim, the sympathetic nods of one's colleagues more often suggest relief that it is "him rather than me."

The more fatalistic among us may find such a period good for the soul; a penance for the general malfeasance of those involved in activity in which a quantity of intellectual self-indulgence is tolerated. Of course, even given the frustration of the exterminating effort, there is the pleasure in locating and ridding a program of the infecting source. The gratification of discovery could only be enhanced at finding the bug was someone else's.

[SDIO87] Overview: The Strategic Defense Initiative Organization (SDIO) Test and Evaluation Master Plan (TEMP) outlines the planning and management of test and evaluation activities for the Strategic Defense System. It is an evolving document. Detailed planning, and results from, individual test and evaluation activities will be included as the software effort proceeds.

[SDIO88a] Overview: The Strategic Defense Initiative Organization (SDIO) Software Policy requires the use of promising software engineering approaches for the development and evolution of all full scale development Strategic Defense System (SDS) software. To ensure that all SDS mission-critical software exhibits the necessary levels of quality, software efforts are required to address requirements of software reliability, security, interoperability, portability, maintainability, and usability throughout the system life cycle.

The policy is restricted to identifying requirements for software engineering practices. The services, and other implementing agents, will develop their own implementation documents that are consistent with their

existing or planned software engineering management practices.

[SDIO88b] Overview: This Strategic Defense Initiative Organization (SDIO) Management Directive specifies the implementation of the SDIO Software Policy [SDIO88a] required on all software efforts sponsored directly by the SDIO. In addition to specifying how the Software Policy must be reflected in requests for proposals, and other contracting documents, the management directive explicitly enumerates those conditions under which requests for waivers to the Software Policy will be accepted.

[SERC87] Abstract: Mutation analysis is a software testing technique that measures test data adequacy that is, the ability of test data to ensure that certain errors are not present in the program under test. Mothra is a software testing environment built on the mutation analysis approach to determining test effectiveness. It consists of an integrated set of tools that allow the user to interactively test Fortran-77 software throughout the software development cycle. Mothra currently runs under 4.3 BSD UNIX, System V UNIX, and ULTRIX 32 1.2.

This document is a user's manual for first time users of Mothra as well as a reference manual for more experienced users. The manual describes the function both of the tools that comprise Mothra and of *cdemo*, a simple interface that was designed to facilitate the use of these tools. The first section provides some background information and an explanation of the steps involved in using Mothra to test software. Readers wishing more detailed information on mutation analysis should consult the bibliography. The second section describes the specifics of *cdemo* itself. Examples of software testing with Mothra are presented throughout the document.

[Sack68] Abstract: Two exploratory experiments were conducted at System Development Corporation to compare debugging performance of programmers working under conditions of online and offline access to a computer. These are the first known studies that measure programmers' performance under controlled conditions for standard tasks.

Statistically significant results of both experiments indicated faster debugging under online conditions, but perhaps the most important practical finding involves the striking individual differences in programmer performance. Methodological problems encountered in designing and conducting these experiments are described; limitations of the findings are pointed out; hypotheses are presented to account for results; and suggestions are made for further research.

[Sahn87] Abstract: A graph-based modeling technique has been developed for the stochastic analysis of systems containing concurrency. The basis of the technique is the use of directed acyclic graphs. These graphs represent event-precedence networks where activities may occur serially, probabilistically, or concurrently. When a set of activities occur concurrently, the condition for the set of activities to complete is that a specified number of the activities must complete. This includes the special cases that one or all of the activities must complete. The cumulative distribution function associated with an activity is assumed to have exponential polynomial form. Further generality is obtained by allowing these distributions to have a mass at the origin and/or at infinity. The distribution function for the time taken to complete the entire graph is computed symbolically in the time parameter t. The technique allows two or more graphs to be combined hierarchically. Applications of the technique to the evaluation of concurrent program execution time and to the reliability analysis of fault-tolerant systems are discussed.

[Salt82] Abstract: Although the Software Science metrics originally proposed by Halstead are appealing, calculation of the metrics depends on the existence of well-defined counting strategies. The strategies require precise definitions of operators and operands. It is important that the strategies employed be described in research papers. Furthermore, the presentation of helpful examples of the application of the strategies is recommended. Good descriptions do not imply correct strategies, but they do ensure that the strategies can be understood, tested, and evaluated. Appendices to this paper provide the description of a Pascal counting strategy and an example of applying the strategy.

[Same76] Abstract: A method for compiler testing using symbolic interpretation is presented. This method is a cross between program proving and program testing. It is useful in demonstrating that programs are correctly translated from a high level language to a low level language thereby improving the reliability of the compiler. The term symbolic interpretation is used to describe the process of obtaining an intermediate form of the low level language program that is suitable for further processing by a proof system. Symbolic interpretation is the heart of the system and enables the recording of a transcript of all computations in the program. This process interprets a set of procedures which describe the effects of machine language instructions corresponding to the target machine on a suitable computation model. The highlights and limitations of the process as well as future work are discussed in a framework of a specific LISP implementation on a PDP-10 computer.

[Sank85] Abstract: Anna is a language extension of Ada to include facilities for formally specifying the intended behavior of Ada programs. It augments Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs.

This paper describes an implementation of a subset of Anna. The implementation is a transformer that accepts as input an Anna parse tree and produces as output an equivalent Ada parse tree that contains the necessary executable runtime checks for the Anna specifications. An approach called the *Checking Function Approach* is used. This involves the generation of a function for each annotation and generating calls to these functions at appropriate places. The transformer has to take care of various details like hiding, overloading, nesting, etc.

It is hoped that the transformer will eventually cover most of Anna and have various features like a good user interface, interaction with a symbolic debugger, and optimization of runtime checks for permanent inclusion.

[Sari84b] Abstract: Protocol testing for the purpose of certifying the implementation's adherence to the protocol specification can be done with a test architecture consisting of remote tester and local responder processes generating specific input stimuli, called test sequences, and observing the output produced by the implementation under test. It is possible to adapt test sequence generation techniques for finite state machines, such as transition tour, characterization, and checking sequence methods, to generate test sequences for protocols specified as incomplete finite state machines. For certain test sequences, the tester or responder processes are forced to consider the timing of an interaction in which they have not taken part; these test sequences are called nonsynchronizable. The three test sequence generation algorithms are modified to obtain synchronizable test sequences. The checking of a given protocol for intrinsic synchronization problems is also discussed. Complexities of synchronizable test sequence generation algorithm are given and complete testing of a protocol is shown to be infeasible.

To extend the applicability of the characterization and checking sequences, different methods are proposed to enhance the protocol specifications: special test input interactions are defined and a methodology is developed to complete the protocol specifications.

[Sari87] Abstract: Communication protocol testing can be done with a test architecture consisting of remote Lower Tester and local Upper Tester processes. For real protocols, tests can be designed based on the formal specification of the protocol which uses an extended finite state machine model. The specification is transformed into a simpler form consisting of normal form transitions. It can then be modeled by a control and a data flow graph. The graphs are decomposed into subtours and data flow functions, respectively. Tests are designed by considering parameter variations of the input primitives of each data flow function and determining the expected outputs. The methodology gives complete test coverage of all data flow functions and control paths in the specification. Functional fault models are proposed for functions that are not formally specified.

[Sari88a] Abstract: With wide-spread acceptance of the ISO-OSI reference model and its standardized protocols in the areas of computer communication and information exchange, various types of protocol testing [have] become an area of active research and development. This paper surveys recent developments in protocol validation. The discussion includes two important components any protocol test system must have: test sequence

generator and trace checker as well as protocol verification techniques.

[Sark89] Abstract: At the heart of any program verifier lies a theorem prover which proves theorems over the domain of the program. For any meaningful program, the theorems encountered are quite complex. The problem, which is equivalent to the validity problem of second-order logic, reduces to that of first-order logic when the assertions of the program are available. In both the cases, the problem remains undecidable and human intervention at some stage or other becomes essential. Resolution-based theorem provers proposed for first-order logic are very popular because they allow easy human intervention. However, the theorems encountered in proving programs do not follow the exact syntax of predicate calculus; rather, they are obtained in more popular algebraic notation. Thus, the inference rules available in first-order logic are not directly applicable to the verification conditions of the paths of the program.

In the present paper significant modifications of the first-order rules have been developed so that they apply directly to the algebraic expressions. The importance and implication of normalization of formulas in any theorem prover have been discussed. It has also been shown how the properties of the domain of discourse have been taken care of either by the normalizer or by the inference rules proposed. Through a nontrivial example the following capabilities of the verifier, which would use these inference rules, have been highlighted: 1) closeness of the proof construction process to human thought process and 2) efficient handling of user provided axioms; such capabilities make the interfacing with human element easy.

[Satt72] Summary: The design of an integrated programming and debugging system using the language ALGOL W is described. The debugging tools are based entirely upon the source language but can be efficiently implemented. The most novel such tool is a selective trace, automatically controlled by execution frequency counts. System performance information is included.

[Scha79] Abstract: This report presents the results of a study and investigation of software reliability models. In particular, the purpose was to investigate the statistical properties of selected software reliability models, including the statistical properties of the parameter estimates, and to investigate the goodness fit of the models to actual software error data. The results indicate that the models fit poorly, generally due to in most part the vagaries of the data rather than shortcomings of the models.

[Schi78] Abstract: This paper examines the most widely used reliability models. The models discussed fall into two categories, the data domain and the time domain. Besides tracing the historical development of the various models their advantages and disadvantages are analyzed. This includes models based on discrete as well as continuous probability distributions. How well a given model performs its purpose in a specific economic environment will determine the usefulness of the model. Each of the models is examined with actual data as to the applicability of the error finding process.

[Schn75] Abstract: A non-homogeneous poisson process is used to model the occurrence of errors detected during functional testing of command and control software. The parameters of the detection process are estimated by using a combination of maximum likelihood and weighted least squares methods. Once parameter estimates are obtained, forecasts can be made of cumulative number of detected errors. Forecasting equations of cumulative corrected errors, errors detected but not corrected, and the time required to detect or correct a specified number of errors, are derived from the detected error function. The various forecasts provide decision aids for managing software testing activities. Naval tactical data system software error data are used to evaluate several variations of the forecasting methodology and to test the accuracy of the forecasting equations.

[Schn77b] Abbreviated Introduction: The significance of program structural characteristics has been recognized for some time, as witnessed by the emergence of structured programming. But there is another tool available that has usually been overlooked in the software development process: simulation.

Simulation is relatively new to the evaluation and measurement of software— even though examples abound of simulation and analytical models that have been developed for modeling software error detection.

This paper attempts to show how simulation can be used both to evaluate alternatives during design and to simulate the detection of errors during testing.

To improve program quality we must not only avoid errors during program design; we must also detect them during testing. Hence, one of the characteristics of a good design is a program structure that allows easy error detection.

A convenient way of describing program structure and simulating the detection of errors is to represent the program in a directed graph. By using a directed graph to represent the structure of a program and simulation to study program error detection, the following information can be obtained:

1. Error detection (number or fraction of errors detected) as a function of a program's structural characteristics, for a given number of tests. The test consists of beginning simulated program execution at the start node, detecting and correcting any errors, restarting at the start node, and repeating this process until a terminal node is reached.
2. Error detection as a function of number of tests for given structural characteristics.

Structural characteristics correspond to program characteristics. For example, numbers of nodes, arcs, paths and source statements correspond to branching and merging, arithmetic and data transfer operations, execution sequences, and size.

[Schn77c] Abstract: Program structure and modularity are important considerations for the development of reliable software. Most software specialists agree that higher reliability is achieved when software systems are highly modularized and module structure is kept simple. However if this principle is carried too far in the design of large systems, lower rather than higher reliability may result. This may occur because the added complexity of a large number of communication paths among a large number of small modules may exceed the reduction in complexity of individual modules. Real time operating system structures are examined in terms of their modularity characteristics. Proposals are advanced for improving the structure of real time operating systems.

[Schn79a] Abstract: The propensity to make programming errors and the rates of error detection and correction are dependent on program complexity. Knowledge of these relationships can be used to avoid error prone structures in software design and to devise a testing strategy which is based on anticipated difficulty of error detection and correction. An experiment in software error data collection and analysis was conducted in order to study these relationships under conditions where the error data could be carefully defined and collected. Several complexity measures which can be defined in terms of the directed graph representation of a program, such as cyclomatic number, were analyzed with respect to the following error characteristics: errors found, time between error detections, and error correction time. Significant relationships were found between complexity measures and error characteristics. The meaning of directed graph structural properties in terms of the complexity of the programming and testing tasks was examined.

[Schn79b] Introduction: Computer program graphs have proven very useful because they eliminate the structural characteristics of a program. Structural characteristics, as a representation of program complexity, have been shown to be strongly related to program development time, program quality and difficulty of debugging. The use of graphs for these purposes is not widely known or understood in the data processing community. It is the aim of this paper to provide an introduction to graphs as they apply to program representation and to show examples of their use in program design and debugging.

[Schr84] Abstract: This paper describes an attempt to integrate the collection and the efficient utilization of measurements in the development and the use of programs. The work presented consists in three parts:

- the design of both static and dynamic measurement tools,
- examples of data processing on measurements collected on a sample of Pascal programs,
- the design of a quantitative documentation of a program, which is automatically built as measurements are collected.

The first and third steps have been developed inside an existing programming environment, *Mentor*, and we shall discuss the advantages we found in integrating the tools in such an environment.

[Schu81] Abstract: This paper addresses the problem of programming distributed systems within the framework of the Ada language, which provides primitives for interprocess communication based upon the model of Communicating Sequential processes. We first discuss our basic assumptions concerning the underlying target configuration, the physical communication medium which is to support that application and pattern of the logical communication within the application proper. We then develop a first approach for constructing such applications using the separate compilation facilities of Ada. Finally, we consider two possible protocols for implementing the requisite distributed interprocess communication, referred to as the Remote Entry Call and the Remote Procedure Call, respectively.

[Schw70a] Overview: A survey of the type, frequency, and habitat of bugs is outlined. Debugging tools presently available are discussed and suggestions for their development advanced. The role of "proofs of program correctness" and the debugging process itself are discussed.

[Scot84a] Abstract: New data domain reliability models have been developed for the N-version, Recovery Block and Consensus Recovery Block approaches to fault-tolerant software and investigation of the validity of each of these models is underway. Central to validation is the underlying dependence of the multiple versions of software modules required by these approaches and the impact of this dependence on reliability predictions. This paper presents reliability models for all three fault-tolerance approaches using assumptions of both independence and dependence. The presentation of the experimental investigation focuses in the Recovery Block strategy. The results can be summarized by saying the models relying on the assumption of module independence did not adequately predict reliability on the experiments. The dependent models were successful. Furthermore, the underlying dependence could not be attributed to common cause errors resulting from similarities in the solution algorithms. Rather, the dependence was attributable to the difficulty of the input test cases.

[Scot84b] Abstract: Results are presented for an experiment conducted at North Carolina State University to validate the author's fault-tolerant software reliability models. Both independent and dependent versions of the Recovery Block, N-Version Programming, and Consensus Recovery Block reliability models were studied. It was shown that the assumption of version independence leads to poor predictions of reliability. The reliability gains offered by each of the three methods of software fault-tolerance were also compared.

[Scot87] Abstract: In situations in which computers are used to manage life-critical situations, software errors that could arise due to inadequate or incomplete testing cannot be tolerated. This paper examines three methods of creating fault-tolerant software systems, Recovery Block, N-Version Programming, and Consensus Recovery Block, and it presents reliability models for each. The models are used to show that one method, the Consensus Recovery Block, is more reliable than the other two.

The results of an experiment used to validate the models are presented. It is demonstrated that, for highly reliable acceptance tests, the Consensus Recovery Block system gave the highest reliability. In all cases, the Consensus Recovery Block and Recovery Block systems were better than the N-Version Programming systems.

A simple cost model that shows the relative costs of increasing software reliability using the three fault-tolerant methods is presented.

[Sedl83] Abstract: Fault localization in program debugging is the process of identifying program statements which cause anomalous behavior. We have developed a prototype, knowledge-based model of the fault localization process. Novel features of the model include multiple localization tactics and a recognition-based mechanism for program abstraction. An explicit division of knowledge from the applications, programming and language domains facilitate model tuning within as well as across applications domains. We describe model structure and performance for a class of faults associated with master file update programs. We foresee applications of the model as an initial cognitive theory of expertise in fault localization and as a partially automated debugging tool.

