Computer Science
Technical Report

Multiversion Software Reliability Through Fault-Avoidance and Fault-Tolerance
Report #1 (3/1/89-8/31/89) on NAG-1-983

by

Mladen A. Vouk and David F. McAllister

(NASA-CR-186502) MULTI-VERSION SOFTWARE RELIABILITY THROUGH FAULT-AVOIDANCE AND FAULT-TOLERANCE Semiannual Technical Report, 1 Mar. - 31 Aug. 1989 (North Carolina State Univ.) 42 p

N90-25584

Unclas
CSCL 09B G3/61 0273074

North Carolina State University

Box 8206
Raleigh, NC 27695

Semi-Annual Technical Report Submitted to the

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Langley Research Center, Hampton, Va.

for research entitled

MULTI-VERSION SOFTWARE RELIABILITY THROUGH FAULT-AVOIDANCE AND FAULT-TOLERANCE

(NAG-1-983)

from

Mladen A. Vouk, Co-Principal Investigator, Assistant Professor
David F. McAllister, Co-Principal Investigator, Professor

Department of Computer Science
North Carolina State University
Raleigh, N.C. 27695-8206
(919) 737-2858

Report Period
Beginning Date: March 1, 1989.
Ending Date: August 31, 1989.

Raleigh, September 1, 1989

Table of Contents

Project Progress Summary

In this project we have proposed to investigate a number of experimental and theoretical issues associated with the practical use of multi-version software to provide run-time tolerance to software faults. In the period reported here we have worked on the following:

- We have finished developing and evaluating a specialized tool for measuring testing coverage for a variety of metrics.
- We have started using the tool to collect information on the relationships between software faults and coverage provided by the testing process as measured by different metrics (including data flow metrics). We have found considerable correlation between coverage provided by some higher metrics and the elimination of faults in the code.
- We have continued studying back-to-back testing as an efficient mechanism for removal of un-correlated faults, and common-cause faults of variable span.
- We continued studying software reliability estimation methods based on non-random sampling, and the relationship between software reliability and code coverage provided through testing.
- We continued investigating existing, and worked on formulation of new fault-tolerance models. In particular, we have finished simulation studies of the Acceptance Voting and Multi-stage Voting algorithms, and found that these two schemes for software fault-tolerance are superior in many respects to some commonly used schemes. Particularly encouraging are the safety properties of the Acceptance testing scheme.

This report describes the results obtained in the period March 1, 1989 to August 31, 1989.

1. General Project Description

Software reliability is very important in critical software application areas. For example, space based systems, avionics systems, critical nuclear power plant systems, and life-critical medical systems are all expected to operate reliably even under extremely severe conditions. However, practice shows that critical systems are not immune to software related failures [e.g. Neu85].

Currently there are two basic ways of showing that code is 100% correct. One is program proving and the other exhaustive testing [Adr82, AnR74, Cri85, How82,87]. Neither approach is currently practical for use with complex software systems. Techniques for proving software correct are not mature enough and exhaustive testing is ruled out principally by the huge number of possible inputs. Although significant progress has been made in developing efficient and effective development and testing techniques which greatly aid in avoiding software faults through formal constructive and analytical methods [e.g. Adr82, How87, Hor87], these techniques do not guarantee production of error-free code. Furthermore, quantitative relationships between software reliability and the quality of the applied development and testing techniques have received relatively little attention. In modern critical systems the problem is further aggravated by the need for extensive concurrent processing.

The only way of handling unknown and unpredictable software failures (faults) is through fault-tolerance. Fault-tolerance already is, or is planned to be, part of many critical software and hardware systems such as nuclear power plants [Gme79, Bis86] and aerospace systems [Mar82, Wil83, Spe84, Mad84, Tro85, Hil85, Avi87, Vog88a]. Two methods for achieving software fault-tolerance are in common use today. These are the N-version programming scheme [Avi77, Che78, Avi84] and the recovery block scheme [Ran75]. Both schemes are based on software component redundancy and the assumption that coincident failures of components are rare and when they do occur responses are sufficiently dissimilar so that the mechanism for deciding answer correctness is not ambiguous. For best results all of these techniques require the component failures to be mutually independent, or at least that the positive inter-component failure correlation is low. Fault-tolerant software (FTS) mechanisms based on redundancy are particularly well suited for parallel processing environments where concurrent execution of redundant components may drastically improve sometimes prohibitive costs associated with their serial execution.

Hence, the study of both multi-version and single version software fault-avoidance and fault-tolerance issues, with an emphasis on the issue of fault correlation in multiple software versions, is of utmost importance where critical software is concerned. We have proposed to study different testing approaches suitable for development of single and multi-version high-reliability software, model single and multi-version reliability, and investigate different fault-tolerance mechanisms.

In the period 1985-87 NASA funded a multi-university experiment to develop 20 functionally equivalent software versions, known as RSDIMU software versions. These versions are to be used to determine the reliability gains of several common fault-tolerant software systems, including N-version programming, recovery-block [Ran75, Avi84], and hybrid schemes such as the consensus recovery block technique [Sco87].

In the period reported here we have worked on the following:

- We have finished developing and evaluating a specialized tool for measuring testing coverage for a variety of metrics.
- We have started using the tool to collect information on the relationships between software faults and coverage provided by the testing process as measured by different metrics (including data flow metrics). We have found considerable correlation between coverage provided by some higher metrics and the elimination of faults in the code.
- We have continued studying back-to-back testing as an efficient mechanism for removal of un-correlated faults, and common-cause faults of variable span.
- We continued studying software reliability estimation methods based on non-random sampling, and the relationship between software reliability and code coverage provided through testing.
- We continued investigating existing, and worked on formulation of new fault-tolerance models. In particular, we have finished simulation studies of the Acceptance Voting and Multi-stage Voting algorithms, and found that these two schemes for software fault-tolerance are superior in many respects to some commonly used schemes. Particularly encouraging are the safety properties of the Acceptance testing scheme.

This report describes the results obtained in the period March 1, 1989 to August 31, 1989.

2. Results

2.1 Fault-Avoidance Through Coverage Testing

BGG, Basic Graph Generation and Analysis tool, was developed to help studies of static and dynamic software complexity, and testing coverage metrics. It is composed of several stand-alone modules, it runs in UNIX environment, and currently handles static and dynamic analysis of control and data flow graphs (global, intra-, and inter-procedural data flow) for programs written in full Pascal. Extension to C is planned. The tool is described in more detail in Appendix I where we describe the structure of BGG, give details concerning the implementation of different metrics, and discuss the options it provides for treatment of global and inter-procedural data flow.

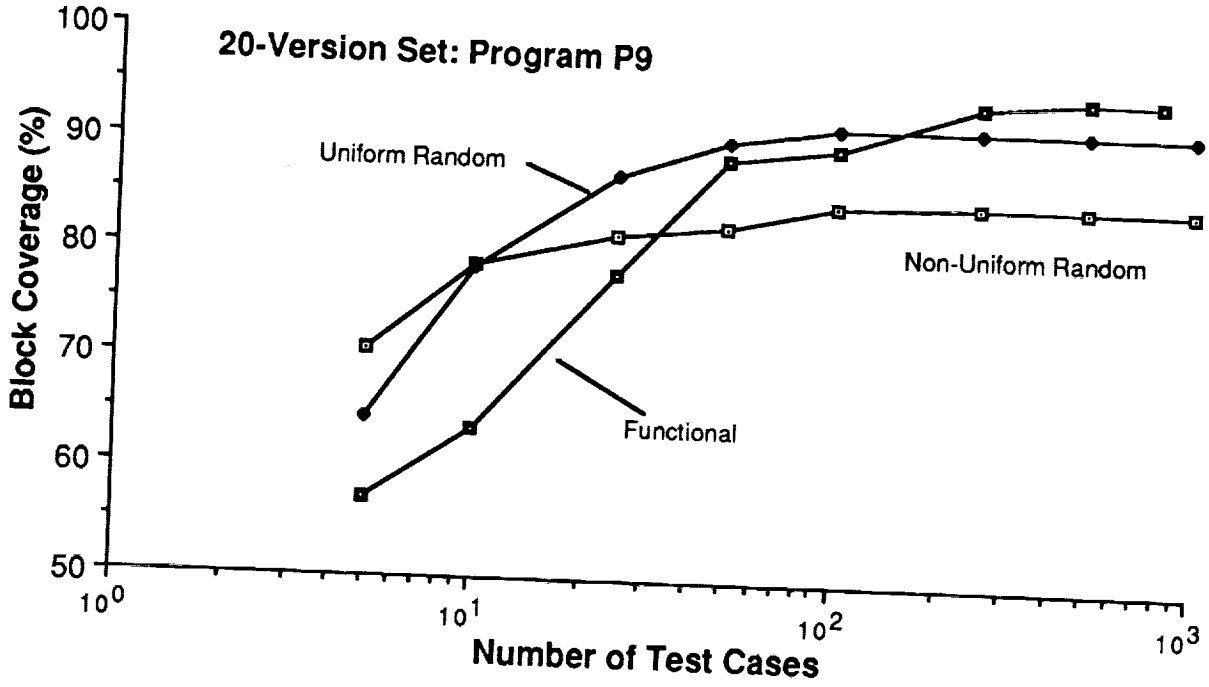

Figure 1. Comparison of linear block coverage observed for two random testing profiles and a functional data for a program out of the 20-version set.

BGG is currently being used to obtain coverage growth curves for acceptance, and other, test data used in the RSDIMU experiment. Figure 1 illustrates the coverage growth curves we have observed with random and functional (designed) test cases for the program program P9 (uclaD) using an early version of the system.

It is interesting to note that coverage growth follows an exponential growth curve, and reaches a plateau extremely quickly. In the example, this happens after about 100 cases. Once the coverage is close to saturation for a particular testing profile, its fault detection efficiency drops sharply. This is illustrated in Figure 2 where we plot the coverage provided by the functional testing profile shown in Figure 1, and the cumulative number of different faults detected using these test cases. Out of the 10 faults that the code contained, 9 were detected with the functional data set used within the first 160 cases.

It is clear that apart from providing static information on the code complexity, and dynamic information on the quality of test data in terms of a particular metric, BGG can also be used to determine the point of diminishing returns for a given data set, and help in making the decisions on when to switch to another testing profile or strategy.