[Selb85] Abbreviated Abstract: The evaluation of software technologies suffers because of the lack of

quantitative assessment of their effect on software development and modification. A seven-step approach for quantitatively evaluating software technologies couples software methodology evaluation with software measurement. The approach is applied in-depth in the following three areas. 1) Software Testing Strategies: A 74-subject study, including 32 professional programmers and 42 advanced university students, compared code reading, functional testing, and structural testing in a fractional factorial design. 2) CLEANROOM Software Development: Fifteen three-person teams separately built a 1200-line message system to compare CLEANROOM software development (in which software is developed completely off-line) with a more traditional approach. 3) Characteristic Software Metric Sets: In the NASA SEL production environment, a study of 65 candidate product and process measures of 652 modules from six (51,000 - 112,000 line) projects yielded a characteristic set of software cost/quality metrics.

[Selb86] Abstract: This study compares the three testing strategies of (1) code reading by stepwise abstraction, (2) functional testing using equivalence partitioning and boundary value analysis, and (3) structural testing with 100% statement coverage criteria - and the six pairwise combinations of these techniques. Thirty two professional programmers applied the techniques to three unit-sized programs in a fractional factorial experimental design.

The major results of this study are the following.

1. The six combined testing approaches detected 17.7% more of the program' faults on the average than did the three single techniques, which was a 35.5% improvement in fault detection.
2. The highest percentage of the programs' faults were detected when there was a combination of either two code readers or a code reader and a functional tester. However, a pairing of two code readers detected more faults per hour than did a pairing of a code reader and a functional tester.
3. The pairing of two individuals of advanced expertise resulted in the highest percentage of faults being detected.
4. The most cost-effective (number of faults detected per hour) testing approach overall was when code reading was applied by an individual. The most cost-effective combined testing approach was when a code reader was paired with either another code reader or a structural tester.
5. Both the percentage of faults detected and the fault detection cost-effectiveness depended on the type of software being tested.

[Selb87a] Abstract: Software metrics have been useful to measure, evaluate, and control the software development process and evolving software product. Software environments provide software tools and infrastructure to support a variety of activities related to software development. This paper proposes 23 guidelines for incorporating metrics into software environments. The guidelines are organized into five areas: the purpose, type, scope, collection, and analysis of metrics. An example application of the guidelines in a software environment project is described briefly.

[Selb87b] Abstract: The CLEANROOM software development approach is intended to produce highly reliable software by integrating formal methods for specification and design, nonexecution-based program development, and statistically based independent testing. In an empirical study, 15 three-person teams developed versions of the same software system (800-2300 source lines); ten teams applied CLEANROOM, while five applied a more traditional approach. This analysis characterizes the effect of CLEANROOM on the delivered product, the software development process, and the developers.

The major results of this study are the following.

1. Most of the developers were able to apply the techniques of CLEANROOM effectively (six of the ten CLEANROOM teams delivered at least 91% of the required system functions).
2. The CLEANROOM teams products met system requirements more completely and had a higher percentage of successful operationally generated test cases.
3. The source code developed using CLEANROOM had more comments and less dense control-flow complexity.

4. The more successful CLEANROOM developers modified their use of the implementation language; they used more procedure calls and IF statements, used fewer CASE statements and WHILE statements, and had a lower frequency of variable reuse (average number of occurrences per variable).
5. All ten CLEANROOM teams made all of their scheduled intermediate product deliveries, while only two of the five non-CLEANROOM teams did.
6. Although 86% of the CLEANROOM developers indicated that they missed the satisfaction of program execution to some extent, this had no relation to the product quality measures of implementation completeness and successful operational tests.
7. 81% of the CLEANROOM developers said that they would use the approach again.

[Selb88a] Abstract: One central feature of the structure of a software system is the coupling among its components (e.g., subsystems, modules) and the cohesion within them. The purpose of this study is to quantify ratios of coupling and cohesion and use them in the generation of hierarchical system descriptions. The ability of the hierarchical descriptions to localize errors by identifying error-prone system structure is evaluated using actual error data. Measures of data interaction, called data bindings, are used as the basis for calculating software coupling and cohesion. A 135,000 source line system from a production environment has been selected for empirical analysis. Software error data was collected from high-level system design through system test and from some field operation of the system. A set of five tools is applied to calculate the data bindings automatically, and cluster analysis is used to determine a hierarchical description of each of the system's 77 subsystems. An analysis of variance model is used to characterize subsystems and individual routines that had either many/few errors or high/low error correction effort.

[Shan80] Abstract: The main intent of this paper is to derive expressions for software performance prediction using a state-dependent error occurrence-rate model. Using a Markov process representation for the remaining number of errors in the software system we derive a set of linear difference-differential equations for the probability distribution of the number of remaining errors at an arbitrary time t. Solving this set of equations we obtain a binomial distribution for the number of remaining errors. We also obtain the relevant system performance measures for the software system. This analysis is first carried out assuming that the initial error content at the time t=0 is a fixed unknown constant and subsequently extend it for the case in which the initial error content is a random variable. Using these results we exhibit an interesting insensitivity characteristic of this model.

[Shan81] Abstract: In this paper, assuming a state- and time-dependent software failure rate and imperfect debuggings, we develop a simple binomial model for software error occurrences. Maximum likelihood estimates for the required parameters of this model are also derived. It is established that the Jelinski-Moranda, imperfect debugging and non-homogeneous Poisson process models are all special cases of ours.

[Shan82] Abstract: The purpose of this paper is to develop a method for designing and verifying data abstractions using the functional approach. Before doing so, the existing techniques for designing and verifying procedure and data abstractions will be surveyed briefly. These techniques will then be modified and extended to verify data abstractions. By using the concept of a mathematical function, one can model the behavior of a procedure abstraction and give a more uniform and clearer meaning to the stepwise refinement and verification of procedure abstractions. The concept of a state machine is then used as a basis to specify data abstractions. Using state machine specification, a technique for expressing the design of a data abstraction is then given. A method is then developed to verify the design of a data abstraction with respect to its specifications.

[Shat88] Abstract: In order to understand and analyze real-time distributed programs, one must account for interactions between processes. Unfortunately, these interactions can be quite complex due to concurrency and nondeterminism. This paper describes a framework for automated static analysis of distributed programs written in Ada. The analysis is aimed at discovery of a program's potential tasking behavior, that is, behavior in terms of tasking-related issues. Central to the framework is the translation of a program into an abstract grammar system that represents a Petri net graph model.

[Shaw78] Abstract: Flow expressions describe sequential and concurrent flows of entities, such as control, messages, commands, jobs, and resources, through system software components, such as programs, procedures, modules, and processes. They consist of regular expressions extended with cyclic and interleaving operators and a synchronization facility. The language of flow expressions is defined and some of its formal properties are presented. Applications are exhibited in the modeling of concurrent programs, the description of operating system architectures, the specification and solution of synchronization problems, the flow and description of command languages, and in systems analysis and verification.

[Shaw89] Abstract: Halstead's theory of software science is used to describe the compilation process and generate a compiler performance index. A nonlinear model of compile time is estimated for four Ada compilers. A fundamental relation between compile time and program modularity is proposed. Issues considered include data collection procedures, the development of a counting strategy, the analysis of the complexity measures used, and the investigation of significant relationships between program characteristics and compile time. The results suggest that the model has a high predictive power and provides interesting insights into compiler performance phenomena. The research suggests that the discrimination rate of a compiler is a valuable performance index and is preferred to average compile time statistics.

[Shei81] Abstract: Most innovations in programming languages and methodology are motivated by a belief that they will improve the performance of the programmers who use them. Although such claims are usually advanced informally, there is a growing body of research which attempts to verify them by controlled observation of programmers' behavior. Surprisingly, these studies have found few clear effects of changes in either programming notation or practice. Less surprisingly, the computing community has paid relatively little attention to these results. This paper reviews the psychological research on programming and argues that its ineffectiveness is the result of both unsophisticated experimental technique and a shallow view of the nature of programming skill.

[Shen83] Abstract: The theory of software science was developed by the late M.H. Halstead of Purdue University during the early 1970's. It was first presented in unified form in the monograph "Elements of Software Science" published by Elsevier North-Holland in 1977. Since it claimed to apply scientific method to the very complex and important problem of software production, and since experimental evidence supplied by Halstead and others seemed to support the theory, it drew widespread attention from the computer science community.

Some researchers have raised serious questions about the underlying theory of software science. At the same time, experimental evidence supporting some of the metrics continues to be presented. This paper is a critique of the theory as presented by Halstead and a review of experimental results concerning software science metrics published since 1977.

[Shen85] Abstract: A major portion of the effort expended in developing commercial software today is associated with program testing. Schedule and/or resource constraints frequently require that testing be conducted so as to uncover the greatest number of errors possible in the time allowed. In this paper we describe a study undertaken to assess the potential usefulness of various product- and process-related measures in identifying error-prone software. Our goal was to establish an empirical basis for the efficient utilization of limited testing resources using objective, measurable criteria. Through a detailed analysis of three software products and their error discovery histories, we have found simple metrics related to the amount of data and the structural complexity of programs to be of value for this purpose.

[Shep78] Overview: The late 70's find structured programming increasingly popular—this and other techniques are programming's future. But what does experimental evaluation say about their actual effects on programmer performance?

[Shep79] Abbreviated Introduction: In a series of experiments we investigated the effects of modern coding practices on three different programming tasks. The first experiment examined the effects of structured coding and mnemonic variable names on *program comprehension*. The second studied the influence of structured

coding and commenting style on *modification tasks*. The third studied the influence of structured coding and of several code-structuring methods on *debugging performance*. Participants in these experiments were all professional programmers whose experience ranged from several months to 25 years and averaged six or more years. Participants in each experiment were selected from several locations in order to increase the diversity of programming backgrounds.

[Shim88] Abbreviated Introduction: Reliability is a pressing concern in the development of software for modern systems. Many techniques have been proposed to improve software reliability. One technique, N-Version Programming, has been used in software to control aircraft and railroads and has been proposed for nuclear power plants. One drawback to the n-version technique is that the total development costs are increased due to the costs of developing multiple versions.

In order to make the technique affordable, it has been suggested that n-version programming will be so effective that it can be used as a partial substitute for current software verification and validation procedures. It seems important to investigate the hypothesis that testing can be reduced in n-version systems, and in general, to study the relationship between fault elimination techniques and fault tolerance techniques.

There have also been proposals to use n-version voting in the testing process. In this method, the vote itself is used as the test oracle, and, therefore, a larger number of tests can be executed. The underlying assumptions here are that (1) given that a fault leads to an erroneous output, it will be detected by the voting process, and (2) the faults that would have been detected by other testing techniques, such as structural testing or static analysis techniques, will be elicited and detected by voting on random or functional test cases alone.

The authors of this paper are engaged in a large-scale experiment comparing software fault-tolerance and software fault elimination as approaches to improving software quality. This paper describes the experiment and the results that apply to the appropriateness and underlying assumptions of these two proposals.

[Shne75] Abbreviated Background: In the early stages of the development of high-level languages, radically differing alternatives were often promulgated. Now as the field matures, there is a widespread recognition of the usefulness of a variety of languages.

Although Dijkstra explicitly stated that computer programming was primarily a *human* activity as early as 1965, it was not until the publication, in 1971, of Gerald Weinberg's text *The Psychology of Computer Programming* that this notion was widely recognized. [This] text concentrates on defining the programming task in the context of the professional environment and promotes the notion of "egoless programming teams." This team organization concept may be contrasted with the "chief programmer team" strategy advocated by IBM. Experimental comparison of interactions in these personal organization strategies would be an intriguing task for social psychologists. Other sections of Weinberg's book concentrate on individual personality factors, training, and motivational factors. Much more research needs to be done on the psychological make-up of programmers. Fortunately, psychologists have begun to study programming behavior as an aspect of problem solving. Training and teaching of programming has long been of interest to academically oriented researchers. Programming has only recently become a subject for related disciplines such as educational psychology.

Although experimentation in the above mentioned areas would undoubtedly be welcome, the focus of this paper is on experiments in programming language features, stylistic considerations and design techniques.

[Shne77a] Abstract: This paper describes previous research on flowcharts and a series of controlled experiments to test the utility of detailed flowcharts as an aid to program composition, comprehension, debugging, and modification. No statistically significant difference between flowchart and nonflowchart groups has been shown, thereby calling into question the utility of detailed flowcharting. A program of further research is suggested.

[Shol75] Abstract: An engineering-oriented performance model of a computation is developed by extending the concept of a computation structure to cover the performance costs appropriate to software modeling. The model allows both serial and parallel (multiprocessor) configurations, and the evaluation of both time and space parameters for alternate realizations.

A brief discussion on the use of the model as a mechanism to guide the performance optimization of

programs is included.

[Shoo72] Abstract: This paper discusses a probabilistic model for predicting software reliability. The model constants are calculated from debugging data collected from similar previous programs. The calculations result in a decreasing probability of number of software errors vs. operating time (reliability function). The decay rate of the reliability function (reciprocal of the mean time to failure) decreases as a function of the man-months of debugging time. The model provides initial estimates of software reliability before any code is written and allows later updating to improve the accuracy of the parameters when integration or operational test begin.

[Shoo75] Abstract: In order to develop some basic information on software errors, an experiment in collecting data on types and frequencies of such errors was conducted at Bell Laboratories.

The paper reports the results of this experiment, whose objectives were to: (1) Develop and utilize a set of terms for describing possible types of errors, their nature, and their frequency; (2) Perform a pilot study to determine if data of the type reported in this paper could be collected; (3) Investigate the error density and its correspondence to predictions from previous data reported; (4) Develop data on how resources are expended in debugging.

A program of approximately 4K machine instructions (final size) was chosen. Programmers were asked to fill out for each error, in addition to the regular Trouble Report/Correction Report (TR/CR) form, a special Supplementary TR/CR form for the purpose of this experiment. Sixty-three TR/CR and Supplementary forms were completed during the Test and Integration phase of the program.

In general, the data collected were felt to be accurate enough for the purpose of the analyses presented. The 63 forms represented a little over 1-1/2% of the total number of machine instructions of the program. (In good agreement with the 1% to 2% range noted on previous studies.)

It was discovered that a large percentage of the errors was found by hand processing (without the aid of a computer). This method was found to be much cheaper than techniques involving machine testing.

[Shoo76] Abstract: Many previous software reliability prediction models by this author and others have concentrated on the bulk (macro) aspects of the program. This paper describes a newly developed micro model which is based on program structure.

It is assumed that the program has been written in structured or modular form so that decomposition into its constituent parts is simple. Further, we assume that via analysis of the program the decomposition can be related to several paths or other functional structures within the program.

The model is constructed based upon the frequencies with which each of the j paths are run, (f_j), the running time of each path, (t_j), and the probability of error along each path, (q_j).

Several methods of calculating or measuring the f_j, t_j, and q_j parameters are suggested. In fact it is possible to use one technique (historical data) to produce crude estimates at the start of the design, and refine the estimates with more accurate values as the design progresses.

The paper concludes with the application of the model to a particular example: calculation of the roots of a quadratic equation, and a discussion of proposed experiments for validating the model.

[Shoo77a] Abstract: The paper begins by describing the types and causes of software errors and provides working definitions of software errors and software reliability. Some of the basic data on frequency of occurrence of errors is then discussed. The paper then summarizes and references some of the software reliability models which have been proposed and concentrates on one developed by the author. One of the probabilistic models, the macro model, predicts reliability based on the initial number of errors in a program, the number removed, and the number remaining in the program. The model constants are calculated from operational test data taken on the software performance. The other, the micro model, focuses on the paths in the program, their frequency and time of traversal, and the error rate along these paths.

[Shoo79] Abbreviated Abstract: This interim report summarizes the research performed by Polytechnic Institute of New York for Rome Air Development Center under contract F30602-78-C-0057. The principal topics

covered are (1) software test models and implementation of automated test drivers to force-execute every program path, (2) development of new measures of program complexity based upon information theory, (3) models of software management and organizational structure, and (4) statistical measures relating the probability of finding a program error to the testing of that program.

Recursive function theory was applied to the problem of program complexity. This study was completed and a technical report was issued. The present report contains the abstract of the technical reports.

The inquiry into the number of tests necessary to verify a computer program was undertaken. One phase of this study was completed, and a technical report was issued. The present report contains the abstract of the technical report.

A study was undertaken of software test models and of the implementation of associated test drivers. The present report describes this work as well as the test drivers obtained so far.

A new measure of complexity based upon information theory is introduced. This measure assumes that a language feature used infrequently is more likely to be used incorrectly than a language feature used frequently. The measure has the advantage of being sensitive to the different levels of nestings in either IF'S, DO'S, or procedures.

A number of different schemes are suggested for the calculation of the measure. A method for automatic calculation of the measure at an installation is also discussed.

The relation between program complexity and the program's information content was also investigated. The results obtained so far are described in this report.

Two models for the management of software were investigated. The first one models the productivity (measured in instructions per months), as well as the man-months required. The second model investigates different communication schemes that can be evolved when a problem is partitioned into several subproblems.

The concluding section section of the report describes the planned work in the next period and lists professional activities of the personnel during the present reporting period.

[Sidh89] Abstract: A protocol standard, in general, can lead to different implementations, which necessitate the need for conformance testing of an implementation to its standard. Testing is carried out with the help of a test sequence generated from a protocol specification. This paper presents a detailed study of four formal methods (T-, U-, D-, and W-methods) for generating test sequences for protocols. Applications of these methods to NBS Class 4 Transport Protocol are discussed. This paper also presents an estimation of fault coverage of four protocol test sequences generation techniques using Monte Carlo simulation. The ability of a test sequence to decide whether a protocol implementation conforms to its specification heavily relies upon the range of faults that it can capture. Conformance is defined at two levels, namely, weak and strong conformance. This study shows that a test sequence produced by T-method has a poor fault detection capability whereas test sequences produced by U-, D- and W-methods have comparable (superior to that for T-method) fault coverage on several classes of randomly generated machines used in this study. Also, some problems with a straightforward application of the four protocol test sequence generation methods to real-world communication problems are pointed out.

[Sief88] Abstract: The purpose of this project was to develop a tool to automate the method for evaluating software quality in Software Quality Evaluation Guidebook RADC-TR-85-37 Vol III (of three). The Automated Measurement System (AMS), a computer-based software tool, provides the capabilities to monitor the overall quality and resource expenditure of software under development. The AMS collects, stores and analyzes software measurement data for use by software acquisition and software project personnel. It provides managers with a means to quantitatively specify goals and track progress toward those goals during all phases of the software life cycle (in concert with DOD-STD-2167). The underlying philosophy of the AMS is based on a framework consisting of a set of 13 software factors (i.e., reliability, maintainability, reusability, portability, interoperability, usability, integrity, flexibility, expandability, verifiability, correctness, survivability, and efficiency) which are associated with high level concerns of software quality.

[Skil89] Abstract: We present a methodology for transforming a functional specification written in Lucid, to an equivalent specification that captures its real-time properties. The enhanced specification consists of a set of

equations. These equations can be solved for several properties, including execution time and external requirements, or they may simply be checked for the existence of a solution. Lucid has a set of meaning-preserving transformations, and a proof system corresponding to a behavioral semantics has been constructed. Both of these tools can be used to reason about properties of the specification.

[Snee84] Abstract: The data processing community needs to apply software engineering techniques and tools to real projects to determine their practical usefulness. Such an opportunity was provided by the Bertelsmann Publishing Corporation of Gutersloh, West Germany, during a two year period from 1981 to 1983. This article reports the results of that project and the experience gained from it.

[Snee85] Abstract: This paper describes a family of tools which not only supports software development, but also assures the quality of each software product from the requirements definition to the integrated system. It is based upon an explicit definition of the design objectives and includes specification verification, design evaluation, static program analysis, dynamic program analysis, integration test auditing, and configuration management.

[Snee86] Abstract: The following paper presents a metric for measuring test coverage which will enhance the present test metrics. This measurement focuses on the data used by the program under test. By dynamically monitoring the change of data states at test time it is possible to record how data are actually used. By statically analyzing the operands of a program it is possible to record how they are referenced by the program. By analyzing the specification it is possible to derive how the data should be used. Finally, the specified use is compared with the programmed use and the programmed use with the tested use, in order to determine to what degree all specified data usages have been tested. The ratio of actual tested usage to the specified usage gives the total data coverage.

[Solo84] Abstract: We suggest that expert programmers have and use two types of programming knowledge: 1) *programming plans*, which are generic program fragments that represent stereotypic action sequences in programming, and 2) *rules of programming discourse*, which capture the conventions in programming and govern the composition of the plans into programs. We report here on two empirical studies that attempt to evaluate the above hypothesis. Results from these studies do in fact support our claim.

[Sone80] Abstract: A finite state, continuous time Markov model is presented, which provides reliability measures (i.e., expected down time) for duplicated and repairable fault-tolerant computing systems whose main penalty depends on the total duration of failures over a given time period. Most of the existing models estimate reliability measures (e.g., mean time before failure) derived from the reliability function, meaningful only for systems whose main penalty depends on the frequency of failures. In addition, the model described here removes the simplifying assumption, made by some of the previous models, that the system is made of independent subsystems and this each subsystem can be modeled separately. Recognizing the fact that certain faults may affect more than one subsystem, this model represents the entire system, assuming however a small number of duplicated subsystem. The model has been implemented as a general interactive program to provide speedy estimation of reliability measures in the evaluation of fault-tolerant computer architecture designs. An example is included to illustrate the capability of the model.

[Sone81] Abstract:

[Soon77] Summary: This paper contributes to the understanding of program structures in terms of its stability and reliability in a quantitative sense. Distinctions are made between the logical structure of a program and the information structure of a program.

The general characteristics of a good program will not be discussed in this paper other than citing relevant references. The term stability is defined as the resistance to the amplification of changes that has been made to a given program. The information structure of a program is based on the sharing of information between the

components of the program.

Some quantitative analysis is derived to measure the quality of a program in terms of its information structure. The techniques used here are the method of connectivity matrix and that of random Markovian process. A high level quantitative measure of the information structure will be presented together with an informal proof for the uniqueness and the existence of this measure.

Applying this technique, several simple program information structures are measured for their stability characteristics. The simplicity of the structure is chosen deliberately so that this technique can be checked against intuitive preference of stability. Another example with a structure of some sophistication is presented to compare a tree structure to a pair of cooperating sequential processes.

[Sork79] Abstract: Here is a presently operational plan to improve the quality of program testing. After all programs are tested alone, an independent quality control staff uses automated tools to certify that minimum testing criteria have been met.

[Srin85] Introduction: The dataflow model of computation allows functions to be run concurrently on multiple processors, reducing execution time significantly. This advantage and the partial results of the computation will be lost if processors fail. Therefore, a crucial feature in a concurrent system is the ability to continue the computation when components fail, a feature known as fault tolerance.

Although several dataflow architectures have been proposed, few are fault tolerant and able to balance the load on the system dynamically. But a distributed computer system (DCS) based on a task-level dataflow architecture can reduce traffic, speed communication between processors, and tolerate hardware faults by automatically reassigning computations to a healthy processor. Such a DCS has the potential to provide better performance than conventional multiprocessors because the execution of a function is free of side effects.

By asking when and how to do node reassignment as the dataflow architecture and processor are designed, designers can incorporate the necessary support mechanisms. This article considers dataflow graphs with nodes representing asynchronous tasks.

[Stan83] Abstract: Arcturus offers an approach to the integrated use of compiled and interpreted Ada, template-driven Ada text editing, an Ada Program Design Language (PDL), Ada program performance measurement with color profiles, formated printing of Ada programs with useful listing options, and automated stepwise refinement from Ada program designs written in Ada PDL into executable Ada. This paper has two objectives: (a) to provide a two page thumbnail sketch of some interactive Ada capabilities in Arcturus, and (b) to provide a detailed scenario of interactive Ada programming at the very simplest level, using Ada-oriented variants of interactive programming techniques that have proven effective in practice — the hope being that the reader will be convinced that *interactive Ada* is an idea worth vigorous further pursuit.

[Stan84a] Abstract: The Arcturus system demonstrates several important principles that will characterize advanced Ada programming support environments. These include conceptual simplicity, tight coupling of tools, and effective command and editing concepts. Arcturus supports interactive program development and permits the combined used of interpretive and complied execution. Arcturus is not complete however, as practical, mature environments for Ada must also support the development, analysis, testing, and debugging of concurrent programs. These issues are currently being explored. Arcturus, therefore is a platform for experimental exploration of key programming environment issues. This paper focuses primarily on the current system, describing and illustrating some of its components, while issues less fully developed are more briefly described.

[Stet86] Abstract: In a recent paper, an approximate formula for the number of faults per line of code was developed. We show that there is an approximation which is easier to develop, more accurate, and simpler to use.

[Stev74] Abstract: Considerations and techniques are proposed that reduce the complexity of programs by dividing them into functional modules. This can make it possible to create complex systems from simple,

independent, reusable modules. Debugging and modifying programs, reconfiguring I/O devices, and managing large programming projects can all be greatly simplified. And, as the module library grows, increasingly sophisticated programs can be implemented using less and less new code.

[Stig74] Abbreviated Introduction: The complexity of digital computers and their large scale use have led some researchers to investigate tools not commonly used. In recent years, applications of graph theory to computers as well as other fields of study have given fruitful results and have attracted more and more scientists. The attempt here will be to review previous accomplishments on a fundamental level and to stimulate the reader to investigate an area where valuable work is being performed.

[Stuc72] Abbreviated Introduction: The measurement process plays a vital role in the quality assurance and testing of new hardware systems. To insure the reliability of the final hardware system, each stage of development incorporates performance standards and testing procedures. The establishment of software performance criteria has been very nebulous. At first the desire to "just get it working" prevailed in most software development efforts. With the increasing complexity of new and evolving software systems, improved measurement techniques are needed to facilitate disciplined program testing beyond merely debugging. The Program Testing Translator is an automatic tool designed to aid in the measurement and testing of software systems.

Early attempts at the application of measurement techniques to software dealt mainly with efforts to measure the hardware utilization characteristics. In an attempt to further improve hardware utilization, several aids have been developed ranging from optimized compilers to automated execution monitoring systems. The Program Testing Translator, designed to aid in the testing of programs, goes further. In addition to providing execution time statistics on the frequency of execution for various program statements, the Program Testing Translator performs a "standards" check to insure programmers' compliance to an established coding standard, gathers data on the extent to which various branches of a program are executed, and provides data range values on assignment statements and *DO-loop control variables*.

[Stuc75a] Abstract: Automated tools and structured programming techniques are in use on a variety of scientific and business application programming projects within the McDonnell Douglas Corp. An examination of the resulting programs reveals certain development and maintenance characteristics that suggest new and very interesting applications for automated tools.

An extension of PET (a currently operational McDonnell Douglas validation tool for FORTRAN) to include a user embedded assertion capability offers a step in the direction of automatically verifying the dynamic execution of programs. A user-oriented local and global assertion capability is introduced and its implementation is discussed.

Application of these tools within a well-conceived structured programming environment offers a positive step forward in the development of more reliable software systems.

[Stuc77] Abbreviated Background: Another set of tools has also been introduced over the last few years to deal with control flow through programs. Software probes or instrumentation are automatically placed into a program for monitoring the dynamic execution behavior of an algorithm. Software probes in the form of source language statements are inserted into the source code to gather statistics during program execution. These probes can provide insight into many aspects of algorithmic behavior beyond a simple flow of control analysis. The notion of building self-metric software has been introduced previously by the author; however, significant expansion of this concept is now being explored as a vehicle for improving software quality.

In order to illustrate the type of automated tool capabilities currently available and some of the new techniques now under construction, the tool most familiar to the author will be described. It is hoped that this currently operational system will offer some insight into the concept of self-metric software and show a few of the measurement schemes available for dynamic program analysis.

[Suke77a] Abstract: Reports on the initial phase of a software reliability modeling study, in which nine software reliability models are applied against software error data detailing the complete error history from the start of

formal testing through delivery of a large command and control software development project with over 100,000 lines of Jovial code. The paper describes the models considered and the procedures used to prepare the data for model input. Model predictions are then compared and analyzed against the actual post-delivery error data for this project. From this analysis, conclusions concerning model applicability and some possible extensions of this study are discussed.

[Suke79] Abstract: From 1974 Aug to 1978 May a study to validate several mathematical models for predicting the reliability and error content of a software package against error data extracted from four large U.S.A. Department of Defense software development projects was undertaken by Rome Air Development Center. This paper will describe the results of this empirical study for three such models: Jelinski-Moranda, Schick-Wolverton, modified Schick-Wolverton. Model predictions will be compared on a total project, functional, and error severity basis, and on a daily vs. weekly basis for defining model time intervals. The question of when to begin applying these models will be addressed. General conclusions are drawn as to model applicability.

[Suno82] Abstract: The program complexity measure currently seems to be the most capable measure for both quantitative and objective control of the software project. Five program complexity measures (step count, McCabe's V(G), Halstead's E, Weighted Statement Count and Process V(G)) were assessed from such a viewpoint. This empirical study was done with the data collected through a practical software project. All of these measures have highly significant correlations with the management data. Application of complexity measures to software development management is discussed and a method for the detection of anomalous modules in a program is proposed.

[Suns77] Abstract: It is becoming increasingly important that communication protocols be formally specified and verified. This paper describes a particular approach - the state transition model - using a collection of mechanically supported specification and verification tools incorporated in a running system called AFFIRM. Although developed for the specification of abstract data types and the verification of their properties, the formalism embodied in AFFIRM can also express the concepts underlying state transition machines. Such models easily express most of the events occurring in protocol systems including those of the users, their agent processes, and the communication channels. The paper reviews the basic concepts of state transition models, and the AFFIRM formalism and methodology and describes their union. A detailed example, the alternating bit protocol, illustrates various properties of interest for specification and verification. Other examples explored using this formalism are briefly described and the accumulated experience is discussed.

[Symo88] Abstract: The method of Function Point Analysis was developed by Allan Albrecht to help measure the size of a computerized business information system. Such sizes are needed as a component of the measurement of productivity in system development and maintenance activities, and as a component of estimating the effort needed for such activities. Close examination of the method shows certain weaknesses, and the author proposes a partial alternative. The paper describes the principles of this "Mark II" approach, the results of some measurements of actual systems to calibrate the Mark II approach, and conclusions on the validity and applicability of function point analysis generally.

[Szul84] Introduction: The manner in which software for DoD applications is developed is undergoing evolutionary change with the introduction of Ada and its support tools. This change has been prompted by the desire to increase software quality and developer productivity. Although design-aid tools, and techniques for measuring software quality have been of interest to the research community for some time now, the user community has only recently expressed a need for this technology as evidenced by the Software Technology for Adaptable and Reliable Systems (STARS) program. An important part of the STARS program is the development of metrics to measure the quality of both the software development process and software products. Even though the STARS focus is not Ada, tools and techniques developed through this effort will likely become part of the Ada Programming Support Environments (APSEs).

This paper reports on work done in investigating the use of Ada as a Program Design Language (PDL),

and the evaluation of Ada designs with a design metric. The first section provides background and describes the context for the work. The second section defines the Halstead metrics and discusses their application during the design phase. The third section discusses using Ada as a Program Design Language. The fourth section presents an example which illustrates the usefulness of the design metrics on the Ada PDL design medium. Finally, the conclusions of this work are presented.

[Tai80] Abstract: This paper explores the *testing complexity* of several classes of programs, where the testing complexity is measured in terms of the number of test data required for demonstrating program correctness by testing. It is shown that even for very restrictive classes of programs, none of the commonly used criteria, namely having every statement, branch, and path executed at least once, is nearly sufficient to guarantee absence of errors.

Based on the study of testing complexity, this paper proposes two new test criteria, one for testing a path and the other for testing a program. These new criterion suggest how to select test data to obtain confidence in program correctness beyond the requirement of having each statement, branch, or path tested at least once.

[Tai85a] Abstract: In this paper a type of error in concurrent software, called synchronization error, is defined. How to analyze a concurrent specification or design in order to detect synchronization errors is discussed.

[Tai85c] Abstract: Repeated executions of a concurrent program with the same input may exercise different paths in this program, thus making concurrent programs more difficult to test than sequential programs. This paper addresses several fundamental issues on the testing of concurrent programs. A type of error in concurrent programs, called synchronization error, is formally defined. To detect such errors, a new form of test case is proposed, which consists of an input and a synchronization sequence and is called an IN_SYN test case. How to generate IN_SYN test cases for a concurrent program is discussed. In order to execute an IN_SYM test case, the problem of *reproducing a sequence of synchronizations* between concurrent processes arises, which is referred to as the reproducible testing problem. Four basic approaches to solving this problem are presented.