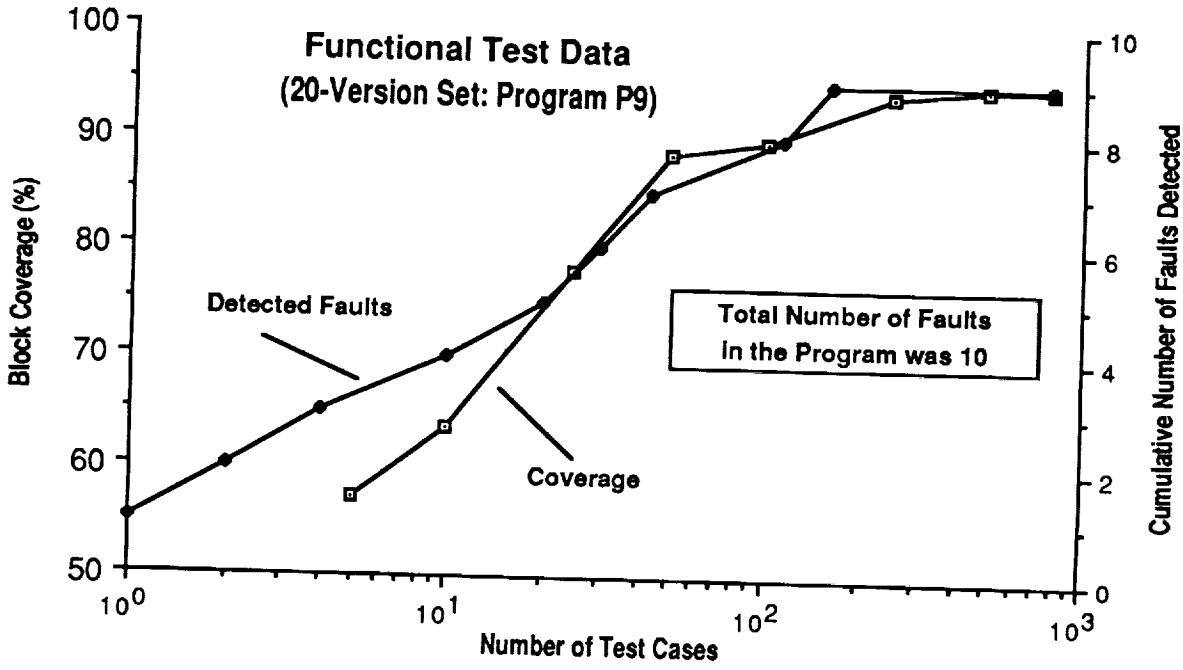

Figure 2. Linear block coverage and fault detection efficiency observed for program P9 with functional acceptance test cases.

Similar measurements have been taken for all 20 version of the RSDIMU set. We are currently studying the correlation between higher metrics, such as p-uses [Fra88], and the reliability of the versions.

2.2 Fault-Avoidance: Using Back-to-Back Testing for Regression Testing

An interesting variant of back-to-back testing is its application to regression testing of a single program. Regression testing is typically conducted either during the production, or in the maintenance phase, after modification of software. The intention is to check back on any changes, and make sure that the changes have not injected, and/or stopped masking, faults, or have corrupted already tested functions and parts of the code. Sometimes it is possible to conduct regression testing using all of the data available for testing, but often, due to execution time and schedule constraints, it is necessary to limit the regression testing to a smaller subset of the test data. An obvious problem that arises during regression testing is the evaluation of the responses received from the newly modified software. If the only failures of concern are self-reporting failures (e.g. system crash, or an obvious disruption of the computer service) a relatively simple acceptance test, or consistency check, may be sufficient to verify the correctness of the answers. On the other hand, if the correctness of the responses is less obvious, then a more elaborate, and often very time consuming, scheme must be used. Comparison of the answers with an existing, progressively generated and growing, database of "correct" answers is a natural solution.

Some of the problems associated with regression testing may be:

1. Regression testing is limited to a smaller subset of the total data set. In this situation there is always some doubt that the "important" test case(s), which could reveal an inadvertently injected bug, is(are) not part of the regression set. Regression testing could be limited to a subset for several reasons. For example, only a limited execution and calendar time is available for the regression testing. This can possibly be alleviated through parallel execution of mutually exclusive but exhaustive subsets of the full test set. Another problem, which may be more difficult to resolve, is the storage problem. It is quite conceivable that the amount of storage required to record the input and output data for a complete set may be inordinately high. However, it is possible that the input set can nevertheless be reproduced, within an acceptable time frame, using some generation algorithm, but that the output verification remains a problem.

2. Regression testing does not employ random data. There are indications that in some circumstances random data may detect more faults than more conventional structured, partitioned and special value testing (e.g. EhE88, Ham88). Therefore, it is desirable to supplement testing

based on a designed (fixed or growing) test set with random test data. The problem is that, unless failures are self-reporting, it may be very expensive to regression test with random data because of storage problems, answer correctness problems and similar.

3. Regression testing does not monitor intermediate program states. There is experimental evidence (e.g. ShL88) that monitoring of internal program states can considerably enhance failure detection efficiency of a testing approach. However, time, storage and correctness problems can present a considerable deterrent to practical use of this technique for regression testing based on the data-base approach.

There are, of course, other possible deficiencies of regression testing that could be discussed, not the least of of them being diminished flexibility of "fixed" data regression sets to changes in the operational input profiles.