[Tai86] Abstract: Repeated executions of a concurrent Ada program with the same input may exercise different sequences of rendezvous, thus making concurrent Ada programs more difficult to test than sequential Ada programs. The reproducible testing problem for Ada is how to reproduce a sequence of rendezvous of a concurrent Ada program. This problem exists not only for debugging concurrent Ada programs, but also for determining the correctness of concurrent Ada programs.

In this paper, we present a solution to the reproducible testing problem for an arbitrary concurrent Ada program. This solution transforms a concurrent Ada program P into P' such that the reproduction of a sequence of rendezvous, say S, of P with input X requires exactly one execution of P' with (X,S) as input. The proposed solution can be easily automated.

[Taka89] Abstract: Accuracy in program error prediction is a major problem in quality control of a large-scale software system. This paper presents a model to estimate the number of errors remaining in a program at the beginning of the testing phase of development. In the first part of the study, the relationships between the errors occurring in a program and the various factors which have an effect on software development, such as programmer's skill, are statistically analyzed. The model is then derived by using the factors significantly identified in the analysis. This empirical study is based on data collected during the development of large-scale software systems. Results of the study indicate that factors such as frequency of program specification change, programmer's skill, and volume of program design documentation are significant and that the model based on these factors is more reliable than conventional error prediction methods based on program size alone.

[Taus88] Abstract: This article contains the results of initial research work performed to extend the applicability of McCabe's Cyclomatic Complexity Metric for the analysis of Ada software. Having proved useful both as a logical measurement technique and as a testing aid, the Ada Complexity Extension (ACE) is proposed for general acceptance as a standard to provide a useful metric that may assist in improving the quality of Ada software

programs.

[Tayl78b] Abstract: This paper describes the overall design of some modular capabilities for error detection testing, verification, and documentation of concurrent process HAL/S programs. The work described draws upon many ideas first advanced in building tools for single process software. In this paper, these ideas are significantly extended and adapted to realize the power of these tools for concurrent software. Particular attention is paid to the design of static data flow analysis capabilities for concurrent software.

[Tayl80a] Abstract: The increasing cost of computer system failure has stimulated interest in improving software reliability. One way to do this is by adding redundant structural data to data structures. Such redundancy can be used to detect and correct (structural) errors in instances of a data structure. The intuitive approach of this paper, which makes heavy use of examples, is complemented by the more formal development of the companion paper, "Redundancy in Data Structures: Some Theoretical Results."

[Tayl80b] Abstract: Algorithms are presented for detecting errors and other anomalies in programs which use synchronization constructs to implement concurrency. The algorithms employ data flow analysis techniques. First used in compiler object code optimization, the techniques have more recently been used in the detection of variable usage errors in single process programs. By adapting these existing algorithms, the same class of variable usage errors can be detected in concurrent process programs. Important classes of errors unique to concurrent process programs are also described, and algorithms for their detection are presented.

[Tayl82a] Abstract: This paper sets some context, raises issues, and provides [the authors] initial thinking on the characteristics of effective rapid prototyping techniques.

After discussing the role rapid prototyping techniques can play in the software lifecycle, the paper looks at possible technical approaches including: heavily parameterized models, reusable software, rapid prototyping languages, prefabrication techniques for system generation, and reconfigurable test harnesses.

The paper concludes that a multi-faceted approach to rapid prototyping techniques is needed if we are to address a broad range of applications successfully — no single technical approach suffices for all potentially desirable applications.

[Tayl82b] Abstract: A common paradigm for the development of process-control or embedded computer software is to do most of the implementation and testing on a large host computer, then retarget the code for final checkout and production execution on the target machine. The host machine is usually large and provides a variety of program development tools, while the target may be a small, bare machine. A difficulty with the paradigm arises when the software developed has real-time constraints and is composed of multiple communicating processes. If a test execution on the target fails, it may be exceptionally tedious to determine the cause of the failure. Host machine debuggers cannot normally be applied, because the same program processing the same data will frequently exhibit different behavior on the host. Differences in processor speed, scheduling algorithm, and the like, account for the disparity. This paper proposes a partial solution to this problem, in which the errant execution is reconstructed and made amenable to source language level debugging on the host. The solution involves the integrated application of a static concurrency analyzer, an interactive interpreter, and a graphic program visualization aid. Though generally applicable, the solution is described here in the context of multi-task real-time Ada programs.

[Tayl83a] Abstract: Developing and verifying concurrent programs presents several problems. A static analysis algorithm is presented here that addresses the following problems: how processes are synchronized, what determines when programs are run in parallel, and how errors are detected in the synchronization structure. Though the research focuses on Ada, the results can be applied to other concurrent programming languages such as CSP.

[Tayl83b] Summary: Foundational to verification of some aspects of communicating concurrent systems is

knowledge of the synchronization which may occur during execution. The synchronization determines the actions that may occur in parallel, may determine program data flow, and may also lead to inherently erroneous situations (e.g., deadlock). This paper formalizes the notion of the synchronization structure of concurrent programs that use the rendezvous (or similar mechanism for achieving synchronization). The formalism is oriented towards supporting verification as performed by automated static program analysis. Complexity results are presented which indicate what may be expected in this area and which also shed light on the difficulty of correctly constructing concurrent systems. Specifically, most of the analysis tasks considered are shown to be intractable.

[Tayl83c] Abstract: Stand-alone techniques for the analysis and testing of the synchronization structure of concurrent programs have recently been developed. These techniques are able to detect, for example, task blockage, including deadlock. Static analysis provides firm results, but has limited applicability and is potentially expensive. Dynamic analysis makes fewer assumptions, but its assurances are not as strong. This paper presents strategies whereby the two can be employed jointly to advantage. Dynamic analysis can be used to further investigate results from static analysis, and vice versa. Their joint use can be facilitated by an appropriate implementation, some principles for which are outlined.

[Tayl85] Abstract: Conceptual simplicity, tight coupling of tools, and effective support of host-target software development will characterize advanced Ada programming support environments. Several important principles have been demonstrated in the Arcturus system, including template-assisted Ada editing, command completion using Ada as a command language, and combining the advantages of interpretation and compilation. Other principles, relating to analysis, testing, and debugging of concurrent Ada programs, have appeared in other contexts. This paper discusses several of these topics, considers how they can be integrated, and argues for their inclusion in an environment appropriate for software development in the late 1980's.

[Tayl86a] Abstract: Though structural testing techniques are among the weakest available with regard to developing confidence in sequential programs, they are not without merit. This paper extends the notion of structural testing criteria to concurrent programs and proposes tools supporting structural testing techniques. Requisite support tools include a static concurrency analyzer and either a program transformation system or a powerful run-time monitor. Also helpful is a controllable run-time scheduler. The techniques proposed will work for Ada or CSP-like languages. Best results will be obtained for programs having only static naming of task objects.

[Tayl86b] Abstract: The research objectives of the Arcadia project are twofold: discovery and development of environment architecture principles and creation of novel software development tools. The environment architecture is intended to reconcile extensibility with the often conflicting goal of integration, including both a uniform user interface and coordination and management of tools and software objects. Work on tools is focused on analysis of software objects at every stage of software development and maintenance, and is especially aimed at analysis of concurrent and real-time software. A prototype environment architecture and toolset is being developed in Ada, to support Ada software development. The authors describe the research objectives and approaches being taken, the organization of the research endeavor, and current status of the work.

[Tayl88] Abstract: Early software environments have supported a narrow range of activities (*programming* environments) or else been restricted to a single "hard-wired" software development process. The Arcadia research project is investigating the construction of software environments that are tightly integrated, yet flexible and extensible enough to support experimentation with alternative software processes and tools. This had led us to view an environment as being composed of two distinct, cooperating parts. One is the *variant* part, consisting of process programs and the tools and objects used and defined by those programs. The other is the fixed part, or *infrastructure*, supporting creation, execution, and change to the constituents of the variant part. The major components of the infrastructure are a process programming language and interpreter, object management system, and user interface management system. Process programming facilitates precise definition and automated support of software development and maintenance activities. The object management system provides typing,

relationships, persistence, distribution and concurrency control capabilities. The user interface management system mediates communication between human users and executing processes, providing pleasant and uniform access to all facilities of the environment. Research in each of these areas and the interaction among them is described.

[Teic77] Abstract: PSL/PSA is a computer-aided structured documentation and analysis technique that was developed for, and is being used for, analysis and documentation of requirements and preparation of functional specifications for information processing systems. The present status of requirements definition is outlined as the basis for describing the problem which PSL/PSA is intended to solve. The basic concepts of the Problem Statement Language are introduced and the content and use of a number of standard reports that can be produced by the Problem Statement Analyzer are briefly described.

The experience to date indicates that computer-aided methods can be used to aid system development during the requirements definition stage and that the main factors holding back such use are not so much related to the particular characteristics and capabilities of PSL/PSA as they are to organizational considerations involved in any change in methodology and procedure.

[Teit81] Abstract: Interlisp is a programming environment based on the Lisp programming language. In widespread use in the artificial intelligence community, Interlisp has an extensive set of user facilities, including syntax extension, uniform error handling, automatic error correction, an integrated structure-based editor, a sophisticated debugger, a compiler, and a filing system. Its most popular implementation is Interlisp-10, which runs under both the Tenex and Tops-20 operating systems for the DEC PDP-10 family. Interlisp-10 now has approximately 300 users at 20 different sites (mostly universities) in the US and abroad. It is an extremely well documented and well maintained system.

Interlisp has been used to develop and implement a wide variety of large application systems. Examples include the Mycin system for infectious disease diagnosis, the Boyer-Moore theorem prover, and the BBN speech understanding system.

This article describes the Interlisp environment, the facilities available in it, and some of the reasons why Interlisp developed as it has.

[Teit84] Introduction: This paper introduces the reader to many of the salient features of the Cedar Programming Environment, a state-of-the-art programming system that combines in a single integrated environment: high quality graphics, a sophisticated editor and document preparation facility, and a variety of tools for the programmer to use in the construction and debugging of his programs. The Cedar programming language is a strongly-typed, compiler-oriented language of the Pascal family. What is especially interesting about the Cedar project is that it is one of the few examples where an interactive, experimental programming environment has been built for this kind of language. In the past, such environments have been confined to dynamically typed languages like Lisp and Smalltalk.

The paper attempts to give the reader the feel of the Cedar system by emulating a live demonstration. The demonstration is actually taken from a video tape of such a live demo; the sequence of events, as well as the dialogue, is fairly close to what a viewer of this tape would see and hear. Numerous snapshots of the display taken at various points during the session simulate the visual information contained in the tape. Text that would actually appear on the display during the demonstration - either because the user typed it or the system printed it - will appear in this paper in a distinguished font. The explanations that the demonstrator would give will be in the normal font. Comments that would be distracting during a live demonstration but are appropriate for the paper are included as footnotes.

[Thay75] Abbreviated Introduction: The need for improving the reliability of delivered software is becoming increasingly obvious to both the purchasers and producers of today's software systems. As noted by Boehm, the records show many examples of software systems which, when delivered for operational use, either performed in a degraded fashion or failed to perform at all. The results are higher software costs and delays in operational usage.

In a study being performed by TRW for the Rome Air Development Center, data from four large software systems are being analyzed to determine the types of errors found in software during testing. The objective is principally to recommend new development or test techniques for the detection and prevention of software errors, but we are also attempting to model software reliability. In the course of supplying real data descriptive of software reliability and for model evaluation, we have had to determine (1) what data are generally available, (2) methods for collecting and storing these data, (3) methods for describing software errors, (4) methods for characterizing the software, and the development and test processes in quantitative terms, and finally (5) methods of analysis. Although the projects studied have varied greatly in size, language, operating mode, and structure, the data available during the development process were similar for each project: error data, recorded in various forms of software problem reports (SPR) and ancillary project data needed to understand and support analysis of the error data. Although the data were not generated specifically for the study, we found that we could do much to quantify software reliability and the characteristics of the software itself, as well as improving our understanding of both the software and the development process. Some results of the Software Reliability Study will be presented to illustrate the benefits of software reliability data collection and analysis. Also presented are some recommendations for identifying data that need collecting.

[Thay80] Abbreviated Introduction: Nearly every software engineering development project is plagued with numerous problems leading to late delivery, cost overruns, or unsatisfied customers. Often, these problems are technical. However, just as often, they are managerial.

Although both the technological and managerial aspects of software engineering were recognized at about the same time, improvements and developments in management have not kept pace with advances in the technology. The technology of software engineering as a well-defined discipline is relatively new; however, software engineering has progressed to the point where many major issues regarding software production have been identified, and considerable progress in addressing these issues has been made. Practical working tools to support improved production are commonly available, and their design and generation have become a recognized topic for university instruction.

Software engineering project management has not enjoyed the same progress. While it might be argued that SEPM has been defined, it is far from a recognized discipline. The major issues and problems of SEPM have not been agreed on by the computing community as a whole, and consequently, priorities for addressing them have not been widely established. Furthermore, research in this area has been scant.

[Theb84] Abstract: Recent work conducted by members of the Purdue Software Metrics Research group has focused on the complexity associated with coordinating the activities of persons involved in large-scale programming efforts. A resource model is presented which is designed to reflect the impact of this complexity on the economics of software development. The model is based on a formulation in which development effort is functionally related to measures of product size and manloading. The particular formulation used is meant to suggest a logical decomposition of development effort into components related to the independent programming activity of individuals and to the overhead associated with the required information flow within a programming team. The model is evaluated in light of acquired data reflecting a large number of commercially developed software products from two separate sources. Additional sources of data are actively being sought. Although strongly analytic in nature, the model's performance is, for the available data, at least as good in accounting for the observed variability in development effort as some highly publicized empirically based models of comparable complexity. It is argued, however, that the model's principle strength lies *not* in its data fitting ability, but rather in its straightforward and intuitively appealing representation of relationships involving manpower, time, and effort.

[Thom80] Abstract: This paper deals with the statistics of estimating the software reliability of complex real-time systems where an electronic digital computer and associated computer programs are essential elements of system design and function. Testing is conducted in the operating environment or a simulated environment related to the operating environment in some way. The procedure is Bayesian so that improvement of reliability estimation is realized in a formal and convenient way as more and more test data are accumulated. The method provides for estimating a) both hardware and software components of total system reliability and b) Bayesian interval limits

using existing analytic techniques developed by the authors and others. The results apply to measurement and prediction of reliability performance, to acceptance testing, and to contractual definition and implementation of software warranty provisions for embedded computer systems.

The Bayesian method of software-hardware reliability estimation presented here exhibits the following unique features:

— The use of a prior p on the probability that the software contains errors. This prior is updated as test failure data are accumulated. Only a p of 1 (software known to contain errors) corresponds to a case already treated in the literature.
— Hardware, software, and unknown/ambiguous source failure data are combined to yield a system reliability estimation.
— A decision-rule treatment is developed for the continuation or termination of testing on the basis of specification of consumer and producer risks and observed test results.

[Tich86] Abstract: With current compiler technology, changing a single line in a large software system may trigger massive recompilations. If the change occurs in a file with shared declarations, all compilation units depending upon that file must be recompiled to assure consistency. However, many of those recompilations may be redundant, because the change may affect only a small fraction of the overall system.

Smart recompilation is a method for reducing the set of modules that must be recompiled after a change. The method determines whether recompilation is necessary by isolating the differences among program modules and analyzing the effect of changes. The method is applicable to languages with and without overloading. A prototype demonstrates that the method is efficient and can be added with modest effort to existing compilers.

[Tisc83] Abstract: This paper describes how MAP, a tool for understanding software, combines static analysis, some dynamic features, and an interactive presentation to aid programmers in debugging. Static analysis of the sort produced in optimizing compilers could provide programmers with useful information that they cannot get from dynamic debuggers. The challenge for designers of static analysis tools is to present the information in a useful form.

[Triv80] Abstract: This paper addresses the problem of validating the reliability of computer systems used in life-critical applications. Due to extremely high reliability requirements, traditional validation methods based on lifetesting are no longer applicable. A validation approach based on a judicious combination of logical proofs, analytical models, and experimental testing is advocated. The role of Markov reliability models in the validation process is discussed and a taxonomy of validation techniques is presented.