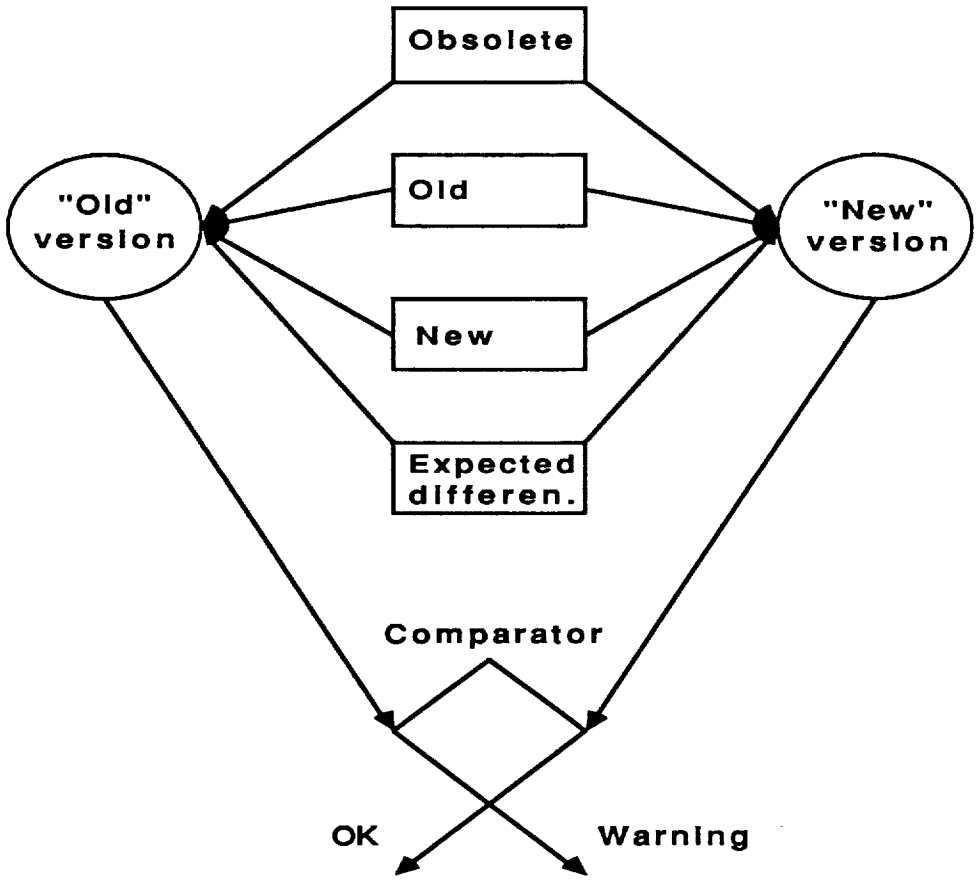

Figure 3. Back-to-back regression testing.

One approach that can help in solving at least some of the problems is back-to-back testing. One of the primary problems with development use of back-to-back testing is the need for independently developed multiple software versions in order to exploit fault detection properties of software diversity. This can be expensive, and it is possible that some of the similar faults will not be detected. This problem does not exist with regression testing. Regression testing is used primarily to make sure that any applied changes have not corrupted the code and functions that have already been tested and found correct. Because generation of a new version of the code is implicit in any software modification, functionally "almost-equivalent" 2-tuples are available at no extra cost.

This means that the "new" and "old" versions of the code can be run against each other to verify invariance of the the functions and responses that were not supposed to be affected by the applied changes. A model of back-to-back regression testing data-flow is illustrated in Figure 3. The circles depict two consecutive versions of the software, the squares the sources of data (files), and the diamond the answer comparator. The response comparisons can be made at almost any desired level; output only, module/function level, intermediate states, even line level. The nice part is that there is practically no problem with the insertion of the sampling probes because the code is not only functionally almost identical, but also structurally very similar (the differences, of course, exist in the modified parts of the code).

We assume that three "types" of regression data are available. An invariant ("old") set, which contains all the test cases which are still valid and completely unchanged following the program modification. A set containing "obsolete" test cases, cases which are no longer valid because of changed requirements, variable ranges, functionality of the code, and similar. And, finally, a set of "new" or changed test cases which contains all the test cases that had to be modified, or were generated completely anew, to accommodate the changes in the functionality and structure of the code. One file, "expected differences", contains a "list" of test cases (and responses) for which the differences between the "old" and "new" code versions would be expected to arise. This data needs to be generated, based on performed modification(s), prior to any regression testing. For example, if upward compatibility of versions is required because the changes are enhancements which should not affect previous performance (e.g. and extension of a communication protocol), then all of the "old" data set responses for key parameters should match (except for new variables), while the "expected differences" will derive primarily from the "new" data set.

There are two general output states of the system. The system either issues a warning, or it accepts the comparison (OK event). In principle, only unexpected differences or unexpected agreement between the outputs should raise an alarm. However, it is prudent to re-examine all outputs where differences arise unless the size and sign of the expected differences is included in the data base. Unexpected disagreements between the versions may be indicative of incompletely corrected faults, newly introduced faults, or old faults that are no longer masked owing to the implemented code changes. The question of tolerances, and false alarms should also be considered [Vou88a]. It is also possible that an expected difference in response does not materialize. This should also be the cause for alarm. The cause could be, for example, that the implemented change was not successful (although not detrimental), or that there is a fault in the test case, etc. The states are illustrated in Figure 4.

A special case is the use of randomly generated data in regression testing. These data sets (either generated dynamically, or in part stored) can be used to probe for possible omissions and "holes" in the regular regression test set. A big advantage that the testing successive versions back-to-back offers is that the random input data, and the corresponding answers, do not have to be stored but can be generated during the testing. Furthermore, the range and the profile of these test cases can be readily changed to accommodate a different operational profile without a (possibly) costly re-generation of the regression data base.

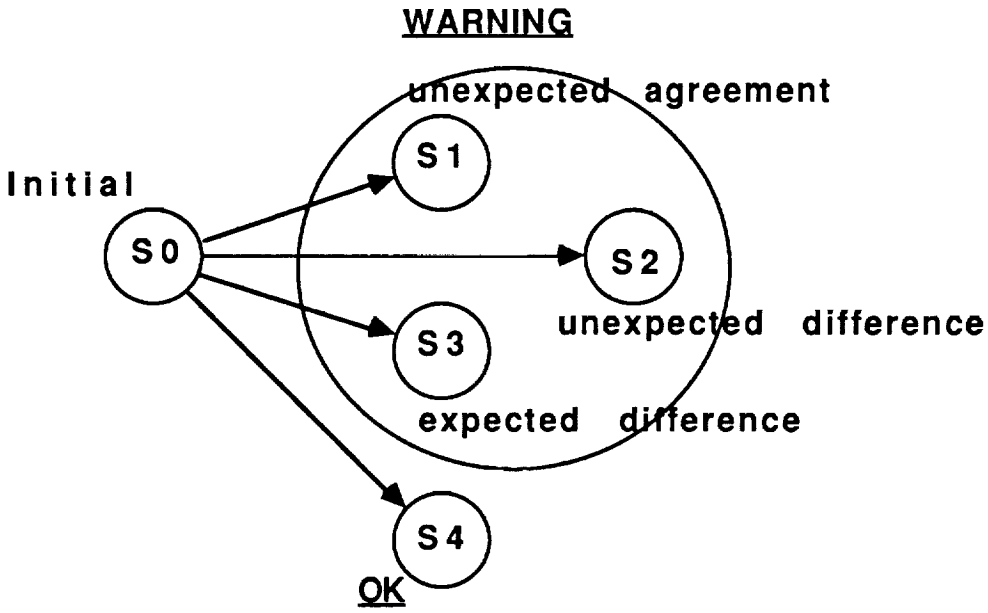

Figure 4. Transition states for (back-to-back) regression testing.

Another obvious advantage of using back-to-back regression testing is that a very large number of variables and intermediate states can be monitored relatively cheaply. This should increase sensitivity of the testing to any anomalies introduced or revealed during the modifications. Furthermore, probing of intermediate states and classification of the expected outputs according to whether a difference would, or would not, be observed with respect to the earlier version can yield useful information about the expected and actual coupling of, and dependencies within, the code (c.f. perturbation or mutation testing).

The cost efficiency of back-to-back regression testing depends on the available resources, and on the nature of the failures. It is shown that the process is not cost-effective if mainly self-reporting failures (differences) are present after the modification, and if the available resources allow for a fast table look-up of the answers. However, the technique becomes particularly effective if random testing is used to supplement regression data sets, a large number of intermediate states is monitored, or there are frequent changes in the operational profile and variable ranges between versions, and, of course, if there are storage problems but input data can be dynamically reproduced.

We are currently in the process of using the incremental correction versions of RSDIMU software to verify usefulness and efficiency of regression back-to-back testing.

2.3 Safety Properties of Some Hybrid Fault-Tolerance Schemes

The performance of classical Majority voting, and of some more reliable hybrid models such as Consensus Recovery Block (CRB) model, deteriorates if the output space is reduced. Binary output space is an extreme case where CRB acts as a simple voter, and the acceptance test is never invoked. This lead us to develop a new hybrid models which with better performance in reduced output space. One ways is to use a better voting strategy (e.g. Consensus Majority Voting [Sun85]. Another is to reduce, or completely eliminate, as many wrong answers as possible before voting.

The model of the scheme we discuss here is called Acceptance Voting (AV). It is illustrated in Figure 5. N functionally equivalent software versions are independently developed, together with an acceptance test, and a voting procedure. When AV is invoked, all versions execute and submit their outputs for acceptance testing. All answers are acceptance tested. Only the outputs

that pass the acceptance test continue on to the voter. Each time the model is invoked it may vote with a different number of outputs, depending on how many results were passed to the voter by the acceptance testing. The voting may be done using any suitable voting scheme. We have examined the influence of three voting schemes, the two-out-of-n voting (2N) [SCO87], the majority voting (MV) and the dynamic majority voting.