[Troy81] Abstract: The purpose of this study is to investigate the possibility of providing some useful measures to aid in the evaluation of software designs. Such measurements should allow some degree of predictability in estimating the quality of a coded software product based upon its design and should allow identification and correction of deficient designs prior to the coding phase, thus providing lower software development costs. The study involves the identification of a set of hypothesized measures of design quality and the collection of these measures from a set of designs for a software system developed in industry. In addition, the number of modifications made to the coded software that resulted from these designs was collected. A data analysis was performed to identify relationships between the measures of design quality and the number of modifications made to the coded programs. The results indicated that module coupling was an important factor in determining the quality of the resulting product. The design metrics accounted for roughly 50-60% of the variability in the modification data, which supports the findings of previous studies. Finally, the weaknesses of the study are identified and proposed improvements are suggested.

[Troy86] Abstract: Over the past decade, a set of models derived from the application of conventional reliability theory to software engineering has been proposed with regard to the evaluation of program reliability. Observations of operating software have shown that these models are not sufficient to account for operational reliability. This limitation requires a cautious utilization of every model: each reliability evaluation must be considered as a

special case which must be based upon a statistical analysis preceding any modeling. This implies suitable methods and means are needed. The purpose of this paper is to propose a stepwise statistical methodology for the study of operating system reliability and associated tools. An example of the application of this method for the ARGOS center of CNES is presented.

[Tsal86] Abstract: This paper describes the concepts, functions and user interface of the tool for unit test construction and execution. This tool, the Interactive Unit Test Facility (IUTF), addresses some of the major concerns in the unit testing process. The first of these is the execution of the unit and reporting its results in terms of success/failure and coverage measures. The other concern, sometimes more painful and time consuming for the programmer, is the preparation and maintenance of test cases for execution.

IUTF performs static and dynamic testing of the unit provided all results from the analysis and execution stage are stored in a central data base. The important design notions such as testing environment, scaffolding, and test drivers and construction mechanisms are introduced in the paper, and the transformation of internal functions of the unit testing tool into usable and consistent interfaces (via the predefined screen hierarchy) is described.

[Turn80] Summary: Choosing the right program structures can lead to better programs and modular design can make large programs more manageable. This paper reviews the possible structural relationships between the modules of a program and generates a tentative morphology of program structure types. It concludes that, with some exceptions, the hypothetical pure tree structure is the best choice for most data processing applications.

[Ullm73] Summary: We give two algorithms for computing the set of available expressions at entrance to the nodes of a flow graph. The first takes $0\ (mn)$ steps on a program flow graph (one in which no node has more than two successors), where n is the number of nodes and m the number of expressions which are ever computed. A modified version of this algorithm requires $0\ (n^2)$ steps of an extended type, where bit vector operations are regarded as one step. We present another algorithm which works only for reducible flow graphs. It requires $0\ (n \log n)$ extended steps.

[Unde63] Introduction: When a scientific program is to be used by physicists as an aid in their investigation, the programmer must pay careful attention to the problem of producing the program in a suitable form. It may happen that parts of the program which the programmer may prefer to regard as peripheral activity, such as input and output processes, then assume a major importance and occupy much of his time and program; the numerical method becomes a small box which works well most of the time and is a great nuisance when it doesn't.

The problem presented by the construction of a good input section is severe. Best efforts to date fall far short of perfection, and this will be attained only when the experience gained by use of a program is stored, not by the user who runs problems on it, but within the program itself, ready for intelligence use by the program when a problem is presented to it.

[Vale89] Abstract: The practice of measuring software is increasingly seen as a valuable tool in the overall development of high-quality software projects. Software measurement attempts to use known, quantifiable, objective, and subjective measures to compare and profile software projects and products. To compute these measures effectively, data that characterize the software project and product are needed. This paper covers aspects of data collection and software measurement as they have bee applied by one particular organization, the Software Engineering Laboratory (SEL). The measurement results include the experiences and lessons learned through numerous experiments conducted by the SEL on nearly 60 flight dynamics software projects. These experiments have attempted to determine the effect of various software development technologies on overall software project quality and on specific measures such as productivity, reliability, and maintainability.

[Vali84] Abstract: Humans appear to be able to learn new concepts without needing to be programmed explicitly in any conventional sense. In this paper we regard learning as the phenomenon of knowledge acquisition in the absence of explicit programming. We give a precise methodology for studying this phenomenon from a

computational viewpoint. It consists of choosing an appropriate information gathering mechanism, the learning protocol, and exploring the class of concepts that can be learned using it in a reasonable (polynomial) number of steps. Although inherent algorithmic complexity appears to set serious limits to the range of concepts that can be learned, we show that there are some important nontrivial classes of propositional concepts that can be learned in a realistic sense.

[Vemu80] **Abstract:** Software and its development are complex. The complexity stems from the multiplicity of objectives and attributes that one has to work with during its development. Human comprehension of multiple objectives and attributes can be aided by displaying the relevant data on a two-dimensional plane. Several display techniques, and in particular the so called snowflakes and Chernoff faces, are discussed and their utility in software research explored. Examples using real and hypothetical data are presented to illustrate the suitability of these pictures.

[Vern89] **Abstract:** Because Function Point Analysis (FPA) has now been in use for a decade, and in spite of its increasing popularity has met with some recent criticisms, it is time to review how appropriate it still is for today's technologies. A critical review of the FPA approach examines in particular the pioneering and continuing work of Albrecht and more recent work by Symons. Technological dependencies in FPA-type metric for a new software technology is given. A model for the calibration of FPA-type metrics for new technologies in terms of a reference technology is also presented. Such calibration is essential for comparative productivity studies. The role of module estimation in exposing parts of the 'anatomy' of the FPA approach is investigated. The derivation and calibration models are applied to a significant case study in which a new FPA-type metric suited to a particular software development technology is derived, calibrated and compared with other published versions of FPA metrics.

[Vess83] **Abstract:** An empirical study of 447 operational commercial and clerical Cobol programs in one Australian organization and two U.S. organizations was carried out to determine whether program complexity, programming style, programmer quality, and the number of times a program was released affected program repair maintenance. In the Australian organization only program complexity and programming style were statistically significant. In the two U.S. organizations only the number of times a program was released was statistically significant. For all organizations repair maintenance constituted a minor problem: over 90 percent of the programs studied had undergone less than three repair maintenance activities during their lifetime.

[Voge80] **Abstract:** This paper describes the automated testing tool SADAT, which supports the testing of single Fortran modules. The different functions which are integrated in this system are explained, the usage of the tool is demonstrated, and some output results are presented. The special benefits of the SADAT system are summarized. The history and the present status of the system are outlined. Finally, a listing of further reference material and information on the program availability are included.

[Vosb84] **Abstract:** Fourteen factors that influence the efficiency of programming projects were identified in a corporate-wide study of 44 ITT programming projects in nine countries. Productivity factors were classified according to project management's ability to control them. Product-related factors are not generally under the control of project management. They describe intrinsic properties of the programming product and tend to place limitations on achievable productivity. Project-related factors, on the other hand, are controllable by project management to varying degrees. These factors provide real opportunities for productivity improvement. The analysis indicates that productivity variation is almost equally attributable to product-related and project-related factors.

[Vouk85c] **Abstract:** Software fault tolerance mechanisms commonly used today suffer from their inability to successfully cope with correlated failures of components of a fault-tolerant software (FTS) system. In this paper methods for computing the reference and observed distributions of multiple component failures (MCF's) of a FTS are given. A MCF of category k refers to existence of a test case for which exactly k components of a FTS

system fail. The reference distribution is based on the response of all components on a randomly selected test set, and the assumption that the conditional intercomponent responses are mutually independent. Identification of correlated failures and the effectiveness of random testing for detecting correlated failures is discussed through comparison of the reference and observed MCF distributions for an experimental FTS system.

[Vouk86a] Abstract: A major weakness of software fault tolerance mechanisms commonly discussed today is their inability to cope successfully with correlated failures of components of a fault-tolerant software (FTS) system. When correlated errors are present, the probability that a FTS system fails may become unacceptably large. The results of a FTS experiment are used to show deficiencies of the simple random testing approach in the context of FTS testing. Inter-version failure dependence was detected in the experiment, and the data indicate that in high reliability FTS components a considerable percentage of correlated failures occur in the domain of extremal or special input values, a region not excited by simple random sampling of the input space. The use of carefully designed test cases as a supplement to random testing, as well as use of structure based testing is recommended.

[Vouk86b] Abstract: Common approaches to software fault-tolerance depend on redundancy of critical software components. Six functionally equivalent programs were tested with specification based random and extremal/special value (ESV) test cases. Statement and branch coverage were used to measure and compare the attained testing effectiveness. It was observed that both measures reached a nearly steady state value after 25 to 75 random test cases. Coverage saturation curves appear to follow an exponential growth model. However, the steady state values for branch coverage of different components, but the *same* input cases, differed by as much as 22%. The effect is the result of the differences in the detailed structure of the components. Improvement in coverage provided by the random test data, after the ESV cases were executed, was only about 1%. Results indicate that extensive random testing can be a process of diminishing returns, and that in the FTS context functional ("black box") testing can provide a very uneven execution coverage of the functionally equivalent software, and therefore should be supplemented by structure based testing.

[Wahl88] Abstract: As a consequence of timing considerations, program execution behavior on a distributed system may not be reproducible from one execution to the next. The situation is exacerbated when the architecture of the distributed system is non von Neumann, as in the case of a dataflow machine. This fact has implications for the testing and debugging of dataflow programs. In this paper a distributed debugging methodology for dataflow architectures is presented. A graphical debugging simulator for a dataflow machine is being developed to implement this methodology. This debugging simulator allows the user to debug compiled high-level dataflow programs written for the machine. The ideas of the debugging methodology are outlined and the debugging simulator is described. Special emphasis is paid to the multi-pass feature of the debugging simulator which solves the nonreproducibility problem of distributed debuggers and allows the user to execute the program more than once with the identical instruction sequence to be sure that a fault has been removed.

[Waka89] Abbreviated Introduction: Generally, telecommunications software must handle a complex, large-scale protocol modeled as extended finite-state machines. Much research has been done on how to specify telecommunications software with formal specification languages. However, these research results have not been completely successful for three main reasons:

1. The methods devised cannot detect errors in individual finite-state machines.
2. They cannot detect protocol errors, such as missing signal-reception definitions, in large-scale protocol specifications.
3. They do not have functions to improve the legibility of manually drafted specifications.

To overcome these defects, we proposed new validation, verification, and simplification methods for telecommunications specifications. At Kokusai Denshin Denwa Co., we have developed a prototype specification support system, Escort, that integrates these proposed methods.

[Wake88] Abstract: Computer scientists are continually attempting to improve software system development.

Systems are developed in a top-down fashion for better modularity and understandability. Performance enhancements are implemented for more speed. One area in which a great deal of effort is being devoted is software maintenance. Brooks estimates that fifty percent of the development costs of a software system is for maintenance activities. Since a large portion of the effort of a system is devoted for maintenance, it is reasonable to assume that driving down maintenance costs would drive down the overall cost of the system. Measuring the complexity of a software system could aid in this attempt. By lowering the complexity of the system or of subsystems within the system, it may be possible to reduce the amount of maintenance necessary. Software quality metrics were developed to measure the complexity of software systems. This study relates the complexity of the system as measured by software metrics to the amount of maintenance to that system. We have developed a model which uses several software quality metrics as parameters to predict maintenance activity.

[Walk81] Preface: The purpose of the *Paradigmatic Approach* is to provide a new image for conceptualizing the software development cycle. It is believed that this new image will endanger methodologies that predictably produce reliable software systems. This text is not a cookbook of techniques. It does not attempt to direct action through prescribing specific behavior patterns. This text does, however, present an integrated image for organizing behavior and a universal metric for evaluating that behavior. The reader will be exposed to a powerful image, a paradigm, which provides an integrated perception of software development. This paradigm will help him to organize and judge technical behavior in a consistent and productive manner. The consistent behaviors which result from paradigmatic thinking are termed "the paradigmatic approach" and will facilitate the evolution of software management from a craft to an engineering discipline.

[Wall89] Abbreviated Introduction: Verification and validation is one of the software-engineering disciplines that help build quality into software. V&V is a collection of analysis and testing activities across the full life cycle and complements the efforts of other quality-engineering functions. This overview article explains what V&V is, shows how V&V groups' efforts relate to other groups' efforts, describes how to apply V&V, and summarizes evaluations of V&V effectiveness.

[Wals77a] Overview: Improvements in programming technology have paralleled improvements in computing system architecture and materials. Along with increasing knowledge of the system and program development processes, there has been some notable research into programming project measurement, estimation, and planning. Discussed is a method of programming project productivity estimation. Also presented are preliminary results of research into methods of measuring and estimating programming project duration, staff size, and computer cost.

[Walt79] Abstract: With the increasing complexity of the software systems being developed today and the requirement to develop them within a short schedule, there is greater emphasis than before on a strong quality management program. In such a program it is essential that we know how to specify and measure software quality so that we can ensure the system meets our overall life cycle objective. It is important not only from the system performance point of view but also in cost.

The role of the manager in a software quality program is important throughout the entire development phase of our program. The impact of the manager's decisions during this phase will be felt not only during operation and maintenance but also during future acquisitions that interface with the system or that incorporate existing software from the current development.

This chapter addresses how both the acquisition manager and the development program manager can identify which quality factors are important and how metrics of these factors can be applied in the software quality management program. (The acquisition manager and the development program manager titles throughout this chapter refer to the organizations rather than the persons, per se.) This approach of applying metrics is based upon the concept of software quality and of the associated metrics described in the previous chapter.

[Wamp85] Abstract: The topic of this thesis is the development of and the results obtained from a system which analyzes Ada tasking programs in order to identify potential concurrency related programming anomalies. The

static data flow used in this thesis are described in great detail in a paper by Taylor.

The major purpose for this undertaking was to characterize the actual size necessary to store the concurrency related information from some sample programs. This goal was deemed appropriate since many of the techniques used in the analysis process have already been shown to be in the set of NP-complete programs. The method employed toward this end was to build a prototype version of the Static Concurrency Analysis system with this sufficient instrumentation in order to gather statistics on the size of several of the data structures involved in the process, and then to run some sample programs through this tool.

It will be seen that the size of the major structure built, the Concurrency State Graph (CSG), grows exponentially in the number of tasks that exist in the program being analyzed for all example programs thus far run through the prototype system. This CSG structure models all possible tasking related program states that a given Ada program could possibly be in as well as the successor and predecessor relationships between these states.

The SCA system currently does not accept the full Ada language and it appears to be a less than trivial task to extend the prototype system so that all Ada tasking programs are analyzable.

[Warn72] Introduction: With the billions of dollars in installed computer equipment deployed worldwide and with vast sums needed to operate, program, and maintain this hardware, computer users are increasingly aware of the need to improve the efficiency of data processing operations.

Corporations use computers to handle many tasks. The range of tasks and size of the computers increase constantly. But corporations do not know how to evaluate the efficiency of their data processing operations. Inability to make an accurate assessment has kept management fearful of the entire data processing experience and has resulted, generally, in a hands-off attitude. This tail-wags-the-dog situation places a particularly heavy burden on that portion of management directly responsible for the data processing operation. They have to make recommendations on new and/or added equipment, but they lack objective techniques for evaluating the DP operation and projecting needs. This paper reviews some of the issues faced and alternative techniques for assessing system performance.

[Warr82] Maintenance of software is a major problem that the data processing industry faces today. This paper describes MAP, a tool, that addresses the problems of software maintenance by helping programmers to understand their programs.

[Wate79] Abstract: This paper presents a method for automatically analyzing loops, and discusses why it is a useful way to look at loops. The method is based on the idea that there are four basic ways in which the logical structure of a loop is built up. An experiment is presented which shows that this accounts for the structure of a large class of loops. The paper discusses how the method can be used to automatically analyze the structure of a loop, and how the resulting analysis can be used to guide a proof of correctness for the loop. An automatic system is described which performs this type of analysis. The paper discusses the relationship between the structure building methods presented and programming language constructs. A system is described which is designed to assist a person who is writing a program. The intent is that the system will cooperate with a programmer throughout all phases of work on a program and be able to communicate with the programmer about it.