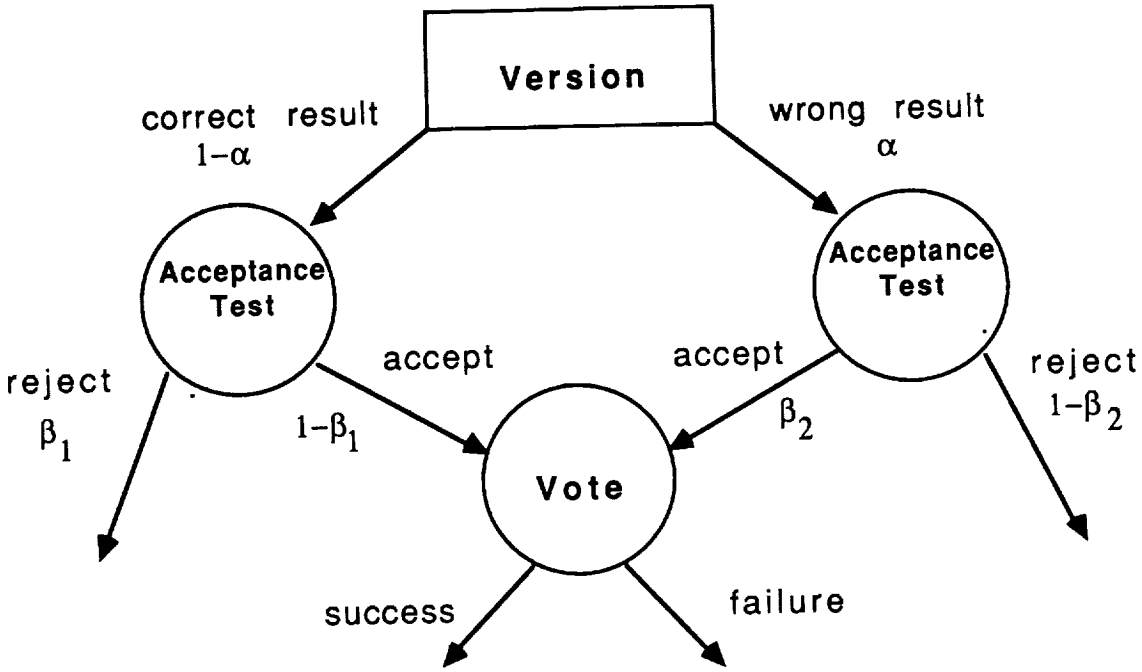

Figure 5 Block diagram of the Acceptance Voting model

Two-out-of-N and Majority voting are well known. In the case of AV we define Dynamic Majority voting in the following way. In dynamic majority voting the agreement number is

$$m = \text{Ceiling } [\frac{(k+1)}{2}]$$

where k is the number of results passed to the voter, and not N. It is important to mention that k changes dynamically , hence m is different for each run. The difference between the dynamic voting and majority voting is that even if a small number of results are passed to the voter, dynamic voting will try to find the majority among them. Majority voting will fail if there are less than majority of the answers passed to voter. Thus, it is better solution than a fixed agreement number used in a majority voting scheme.

A system may be described as safety-critical if an execution time failure result in death, injury, loss of equipment or property, or environmental harm [LEV87]. All failures are not of equal consequences, and a relatively small number of failures are catastrophic in nature. The aim is to eliminate all failures, if possible; if not, as many as possible. This implies that software reliability should be increased or some techniques such as fault-tolerant should be used. A common theory, that a reliable software system is also safe is not necessarily true. This is because, a reliable software may fail causing a catastrophe, on the other hand a less reliable software may fail more number of times, but causing non-vital failures. In fact, it may be desirable to trade a certain amount of overall reliability, for higher safety.

We classify failures into two groups: safe failures and unsafe failures. When

a) System outputs a wrong result as a correct, we have an unsafe failure,

b) System can not decide on the correctness of a result, and is unable to output an answer, but is "aware" of the fact that it will fail, and can therefore forward this knowledge to the user, we have an safe failure.

In Figures 6 and 7 we illustrate our results through the number of observed unsafe-failures (out of a total 100,000 simulation test cases) against the version reliability, for three different methods. We have shown results for binary output space, an extreme situation which approximates the safety behavior of the system in the presence of highly dependent failures, and $\beta_1=\beta_2=\beta$.

In binary output space Consensus Recovery Block acts as a simple voter and is equivalent to N-version programming with majority voting. There is always an answer, right or wrong, that may satisfy required number of agreeing versions. Voter will output this answer as a correct, which may result in an unsafe-failure. The number of unsafe-failures in the AV model is lower than in CRB model. This is because in AV acceptance testing removes most of the wrong answers, there by reducing the probability of them to have the required agreement.

Acceptance test reliability ($1 - \beta$) does not affect unsafe-failures in Consensus Recovery Block model because in binary space the test is never invoked (for odd N). Unfortunately, at the same time, every CRB failure is an unsafe one. Situation improves with larger effective output space cardinality (decision space), but CRB model exhibits a higher number of unsafe-failures than AV model under any output space cardinality or voting strategy. In AV, as the acceptance test deteriorates (β is increases), the number of unsafe-failures is increases for the same version reliability.

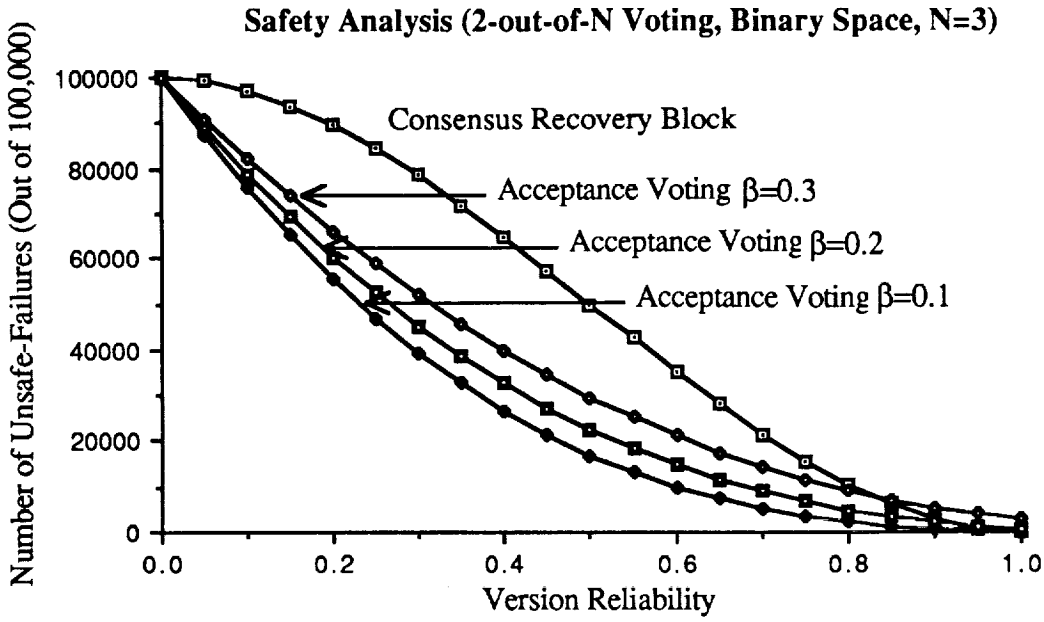

Figure 6 Number of Unsafe Failures (out of 100,000 runs) under 2-out-of-N voting vs. version reliability assuming binary output space, for N = 3, β = 0.1

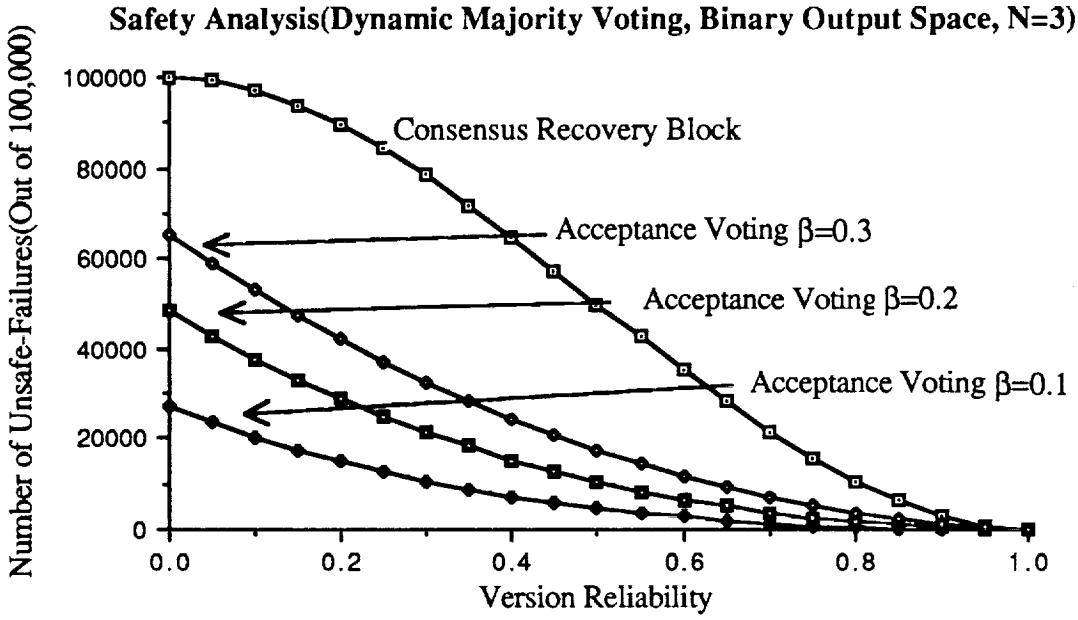

Figure 7 Number of Unsafe Failures (out of 100,000) under Dynamic Majority Voting voting vs. version reliability under binary output space, for N = 3

2.4 Other Work in Progress

1. We are empirically validating Consensus Recovery Block and Majority Consensus voting mechanisms.

2. We continue to investigate cost-effectiveness of multi-version development, testing, and run-time fault-tolerance approaches (assuming single-stage and multi-stage voting). The strategies and methods are being evaluated with respect to the development of a single ultra-high reliability component. Of special interest are the fault-avoidance properties offered by multi-version software development, fault-elimination properties of back-to-back testing, and cost-efficient detection and elimination of correlated faults.

3. We continue to investigate software reliability models in order to provide a basis for estimation of the reliability of the components making up a fault-tolerant software (FTS) system. Software testability modeling, based on control and data flow construct coverage, is being conducted. A coverage based software reliability model will be developed and used as part of the FTS reliability modeling process.

4. We continue to study single stage and multistage voting and fault-tolerant software performance issues. Particular attention is directed towards incorporation of the failure dependencies (positive or negative correlation) into the methods and models used to predict (estimate) reliability offered by a particular fault-tolerance mechanism or strategy.

5. We continue to investigate empirical multi-version software properties. For this we are using the code developed during the summer 1987 RSDIMU certification effort

Bibliography

[Adr82] W.R. Adrion, M.A. Branstad and J.C. Cherniavsky, "Validation, verification, and testing of computer software", ACM Computing Survey, Vol 14(2), 159-192, 1982.

[AnR74] R. Anderson, "Proving programs correct", J. Wiley, 1974

[Avi77] A. Avizienis and L. Chen, "On the Implementation of N- version Programming for Software Fault-Tolerance During Program Execution", Proc. COMPSAC 77, 149-155, 1977.

[Avi84] A. Avizienis and P.A. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol. 17, pp. 67-80, 1984.

[Avi85] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," IEEE Transactions on Software Engineering, Vol. SE-11 (12), pp 1491-1501, 1985.

[Avi87] A. Avizienis and D.E. Ball, " On the Achievement of a Highly Dependable and Fault-Tolerant Air Traffic Control System," IEEE Computer, Vol. 20 (2), pp 84-90,1987.

[Avi88] A. Avizienis, M.R. Lyu, and W. Schutz, "In Search of Effective Diversity: A six-Language Study of Fault-Tolerant Flight Control Software,", Proc. FTCS 18, pp 15-22, June 1988.

[Bha81] B. Bhargava and C. Hua, "Cost Analysis of Recovery Block Scheme and Its Implementation Issues," Int. Journal of Computer and Information Sciences, Vol. 10 (6), pp 359-382, 1981.