[Webe83] Abstract: This paper summarizes techniques for designing and implementing source-level interactive debuggers for concurrent programs. Facilities common to source-level interactive debuggers have been adapted to meet the needs of a concurrent programming environment. Of special interest are those debugging facilities which allow the programmer to monitor and influence the execution of concurrent processes.

[Wegb74] Abstract: Current methods for mechanical program verification require a complete predicate specification on each loop. Because this is tedious and error prone, producing a program with complete, correct predicates is reasonable difficult and would be facilitated by machine assistance. This paper discusses techniques for mechanically synthesizing loop predicates. Two classes of techniques are considered: (1) heuristic methods which derive loop predicates from boundary conditions and/or partially specified inductive assertions: (2)

extraction methods which use input predicates and appropriate weak interpretations to obtain certain classes of loops predicates by an evaluation on the weak interpretation.

[Wegb75] Abstract: One means of analyzing program performance is by deriving closed-form expressions for their execution behavior. This paper discusses the mechanization of such analysis, and describes a system, Metric, which is able to analyze simple Lisp programs and produce, for example, closed-form expressions for their running time expressed in terms of size of input. This paper presents the reasons for mechanizing program analysis, describes the operation of Metric, explains its implementation, and discusses its limitations.

[Wegb76] Abstract: This paper is concerned with proving properties of programs which use data structures. The goal is to be able to prove that all instances of a class (e.g., as defined in Simula) satisfy some property. A method of proof which achieves this goal, *generator induction*, is studied and compared to other proof rules and methods: inductive assertions, recursion induction, computation induction, and, in some detail, structural induction. The paper concludes by using generator induction to prove a characteristic property of an implementation of hashtables.

[Wegb77] Abstract: Most current approaches to mechanical program verification transform a program and its specifications into first-order formulas and prove these formulas valid. Since first-order predicate calculus is not decidable, such approaches are inherently limited. This paper proposes an alternative approach to program verification: correctness proofs are constructively established by proof justification written in algorithmic notation. These proof justifications are written as part of the program, along with the executable code and correctness specifications. A notation is presented in which code, specifications, and justifications are interwoven. For example, if a program contains a specification *{exists x P(x)}*, the program also contains a justification that exhibits the particular value of *x* that makes *P* true. Analogously, justifications may be used to state how universally quantified formulas are to be instantiated when they are used as hypotheses. Programs so justified may be verified by proving quantifier-free formulas. Additional classes of justifications serve related ends. Formally, justifications reduce correctness to a decidable theory. Informally, justifications establish the connection between executable code and correctness specifications, documenting the reasoning on which the correctness is based.

[Wegn79] Introduction and Overview: The primary purpose of this book is to provide an understandable but nontrivial description by active research workers of concepts and research issues in principal subareas of software technology. It should be useful to both the specialist and the technical layman as a source of factual information about research issues and can serve as a starting point for discussions of what to do next. We hope it will make practitioners aware of the practical contributions of research, make researchers aware of the needs of technology, and serve to stimulate greater collaboration between practitioners and research workers. An even more ambitious objective is to encourage dialogue among research workers in different areas (such as computer architecture, programming languages and data base management) so that the basis for an integrated approach to computer systems can be established. Last but not least, this study may be useful to funding agencies and other policy-making bodies in making policy decisions concerning future support of research.

The first four chapters (Part I) consider the nature of the software problem and describe concepts and tools for managing large software systems. The remaining sixteen chapters (Part II) describe and analyze specific research areas. In order to stimulate discussion, over 50 discussion items further explore specific research areas or offer novel and sometimes controversial points of view.

[Weid86] Introduction: Most of the work in the evaluation of software development environments has fallen into one of three categories. First, there are evaluations of particular components such as compilers, editors, or window managers. These evaluations are useful in their own right, but they fail to consider global aspects of the environment or how components interact. Second, there are evaluations of particular environments. These studies usually consider the tools available in that environment but they do not lend themselves to cross environment comparisons. Third, there are lists of questions and criteria without the details of how to answer the questions or apply the criteria. These lists are useful, but are frequently difficult to apply in practice.

The purpose of this paper is to address the shortcomings of the above approaches by providing a methodology that is comprehensive, repeatable, extensible, user-oriented, and partly environment independent. This methodology has been applied to several Ada environments at the Software Engineering Institute so that they may be compared objectively according to the same criteria. This paper provides the requirements for an effective environment evaluation methodology, the individual steps of the methodology, and an example of how the methodology has been applied in practice.

[Wein71] Abbreviated Preface: This book has only one major purpose—to trigger the beginning of a new field of study: computer programming as a human activity, or, in short, the psychology of computer programming. All other goals are subservient to that one. What [the author is] trying to accomplish is to have the reader say, upon finishing the book, "Yes, programming is not just a matter of hardware and software. [The author] shall have to look at things in a new way from now on."

As [the author hopes] the text demonstrates with numerous examples, our profession suffers under an enormous burden of myths and half-truths, many of which my students and [the author] have been able to challenge with extremely simple experiments. But our resources are limited and the problem is great. There are, by various estimates, hundreds of thousands of programmers working today. If our experiences are any indication, each of them could be functioning more efficiently, with greater satisfaction, if he and his manager would only learn to look upon the programmer as a human being, rather than as another one of the machines.

[The author thinks] that great strides are possible in the design of our hardware and software too, if we can adopt the psychological viewpoint. [The author] would hope that this book would encourage our designers to add this new dimension to their design philosophy. Not that the few ideas and speculations in this book will give them all the information they need; but hopefully the book will inspire them to go to new sources for information. At the moment, programming-sophisticated as it may be from an engineering or mathematical point of view-is so crude psychologically that even the tiniest insights should help immeasurably. My own experience, and the experience of my students, in teaching, learning, and doing programming with psychological issues in mind, bears out this assertion. [The author hopes] each of [his] readers will try it for himself.

[Wein80] Abstract: Traditional cost/benefit methods in the risk assessment process are predicated on the occurrence of the threat and the resultant loss incurred. Thus, savings can be obtained only if the threat actually occurs. A more appropriate method is the application of the Bayesian decision model in the analysis of the cost-effectiveness of controls to improve system integrity. The applied Bayesian decision model is specifically designed for cost/benefit decisions under conditions of uncertainty and allows for the calculation of the benefit obtained when implementing controls for a threat that does not occur. This method also allows for the calculation of the cost-effectiveness of taking no action, that is, deciding not to implement any control against an identified threat.

[Weis82] Abstract: Error detection and error correction are now considered to be the major cost factors in software development. Much current and recent research has been devoted to finding ways to prevent software errors. The purpose of this paper is to compare error data obtained from two different software-development environments using different software-development methodologies. The data are used to characterize the similarities and differences in the environments and may be used to evaluate the success with which different methodologies meet the claims made for them. Data were obtained by the use of a goal-directed data-collection process which is described briefly. A key feature of the process is that data are collected and validated concurrently with software development. Validation often involves interviewing the programmers supplying the data. The results are data distributions across characterizations, such as effort to correct error, type of error, locality of error. The distributions show that in both environments the principal error source was in the design and implementation of single routines. Requirements misunderstandings, specifications misunderstandings, and interface misunderstanding were all minor sources of errors. Few errors were the result of changes, few errors required more than one attempt at correction, and few error corrections resulted in other errors. Most errors were correctable in a day or less.

[Weis84] Abstract: Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice," is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.

Some properties of slices are presented. In particular, finding *statement-minimal* slices is in general unsolvable, but using data flow analysis is sufficient to find approximate slices. Potential applications include automatic slicing tools for debugging and parallel processing of slices.

[Weis85a] Abbreviated Introduction: Empirically comparing structural test coverage metrics reveals that test sets that satisfy one metric are likely to satisfy another metric as well.

[Weis85b] Abstract: A definition of software reliability is proposed in which reliability is treated as a generalization of the probability of correctness of the software in question. The definition is parameterized by the distribution characterizing the operational environment. It is shown that the definition can be used to provide many natural models of reliability by varying an integer parameter, and that it may be approximated reasonably using well-chosen test sets. It is proved that, under fairly weak conditions, one cannot hope to measure reliability exactly by using finite test sets.

[Weis85c] Abstract: An effective data collection methodology for evaluating software development methodologies was applied to five different software development projects. Results and data from three of the projects are presented. Goals of the data collection include characterizing changes, errors, projects, and programmers, identifying effective error detection and correction techniques, and investigating ripple effects.

The data collected consists of changes (including error corrections) made to the software after code was written and baselined, but before testing began. Data collections and validations were concurrent with software development. Changes reported were verified by interviews with programmers. Analysis of the data showed patterns that were used in satisfying goals of the data collection. Some of the results are summarized in the following.

1. Error corrections aside, the most frequent type of change was an unplanned design modification.
2. The most common type of error was one made in the design or implementation of a single component of the system. Incorrect requirements, misunderstandings of functional specifications, interfaces, support software and hardware, and languages and compilers were generally not significant sources of errors.
3. Despite a significant number of requirements changes imposed on some projects, there was no corresponding increase in frequency of requirements misunderstandings.
4. More than 75% of all changes took a day or less to make.
5. Changes tended to be nonlocalized with respect to individual components but localized with respect to subsystems.
6. Relatively few changes resulted in errors. Relatively few errors required more than one attempt at correction.
7. Most errors were detected by executing the program. The cause of most errors was found by reading code. Support facilities and techniques such as traces, dumps, cross-reference and attribute listings, and program proving were rarely used.

[Weis86] Abstract: A definition of software reliability is proposed in which reliability is treated as a generalization of the probability of correctness of the software in question. The definition is parameterized by the distribution characterizing the operational environment, and by a tolerance function characterizing a notion of degree of correctness. It is shown that the definition can be used to provide many natural models of reliability by varying the tolerance function, and that it may be reasonably approximated using well-chosen test sets. It is proved that, under fairly weak conditions, one cannot hope to measure reliability exactly by using finite test sets.

[Weis88a] Abstract: Representing a concurrent program as a set of simulating, sequential programs provides a solution to the reproducible testing problem as well as a formal foundation for a theory of concurrent program testing. It is shown how this model of concurrent programs is used to extend the methods and theory of testing

sequential programs to concurrent programs.

[Weis88b] Abstract: A definition of software reliability is proposed in which reliability is treated as a generalization of the probability of correctness of the software in question. A tolerance function is introduced as a method of characterizing an acceptable level of correctness. This in turn is used, together with the probability function defining the operational input distribution, as a parameter of the definition of reliability by varying the tolerance function and that it may be reasonably approximated using well-chosen test sets. It is also shown that there is an inherent limitation to the measurement of reliability using finite test set.

[Weyu80c] Abstract: The theory of test data selection proposed by Goodenough and Gerhart is examined. In order to extent and refine this theory, the concepts of a revealing test criterion and a revealing subdomain are proposed. These notions are then used to provide a basis for constructing program tests.

A subset of a program's input domain is revealing if the existence of one incorrectly processed input implies that all of the subset's elements are processed incorrectly. The intent of this notion is to partition the program's domain in such a way that all elements of an equivalence class are either processed correctly or incorrectly. A test set is then formed by choosing one element from each class. This process represents perfect program testing. For a practical testing strategy, the domain is partitioned into subdomains which are revealing for errors considered likely to occur.

Three programs which have previously appeared in the literature are discussed and tested using the notions developed in the paper.

[Weyu82] Abstract: A frequently invoked assumption in program testing is that there is an oracle (i.e., the tester or an external mechanism can accurately decide whether or not the output produced by a program is correct). A program is non-testable if either an oracle does not exist or the tester must expend some extraordinary amount of time to determine whether or not the output is correct. The reasonableness of the oracle assumption is examined and the conclusion is reached that in many cases this is not a realistic assumption. The consequences of assuming the availability of an oracle are examined and alternatives investigated.

[Weyu83] Abstract: Despite the almost universal reliance on testing as the means of locating software errors and its long history of use, few criteria have been proposed for deciding when software has been thoroughly tested. As a basis for the development of usable notions of test data adequacy, an abstract definition is proposed and examined, and approximations to this definition are considered.

[Weyu84a] Abbreviated Introduction: Rapps and Weyuker introduced a family of test data selection criteria based on data flow analysis as used in optimizing compilers. Most test data selection criteria rely solely on the program's control flow characteristics. By incorporating data flow information into the selection procedure, it is possible to focus on associations between physically disjoint portions of the program which are related by the flow of data. In this paper we determine the upper bounds on the amount of test data needed to satisfy each criterion, and thus the relative difficulty of fulfilling each. We call such an upper bound the *complexity* of the criterion.

[Weyu84b] Abstract: A test data adequacy criterion is a set of rules used to determine whether or not sufficient testing has been performed. A general axiomatic theory of test data adequacy is developed, and five previously proposed adequacy criteria are examined to see which of the axioms are satisfied. It is shown that the axioms are consistent, but that only two of the criteria satisfy all of the axioms.

[Weyu88a] Abstract: A family of test data adequacy criteria employing data flow information has been previously proposed, and theoretical complexity analysis performed. This paper describes an empirical study to help determine the actual cost of using these criteria. This should help establish the practical usefulness of these criteria in testing software, and serve as a means of predicting the amount of testing needed for a given program.

[Weyu89] Overview: Rebuttal of first main point (inconsistent evaluation of previously defined criteria) is based on the fact that definitions used in the original paper were taken directly from the literature. Rebuttal of second main point (lack of precision and formality) revolves around assumption of a software engineer's understanding of an adequacy criterion.

[Whit78b] Abstract: This paper presents a testing strategy designed to detect errors in the control flow of a computer program, and the conditions under which this strategy is reliable are given and characterized. The control flow statements in a computer program partition the input space into a set of mutually exclusive *domains*, each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. The testing strategy generates test points to examine the boundaries of a domain to detect whether a domain error has occurred, as either one or more of these boundaries will have shifted or else the corresponding predicate relational operator has changed. If test points can be chosen within ϵ of each boundary, under the appropriate assumptions, the strategy is shown to be reliable in detecting domain errors of magnitude greater than ϵ. Moreover, the number of test points required to test each domain grows only linearly with both the dimensionality of the input space and the number of predicates along the path being tested.

[Whit80] Abstract: Many current software development methodologies require designers to select design options based on a comparative evaluation of the merits of various design alternatives. However, techniques for the evaluation and continuous monitoring of software quality lack sufficient development or generality to have achieved widespread acceptance. While there is much interesting work addressing the assessment of software code quality, few measurement techniques are applicable to software designs. In this paper, software design quality is emphasized. A general formalism for expressing software designs is presented, and two metrics of design quality, as functions of control flow and data flow complexity, are proposed.

[Whit86] Abstract: An automated testing approach called the *Domain Testing Strategy* has been developed to primarily detect errors in software control flow, though it may also detect errors in computation. Detection of control flow errors is accomplished by determining that the domain boundary is correct within an acceptable error bound. An analysis of this strategy has appeared in the literature which identifies those conditions under which this error bound is not acceptable, and methods were proposed to select test points which achieved a reduced error bound. It is the objective of this paper to provide an alternative measure of error bound which is more easily calculated, and an heuristic method for selecting test points. Although the use of this measure and test selection method will not result in the same level of reduction in error bounds in Domain Testing as those previously proposed, it is argued that the reduction in effort can justify this alternative approach.

[Whit88a] Abstract: One of the serious limitations of domain testing is the potentially infinite number of domains to be examined in the presence of iteration loops in the computer program. The purpose of this paper is to show that only a small number of domains need to be examined, and that one can concentrate on testing certain borders of those domains. It is first shown that for *definite loops*, where the number of iterations is known upon entry, iteration loops can be represented by a primitive recursive schema. This involves the identification of *simple loop patterns*, and it is proven that these simple loop patterns can be used as basic building blocks to form arbitrarily complex loop patterns. It is further shown that domain testing can be adapted to test these simple loop patterns, which precludes the necessity of having to test any of the complex loop patterns. A bound is obtained on the number of loop patterns that have to be tested and worst cases identified for the corresponding control flow graphs: for loop patterns from the perspective of the exit node, for loop patterns required to test all predicate nodes, and for loop patterns required to test all final predicate nodes. The paper concludes with some recommendations for those simple loop patterns which should be selected first for testing in order to provide greatest information about errors in the program, and identifies some problems for future research.

[Whit88b] Position Statements Included:

- Gensheimer, E.L. "Technology Transfer in the Product Verification/Quality Areas."

- Good, D.I. "Transferring Testing and Verification Technology to Industry."
- Hennell, M.A. "Technology Transfer."
- Miller, E. "Testing and Verification Problems in Industry: Technology Transfer."
- Sneed, H.M. "State Coverage of Embedded Real-time Programs."

[Wien84] Abstract: In this paper, a formal model of the software manloading pattern, the Rayleigh model, is described and then applied to four Bankers Trust Company (BTCo.) new development projects processing complete life cycle manloading data (maintenance phase included). To fit the Rayleigh curve to a project's manloading scores, (nonlinear) regression was used to obtain least squares estimates of the Rayleigh parameters, which, in turn, were used to generate the Rayleigh manloading curve. For all four projects, deviation from the Rayleigh curve was small and constant throughout the software development phases (i.e., preliminary design through implementation); however, the Rayleigh curve consistently deviated from the actual manloading during system maintenance, underestimating the amount of maneffort expended. Restricting maintenance maneffort to manpower expended on repair of system faults ("corrective" maintenance) resulted in a single Rayleigh curve that could be applied over the entire BTCo. life cycle. Furthermore, this corrective portion of the maintenance effort could be accurately forecasted from the Rayleigh curve fit to software development. Implications of these findings for software management are discussed.

[Wild88] Overview: Current logic programming systems, as typified by PROLOG, contain limitations which restrict their usefulness during the specification, design and testing of software. A major limitation is the inability to perform analysis in the presence of incomplete information. Three sources of incompleteness are discussed here. First, the analysis is incomplete because the system is only partially finished. Second, in order to provide overall guidance, the analysis is first performed at an abstract level. The abstraction can be done selectively in order to focus the analysis. Third, some forms of incompleteness can only be resolved at run time by examining the properties of objects which are determined dynamically. It is the program itself which resolves the last form of incompleteness.

In logic programming, the program is expressed in terms of predicates relating the input and output and execution proceeds by constructing a proof of these input/output relationships. **Generic Constraint Logic Programming** is a form of logic programming developed to address incompleteness in analysis.

[Wile88] Abstract: Defining, creating, and manipulating *persistent typed objects* will be central activities in future software environments. PGRAPHITE is a working prototype through which we are exploring the requirements for the persistent object capability of an object management system in the Arcadia software environment.

PGRAPHITE represents both a set of abstractions that define a model for dealing with persistent objects in an environment and a set of implementation strategies for realizing that model. PGRAPHITE currently provides a type definition mechanism for one important class of types, namely directed graphs, and the automatic generation of Ada implementations for the defined types, including their persistence capabilities.

We present PGRAPHITE, describe and motivate its model of persistence, outline the implementation strategies that it embodies, and discuss some of our experience with the current version of the system.

[Will79] Abstract: In languages such as Pascal, the programmer can arrange to have the compiler check such things as the range of the value of a variable only by defining a new type or sub-type. [The author has] investigated how more powerful checking facilities might be provided if they were divorced from the type machinery, and also if the necessary language constructs were designed independent of what any particular compiler would check at compile-time.

[Wing89] Abstract: Toward the overall goal of putting formal specifications to practical use in the design of large systems, we explore the combination of two specification methods: using temporal logic to specify concurrency properties and using an existing specification language, Ina Jo, to specify functional behavior of nondeterministic systems. In this paper, we give both informal and formal descriptions of both current Ina Jo and Ina Jo enhanced with temporal logic. We include details of a simple example to demonstrate the use of the proof system and

details of an extended example to demonstrate the expressiveness of the enhanced language. We discuss at length our language design goals, decisions, and their implications. The Appendix contains a list of axioms, rules of inference, derived rules, and theorem schemata for the enhanced formal system.

[Wirs83] **Summary:** Hierarchical abstract data types are algebraic specifications of computation structures where certain sorts, function symbols, and axioms are designated as being primitive. On hierarchical abstract data types additional structure is imposed. An algebraic specification is thus decomposed into several well-separated levels, such that both the understanding and the independent implementation of the levels is supported. This paper provides both model-theoretic and deduction-oriented conditions guaranteeing the soundness of a hierarchical specification. Furthermore necessary and sufficient conditions for the existence of initial and terminal models are investigated, and their close connection to the soundness of a hierarchy is demonstrated. In order to provide freedom and flexibility for specifications a wide class of axioms - namely universal-existential formulas - are admitted.

[Wolf85a] **Abstract:** The Ada programming language is intended for the implementation of large and complex software systems. Such systems often exceed a half-million lines of code; if their developers adhere to the software engineering maxim that no module should contain more than 50 lines of code, then the number of modules in such systems will exceed 10,000! As DeRemer and Kron point out, dealing with a "large collection of modules to form a 'system' is an essentially distinct and different intellectual activity from that of constructing the individual modules." Thus, developers and maintainers of large Ada systems will require tools beyond the syntax-directed editors, compilers, debuggers and so on needed for "programming-in-the-small." They will need extensive support for describing, analyzing, organizing, and managing the modules in a system–that is, an environment for "programming-in-the-large."

[Wolf86c] **Abstract:** Despite the importance of describing and analyzing the relationships among a software system's components, most languages and development environments do not provide suitable support for these activities. While Ada and the various existing Ada environments offer some assistance, the capabilities they offer are inadequate for use in truly large and complex software development projects. To address these shortcomings, we are developing the AdaPIC toolset, which we envision as an important component of an Ada software development environment. The AdaPIC toolset is one particular instantiation, specifically adapted for use with Ada, of the more general collection of language features and analysis capabilities that constitute the PIC approach to describing and analyzing relationships among software system components. This toolset is being tailored to support an incremental approach to the interface control aspects of the software development process. Following a discussion of the interface control and incremental development concepts, this paper describes the AdaPIC toolset, concentrating on its analysis tools and support for incremental development and demonstrating how it contributes to the technology for developing large Ada software systems.

[Wolv74] **Abstract:** The work of software cost forecasting falls into two parts. First we make what we call structural forecasts, and then we calculate the absolute dollar-volume forecasts. Structural forecasts describe the technology and function of a software project, but not its size. We allocate resources (costs) over the project's life cycle from the structural forecasts. Judgement, technical knowledge, and econometric research should combine in making the structural forecasts. A methodology based on a 25 x 7 structural forecast matrix that has been used by TRW with good results over the past few years is presented in this paper. With the structural forecasts in hand, we go on to calculate the absolute dollar-volume forecasts. The general logic followed in "absolute" cost estimating can be used on either a mental process or an explicit algorithm. A cost estimating algorithm is presented and five traditional methods of software cost forecasting are described: top-down estimating, similarities and difference estimating, ratio estimating, standards estimating, and bottom-up estimating. All forecasting methods suffer from the need for a valid cost data base for many estimating situations. Software information elements that experience has shown to be useful in establishing such a data base are given in the body of the paper. Major pricing pitfalls are identified. Two case studies are presented that illustrate the software cost forecasting methodology and historical results. Topics for further work and study are suggested.

[Wood79a] Abstract: This paper discusses the need for measures of complexity and of unstructuredness of programs. A simple language independent concept is put forward as a measure of control flow complexity in program text and is then developed for use as a measure of unstructuredness. The proposed metric is compared with other metrics, the most notable of which is the cyclomatic complexity measure. Some experience with automatic tools for obtaining these metrics is reported.

[Wood79b] Abstract: The effect of a variation in problem complexity and how the variation relates to programming complexity is predicted and measured. An experiment was conducted in which eighteen graduate students programmed two variations of the same small algorithm where the problem complexity varied by 25 percent. Eight measurable program characteristics are compared with predicted values obtained using only two known parameters. The agreement between observed and predicted values is very good. Both predicted and observed measurements indicate that the 25 percent increase in problem complexity results in a 100 percent increase in programming complexity.

[Wood80b] Abstract: There are a number of practical difficulties in performing a path testing strategy for computer programs. One problem is in deciding which paths, out of a possible infinity, to use as test cases. A hierarchy of structural test metrics is suggested to direct the choice and to monitor the coverage of test paths. Another problem is that many of the chosen paths may be unfeasible in the sense that no test data can ever execute them. Experience with the use of "allegations" to circumvent this problem and prevent the static generation of many unfeasible paths is reported.

[Wood81a] Abstract: As the cost of programming becomes a major component of the cost of computer systems, it becomes imperative that program development and maintenance be better managed. One measurement a manager could use is programming complexity. Such a measure can be very useful if the manager is confident that the higher the complexity measure is for a programming project, the more effort it takes to complete the project and perhaps to maintain it. Until recently most measures of complexity were based only on intuition and experience. In the past 3 years two objective metrics have been introduced, McCabe's cyclomatic number $v(G)$ and Halstead's effort measure E. This paper reports an empirical study designed to compare these two metrics with a classic size measure, lines of code. A fourth metric based on a model of programming is introduced and show to be better than the previously known metrics for some experimental data.

[Wood81b] Abstract: An experiment was conducted to investigate how different types of modularization and comments are related to programmers' ability to understand programs. Forty-eight experienced programmers were given eight different versions of the same program and asked to answer a twenty question quiz used to measure comprehension. These eight different versions were the result of the program being constructed with four types of modularization (monolithic, functional, super, and abstract data type), each with and without comments. Those subjects whose programs contained comments were able to answer more questions than those without comments. Also, those subjects who were given the abstract data type version of the program were able to do significantly better than those with any other type of modularization.

[Wood81c] Overview: In order to improve upon the complexity measurement results obtained using the LOC, McCabe's cyclomatic number, and Halstead's software science effort measure, the authors have developed another model for programming complexity. This measure includes consideration of control, data, and implicit module interconnections.

[Wood88] Abstract: Despite the intrinsic appeal of the mutation approach to testing, its disadvantage in being computationally expensive has hampered its widespread acceptance. When weak mutation was introduced as a less expensive and less stringent form of mutation testing, the original technique was renamed strong mutation. This paper argues that strong mutation testing and weak mutation testing are in fact extreme ends of a spectrum. The term *firm mutation* is introduced here to represent the middle ground in this spectrum. This paper also argues, by means of a number of small examples, that there is a potential problem concerning the criterion for

deciding whether a mutant is 'dead' or 'live'. A variety of solutions are suggested. Finally, practical considerations for a firm mutation testing system, with greater user control over the nature of result comparison, are discussed. Such a system is currently under development as part of an interpretive development environment.

[Wu87c] Abstract: Coverage metrics have traditionally been used to evaluate the effectiveness of procedures for testing software systems. In practice, however, the metrics are heavily influenced by the characteristics of traditional programming languages such as Fortran and Pascal. Languages such as Ada differ from traditional languages to such an extent that it is necessary to develop new metrics.

This paper proposes a number of coverage measures for Ada features such as packages, generic units, and tasks, and discusses their interpretations in relation to the traditional coverage metrics. It also proposes a mechanism for collecting these coverage measures. In addition, it suggests that coverage metrics may also be interpreted as indicators of dynamic system performance.

[Wu88] Abstract: Program mutation is a suitable technique for investigating software reliability and quality control since it is able to detect many potential errors. However it is necessary to improve the technique for industrial practice. A new method of program mutation is presented here which increases the feasibility, effectiveness and efficiency of searching for those errors which have escaped from the activities of Afterers and competent programmers. It is based on syntax direction and it is aided by the language semantics. This means that the scope of a program mutation (i.e. the mutation rules of the method), and its corresponding mutants, are rigorously directed by a syntax and related semantics as defined by the tester. A paradigm for the mutation syntax and semantics when limited to boolean expressions and the corresponding test coverage metrics in terms of this method are given in the paper.

[Wulf76] Abstract: The programming language Alphard is designed to provide support for both the methodologies of "well-structured" programming and the techniques of formal program verification. Language constructs allow a programmer to isolate an *abstraction*, specifying its behavior publicly while localizing knowledge about its implementation. The verification of such an abstraction consists of showing that its implementation behaves in accordance with its public specifications; the abstraction can then be used with confidence in constructing other programs, and the verification of that use employs only the public specifications.

This paper introduces Alphard by developing and verifying a data structure definition and a program that uses it. It shows how each language construct contributes to the development of the abstraction and discusses the way the language design and the verification methodology were tailored to each other. It serves not only as an introduction to Alphard, but also as an example of the symbiosis between verification and methodology in language design. The strategy of program structuring, illustrated for Alphard, is also applicable to most of the "data abstraction" mechanisms now appearing.

[Yama83] Abstract: A stochastic model for a software error detection process in which the growth curve of the number of detected software errors for the observed data is S-shaped is investigated. The software error detection model is a nonhomogeneous poisson process where the mean-value function has an S-shaped growth curve. The model is applied to actual software error data, and the maximum-likelihood estimates (MLES) of the unknown parameters and the related quantitative indices are obtained. Statistical inference on the unknown parameters is discussed. Comparison with other models indicates that the model presented fits the observed data better than other models.

[Yau78] Abstract: Maintenance of large-scale software systems is a complex and expensive process. Large-scale software systems often possess both a set of functional and performance requirements. Thus, it is important for maintenance personnel to consider the ramifications of a proposed program modification from both a functional and a performance perspective. In this paper the ripple effect which results as a consequence of program modification will be analyzed. A technique is developed to analyze this ripple effect from both functional and performance perspectives. A figure-of-merit is then proposed to estimate the complexity of program modification. This figure can be used as a basis upon which various modifications can be evaluated.

[Yau79] Abstract: Software maintenance has been the dominant factor contributing to the high cost of software. In this paper, the software maintenance process and the important software quality attributes that affect the maintenance effort are discussed. Among these quality attributes, the stability of a program, which indicates the resistance to the potential ripple effect that the program would have when it is modified, is an important one. Measures for estimating the stability of a program and the modules of which the program is composed are presented, and an algorithm for computing these stability measures is given. Application of these measures during the maintenance phase is discussed along with an example. Further research efforts involving validation of the stability measures, application of these measures during the design phase, and restructuring based on these measures are also discussed.

[Yau80] Abstract: A control flow checking scheme capable of detecting control flow errors of programs resulting from software coding errors, hardware malfunctions, or memory mutilation during the execution of the program is presented. In this approach, the program is partitioned into loop-free intervals and a database containing the path information in each of the loop-free intervals is derived from the detailed design. The path in each loop-free interval actually traversed at run time is recorded and then checked against the information provided in the database, and any discrepancy indicates an error. This approach is general, but can detect all uncompensated illegal branches. Any uncompensated error that occurs during the execution of a loop-free interval and manifests itself as a wrong branch within the loop-free interval is also detectable. The approach can also be used to check the control flow in the testing phase of program development. The capabilities, limitations, implementation, and the overhead of using this approach are discussed.

[Yeh77] Abbreviated Preface: Software validation involves analyzing software to determine the extent to which it performs the logical functions intended by its creator. Techniques in software validation can be classified broadly into two categories: testing and verification. In this volume, the first five chapters are concerned with testing techniques and tools, and the remaining chapters are concerned with verification techniques.

In the first chapter, Henderson argues that testing should be a constructive activity and should be planned during the developmental phase of a program. In the second chapter, Huang gives a tutorial discussion of a specific technique for testing. In Chapter 3, Goodenough and Gerhart make a first attempt to develop a theory for software testing. In Chapter 4, Stucki presents a specific set of software tools as an aid for software testing. In Chapter 5, Ramamoorthy and Ho present a comprehensive survey of automated tools for testing large software. Operational experiences of several major systems and their limitations are also discussed in this chapter.

There are two ways of approaching program verification. The static approach considers a program and its specifications to be given. Mathematical proofs are developed to demonstrate that the logical behavior of a program is as specified, viewing this logical behavior as completely characterized by a set of formal assertions. The constructive approach lays stress on the correct development of a program. The remaining five chapters survey various techniques in the static approach.

In Chapter 6, London discussed the role of software verification and gives a tutorial introduction to the "inductive assertion" proof technique. In Chapter 7, Robinson and Levitt extend the inductive assertion technique to verify hierarchically structured programs. In Chapter 8, Morris and Wegbreit present another proof technique called subgoal induction. In Chapter 9, Yeh gives yet another proof technique which differs from inductive assertion and subgoal induction in [providing a] proof of total correctness. In Chapter 10, Katz and Manna survey existing techniques for proving that programs terminate.

Finally, Ann Marmor-Squires' selected annotated bibliography provides an easy guide to literature in program validation.

[Yeh79] Introduction: Maury Halstead had a dream! By treating computer programs as neither art forms nor as examples of mathematical logic, but instead as basic material which can be investigated with the classical methods of experimental and theoretical natural science, Maury had dreamed of and worked hard toward a unified and coherent new field he called Software Science, in which attributes of a computer program, such as implementation efforts, clarity, structure, error rates, language levels, etc., can be derived from its basic metrics

through quantitative hypotheses.

The special collection of papers on Software Science not only contains some of Maury's final contributions to the field he started, but its diversity and sophistication is an assurance that Maury's dream will be carried on.

[Yin78] Abstract: It has been recognized that success in producing designs that realize reliable software, even using Structured Design, is intimately dependent on the experience level of the designer. The gap in this methodology is the absence of easily applied quantitative measures of quality that ease the dependence of reliable systems on the rare availability of expert designers.

Several metrics have been devised which, when applied to design structure charts, can pinpoint sections of a design that may cause problems during coding, debugging, integration, and modification. These metrics can help provide an independent, unbiased evaluation of design quality. These metrics have been validated against program error data of two recently completed software projects at Hughes, The results indicate that the metrics can provide a predictive measure of program errors experienced during program development.

Guidelines for interpreting the design metric values are summarized and a brief description of an interactive structure chart graphics system to simplify metric value calculation is presented.

[Yin80] Abstract: A software design and testability analysis system has been developed at Hughes Aircraft Company to measure the software quality in terms of reliability, maintainability, and testability. Based on software design structure charts, the system indicates the error-prone and difficult-to-test areas of software design by quantitatively measuring the program complexity and testability. Also the system produces several testing aids which facilitate integration. The results have been successfully validated against several software projects at Hughes-Fullerton.

The system is configured for the AMDAHL/470 and is accessible via a HP2648A graphics terminal. It allows the designers to interactively create and edit the design, and automatically produces structure charts for documentation, metrics for quality measurements, and test plans for integration.

[Youn86a] Abstract: Static concurrency analysis detects anomalous synchronization patterns in concurrent programs, but may also report spurious errors involving unfeasible execution paths. Integrated applications of static concurrency analysis and symbolic execution sharpens the results of the former without incurring the full costs of the latter applied in isolation. Concurrency analysis acts as a path selection mechanism for symbolic execution, while symbolic execution acts as a pruning mechanism for concurrency analysis. Methods for combining the techniques follow naturally from explicit characterization and comparison of the state spaces explored by each, suggesting a general approach for integrating state-based program analysis techniques in a software development environment.

[Youn88a] Abstract: Analyses based on state-space models of execution must omit some details of execution, in order to fold the infinite space of possible program executions into a sufficiently small space for analysis. These simplifications are generally justified by a claim that the resulting analysis is *conservative* with respect to a certain class of faults, i.e., that the simplification will not cause any faults to be overlooked in the analysis. We formalize a notion of *error-preserving* abstractions which captures this claim, give sufficient conditions for verifying this property in practical cases, and discuss the role of error-preserving abstractions in combining fault detection techniques.

[Youn89a] Abbreviated Preface: The purpose of IDA Paper P-2132, *SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation With Recommended R&D Tasks*, is to identify the technology required for effective and efficient testing and evaluation of Strategic Defense System (SDS) software. This document was prepared for the Strategic Defense Initiative Organization (SDIO), and provides an overview of current testing and evaluation technology, a mapping of available technology against SDS needs, and recommendations to close critical gaps in technology.

[Youn89b] Abbreviated Preface: The purpose of IDA Memorandum M-496, *Bibliography of Testing and Evaluation Reference Material*, is to present the reference material acquired in the course of developing IDA Paper P-2132, *SDS Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation With Recommended R&D Tasks*. This document was prepared for the Strategic Defense Initiative Organization (SDIO).

[Youn89c] Abstract: The conventional classification of software fault detection techniques by their operational characteristics (static vs. dynamic analysis) is inadequate as a basis for identifying useful relationships between techniques. A more useful distinction is between which *sample* the space of possible executions, and techniques which *fold* the space. The new distinction provides better insight into the ways different techniques can interact, and is a necessary basis for considering hybrid fault detection techniques.

[Your76] Abbreviated Preface: In the past few years, the programming industry has been revolutionized by a number of new philosophies and techniques. One of the most popular of these techniques, *structured programming*, has led to order-of-magnitude improvements in programmer productivity, program reliability, and program maintenance costs.

More recently, though, there has been a recognition that perfectly structured GOTO-less code is essentially worthless if the basic *design* of the program or system is unsound.

Our concern is with the overall architecture of programs and systems. How should a large system be broken into modules? *Which* modules? Which ones should be subordinate to which? How do we know when we have a "good" module and, more important, how do we know when we have a "bad" module? What information should be passed between modules? Should a module have the opportunity to access data other than that which it needs to know in order to accomplish its task? How should the modules be "packaged" together into efficient executable units in a typical computer?

Naturally, the answers to these questions are influenced by the specific details of hardware, operating system, and programming language – as well as the designer's interest in such things as efficiency, simplicity, maintainability, and reliability. Nevertheless, we argue that questions such as the ones posed above are of a higher level than the detailed coding questions of "Should I use a GOTO here?" or "How can I write a nested IF statement to accomplish this editing logic?"

[Yu88a] Abstract: This paper presents the data and capabilities provided by the Software Metrics Data Collection (SMDC) system. SMDC is an APL-based system that runs on the UNIX 4.3BSD system at Purdue University. The largest software product in SMDC has more than 1,000,000 lines of code. SMDC also provides a number of statistical functions and plotting routines that can be used for detailed analysis of existing data. The data and tools in SMDC are available for use by non-Purdue researchers with some limitations.

[Yu88b] Abstract: This paper presents the results of analyzing several defect models using data collected from two large commercial projects. Traditional models typically use either program metrics (i.e., measurements from software products) or testing time or combinations of these as independent variables. The limitations of such models have been well-documented. For example, program metrics are difficult to compute for those products that consist of code modified from previous versions. Another example is that testing time is not available at the beginning of the testing phase. The models considered in this paper all use the number of defects detected in the earlier phases of the development process as the independent variable. This number can be used to predict the number of defects to detected later, even in modified software products. We have found a very strong correlation between the number of earlier defects and that of later ones. Using this relationship, we constructed a mathematical model which may be used to estimate the number of defects remaining in software. This defect model may also be used to guide software developers in evaluating the effectiveness of the software development and testing processes.

[Zafi80] Abstract: The production of error-free protocols or complex process interactions is essential to reliable communications. This paper presents techniques for both the detection of errors in protocols and for prevention

of errors in their design. The methods have been used successfully to detect and correct errors in existing protocols. A technique based on a reachability analysis is described which detects errors in a design. This "perturbation technique" has been implemented and has successfully detected inconsistencies or errors in existing protocol designs including both X.21 and X.25. The types of errors handled are state deadlocks, unspecified receptions, nonexecutable interactions, and state ambiguities. These errors are discussed and their effects considered. An interactive design technique is then described that prevents design errors. The technique is based on a set of production rules which guarantee that complete reception capability is provided in the interacting processes. These rules have been implemented in the form of a tracking algorithm that prevents a designer from creating unspecified receptions and nonexecutable interactions and monitors for the presence of state deadlocks and ambiguities.

[Zeil81b] Abstract: Many testing methods require the selection of a set of paths over which testing is to be conducted. This paper presents an analysis of the effectiveness of individual paths for testing predicates in linearly domained programs. A measure is derived for the marginal advantage of testing another path after several paths have already been tested. This measure is used to show that any predicate in such programs may be sufficiently tested using at most m+n+1 paths, where m is the number of input values and n is the number of program variables.

[Zeil83a] Abstract: Many testing methods require the selection of a set of paths on which tests are to be conducted. Errors in arithmetic expressions within program statements can be represented as perturbing functions added to the correct expression. It is then possible to derive the set of errors in a chosen functional class which cannot possibly be detected using a given test path. For example, test paths which pass through an assignment statement *"X:= f(Y)"* are incapable of revealing if the expression *"X - f(Y)"* has been added to later statements. In general, there are an infinite number of such undetectable error perturbations for any test path. However, when the chosen functional class of error expressions is a vector space, a finite characterization of all undetectable expressions can be found for one test path, or for combined testing along several paths. An analysis of the undetected perturbations for sequential programs operating on integers and real numbers is presented which permits the detection of multinomial error terms. The reduction of the space of (potential) undetected errors is proposed as a criterion for test path selection.

[Zeil84] Abstract: The use of algebraic techniques in defining a neighborhood of functions is particularly suited to testing for computation errors. Two possible approaches are Howden's algebraic testing method and perturbation testing, which in this paper is generalized to permit analysis of individual test points rather than entire paths. These approaches are shown to be mathematically equivalent when applied to a program's black-box output. Perturbation testing, however, offers more flexibility in the choice of potential errors to be investigated. A significant alternative offered by perturbation testing is the ability to work in the static domain, choosing test data to eliminate possible error terms in specific assignment and output statements.

[Zeil86] Abstract: This paper introduces a new testing strategy, EQUATE testing. EQUATE represents an attempt to merge the strengths of perturbation testing and mutation testing in order to provide a testing strategy that offers support for data and functional abstraction, that detects a wide variety of simple faults, and that also provides good coverage of combinations of those simple faults. EQUATE selects a number of test locations throughout the program and chooses a set of expressions derived from the abstract syntax tree of the modules being tested. Test data is required that distinguishes each pair of these expressions from one another at every test location.

[Zeil87] Abstract: The philosophy of composing new software tools from previously created tool fragments can facilitate the development software systems. An examination is made of the extension of this philosophy to the design of program interpreters, demonstrating how the separation of interpretation into a core algorithm, value-kind definitions, and computation model allows the capture of conventional execution models, symbolic execution models, dynamic dataflow tracking, and other useful forms of program interpretation. An interpretation

system based on this separation, called ARIES, is currently under development.

[Zeil88a] Abstract: A given path selection criterion is more selective than another such criterion with respect to some testing goal if it never requires more, and sometimes requires fewer, test paths to achieve that goal. This paper presents canonical forms of control-flow and data-flow path selection criteria and demonstrates that, for some simple testing goals, the data-flow criteria as a general class are more selective than the control-flow criteria. It is shown, however, that this result does not hold for general testing goals, a limitation that appears to stem directly from the practice of defining data-flow criteria upon the computation history contributing to a single result.

[Zeil88b] Abstract: Most testing methods do not fare well with software whose modules contain data and operations at widely varying levels of abstraction. With the evolution of new design techniques and new languages that encourage the use of more abstract data types, it is becoming increasingly important that testing methods begin to deal with abstraction in a reasonable and consistent manner. The EQUATE testing strategy offers strong support for consistent manner. The EQUATE selects a number of test locations throughout the program and chooses a set of expressions derived from the abstract syntax tree of the module being tested. Test data is required that distinguishes these expressions from one another at every test location. The time complexity of EQUATE is at worst $O(L_p^2)$ and the space complexity $O(L_p^3)$ where L_p is the length of the program under test.

[Zeil88c] Abstract: Despite the important role played by the notion of abstraction in modern methods of software design and implementation, relatively little consideration has been paid to the interaction between abstraction and testing criteria, especially for automatable criteria. A survey of relevant syntactic, semantic, and methodological problems is presented, and a brief overview is presented of research by the author aimed at developing testing criteria free of those problems.

[Zeil89] Abstract: Perturbation testing is an approach to software testing which focuses on faults within arithmetic expressions appearing throughout a program. In this paper perturbation testing is expanded to permit analysis of individual test points rather than entire paths, and to concentrate on domain errors. Faults are modeled as perturbing functions drawn from a vector space of potential faults and added to the correct form of an arithmetic expression. Sensitivity measures are derived which limit the possible size of those faults that would go undetected after the execution of a given test set. These measures open up an interesting new view of testing, in which attempts are made to reduce the volume of possible faults which, were they present in the program being tested, would have escaped detection on all tests performed so far. The combination of these new measures with standard optimization techniques yields a new test data generation method, called *arithmetic fault detection*.

[Zelk78] Abstract: Software engineering refers to the process of creating software systems. It applies loosely to techniques which reduce high software cost and complexity while increasing reliability and modifiability. This paper outlines the procedures used in the development of computer software, emphasizing large-scale software development, and pinpointing areas where problems exist and solutions have been proposed. Solutions from both the management and the programmer points of view are then given for many of these problem areas.

[Zoln81] Abstract: Program complexity is a topic often discussed in the literature. Research is ongoing in verifying existing complexity measures. There is also a continuing effort to produce and validate new approaches to a complexity measure which incorporate ideas from a variety of areas.

Too often, however, approaches to complexity measurement center on a particular aspect of a program, e.g., structures, without incorporating other relevant program characteristics. The question to be answered, then, is, What aspects of a program contribute to its complexity?

This paper presents a first step in answering this question. Preliminary results are presented from a Delphi Survey on program complexity. The survey was sent to a cross-section of programmers, managers and software experts. Respondents rated a large number of characteristics as to their effect on program complexity. The paper

summarizes the results and includes preliminary analyses.

[Zweb79] Abstract: During the past few years, several investigators have noted definite patterns in the distribution of operators in computer programs. Their proposed models have provided explanations for other observed software phenomena and have suggested possible relationships between programming languages and natural languages. However, these models contain notable deficiencies.

This study concentrates on a set of production programs written in PL/I. Using some basic relationships from software science, and a previously published algorithm generation technique, a model for computing operator frequencies is constructed which is based only on the number of distinct operators in the program and the total number of operator occurrences. The model provides a considerable statistical improvement over existing models for the PL/I programs studied.

[Zweb89] Abstract: Weyuker has recently proposed a set of properties which should be satisfied by any reasonable criterion used to claim that a computer program has been adequately tested. She called these properties "axioms." She also evaluated several well-known testing strategies with respect to these properties, and concluded that some of the commonly used strategies failed to satisfy several of the properties.

We question both the fundamental nature of the properties and the precision with which they are presented, and illustrate how a number of ideas in Weyuker's paper can be simplified and clarified through greater precision and a more consistent set of definitions. We also reanalyze the testing strategies after accounting for these inconsistencies. The strategies tend to fare much better as a result of this reanalysis.

[vanH68] Abstract: The designer of a computing system should adopt explicit criteria for accepting or rejecting proposed system features. Three possible criteria of this kind are input recordability, input specifiability, and asynchronous reproducibility of output. These criteria imply that a user can, if he desires, either know or control all the influences affecting the content and extent of his computer's output. To define the scope of the criteria, the notion of an abstract machine of a programming language and the notion of a virtual computer are explained. Examples of applications of the criteria concern the reading of a time-of-day clock, the synchronization of parallel processes, protection in multiprogrammed systems, and the assignment of capability indexes.

[vonH85] Ada packages are the basic building blocks of Ada programs. The separation in Ada into package visible part and body is intended to support a programming style that employs modularization, encapsulation and information hiding. Unfortunately, the visible part provides only the syntactic interface to the package; it does not convey any information about the meaning of, e.g., visible subprograms. Instead, when the user of an Ada package wants to understand what services it provides he needs to study the package body. Thus, the purpose of the separation into visible part and body is somewhat subverted if the body is the only place where semantic information can be found.

Distribution List for IDA Memorandum Report M-496

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Lt Col Chuck W. Lillie SDIO/ENA Room 1E149, The Pentagon Washington, D.C. 20301-7100	2
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Ms. Christine Youngblut 17021 Sioux Ln. Gaithersburg, MD 20878	2
CSED Review Panel	
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, Illinois 61801	1
Dr. Barry W. Boehm DARPA 1400 Wilson Blvd. Arlington, VA 22209-2308	1
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1
Dr. John M. Palms, President Georgia State University University Plaza Atlanta, GA 30303	1
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1
Mr. Keith Uncapher University of Southern California Olin Hall 330A University Park Los Angeles, CA 90089-1454	1
IDA	
General W.Y. Smith, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Dr. Cy R. Adoin, CSED	1
Mr. Bill R. Brykczynski, CSED	5
Ms. Anne Douville, CSED	1
Dr. Dennis W. Fife, CSED	1
Dr. John F. Kramer, CSED	1
Dr. Cathy Jo Linn, CSED	1
Mr. Terry Mayfield, CSED	1
Ms. Katydean Price, CSED	2
Dr. Richard Wexelblat, CSED	1

NAME AND ADDRESS	NUMBER OF COPIES
IDA Control & Distribution Vault	3

END