[Bis86] P.G. Bishop, D.G. Esp, M. Barnes, P Humphreys, G. Dahl, and J. Lahti, "PODS--A Project on Diverse Software", IEEE Trans. Soft. Eng., Vol. SE-12(9), 929-940, 1986.

[Bis88] P.G. Bishop, and F.D. Pullen, "PODS Revisited--A Study of Software Failure Behavior", Proc. FTCS 18, pp 2-8, June 1988.

[Che78] L. Chen and A. Avizienis, " N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," Proc. FTCS 8, Toulouse, France, pp 3-9, 1978.

[Cri85] F. Cristian, "A rigorous approach to fault-tolerant programming", IEEE Trans. Soft. Eng. Vol. SE-11, 23-31, 1985.

[Dur84] J.W. Duran and S.C. Ntafos, "An evaluation of random testing", IEEE Trans. Soft. Eng., Vol. SE-10, 438-444, 1984

[Eck85] D.E. Eckhardt, Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multi-version Software Subject to Coincident Errors", IEEE Trans. Soft. Eng., Vol. SE-11(12), 1511-1517, 1985.

[Ehr85] W. Ehrenberger, "Statistical Testing of Real Time Software", in "Verification and Validation of Real Time Software", ed. W.J. Quirk, Springer-Verlag, 147-178, 1985.

[EhE88] Willa K. Ehrlich, and Thomas J. Emerson, "The Effect of Test Strategy on Software Reliability Measurement," 11th Minnowbrook Workshop on Software Reliability, July 1988.

[Gil77] T. Gilb, *Software Metrics*, Winthrop Publishers Inc., Cambridge, Massachusetts, 1977.

[Gme79] L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection Systems: An Experiment," Proc. IFAC Workshop SAFECOMP '79, pp 75-79, 1979.

[Grn80] A. Grnarov, J. Arlat, and A. Avizienis, "On the Performance of Software Fault-Tolerance Strategies," Proc. FTCS 10, pp 251-253, 1980.

[Ham88] D. Hamlet, and R. Taylor, "Partition Testing Does not Inspire Confidence," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 206-215, July 1988.

[Hec79] H. Hecht, "Fault Tolerant Software", IEEE Trans. Reliability, Vol. R-28, pp. 227-232, 1979

[Hil85] A.D. Hills, "Digital Fly-By-Wire experience", Proc. AGARD Lecture Series (143), October 1985.

[Hor87] C.A.R. Hoare, "An Overview of Some Formal Methods for Program Design," IEEE Computer, September 1987, pp 85-91.

[How82] W.E. Howden, "Validation of scientific programs", ACM Computing Surveys, Vol. 14(2), 193-227, 1982.

[How87] W.E. Howden, "Functional Program Testing and Analysis", McGraw-Hill Book Co., 1987.

[Kel88] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS 18, pp 9-14, June 1988.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multi-version Programming", IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109, 1986.

[Las83] J.W. Laski and B Korel, "A data oriented program testing strategy", IEEE Trans. Soft. Eng., Vol. SE-9, 347-354, 1983

[Lit87] B. Littlewood, and D.R. Miller, "A Conceptual Model of Multi-Version Software," FTCS 17, Digest of Papers, IEEE Comp. Soc. Press, pp 150-155, July 1987.

[Mad84] W.A. Madden, and K.Y. Rone, "Design, Development, Integration: Space Shuttle Primary Flight Software System", Comm. of the ACM, Vol. 27(8), 902-913, 1984.

[Mar82] D.J. Martin, "Dissimilar Software in High Integrity Applications in Flight Controls", Proc. AGARD - CP 330, 36.1-36.13, September 1982.

[McA85] D.F. McAllister, "Some observations on costs and reliability in software fault-tolerant techniques", TIMS-ORSA Conference, Boston, Mass., April 1985

[McA86] D.F. McAllister, M.A. Vouk and S.W. Yeh, "Execution Time Distributions for Multiversion Software Systems", 1986,

[McA87] D.F. McAllister, C.E. Sun, and M.A. Vouk, "Reliability of Voting in Fault-Tolerant Software Systems for Small Output Spaces", North Carolina State University, Department of Computer Science, Technical Report, TR-87-16, 1987, submitted to IEEE Trans. Reliability.

[Mus87] J. Musa, A. Iannino, and K. Okumoto, "Software Reliability: Measurement, Prediction, Application," McGraw-Hill Book Co., 1987.

[Nag82] P.M. Nagel and J.A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling", BSC-40366, Boeing, Seattle, Wa., 1982

[Nag84] P.M. Nagel, F.W. Scholz and J.A. Skrivan, "Software Reliability: Additional Investigation into Modeling with Replicated Experiments", NASA CR172378, Boeing, Seattle, Wa., 1984

[Neu85] P.G. Neumann, "Some computer related disasters and other egregious horrors", ACM SIGSOFT, Software Engineering Notes, Vol. 10(4), 6-9, 1985; (now a regular SEN column called "Risks to public in computer systems" by the same author).

[Pan81] D.J. Panzl, " A Method for Evaluating Software Development Techniques", The Journal of Systems Software, Vol. 2, 133-137, 1981.

[Ram81] C.V. Ramamoorthy, Y.K.R. Mok, F.B. Bastani, G.H. Chin and K. Suzuki, "Application of a Methodology for the development and validation of reliable process control software," IEEE Trans. Soft. Eng., Vol. SE-7 (6), 537-555, 1981.

[Ram82] C.V. Ramamoorthy and F.B. Bastani, "Software reliability - status and perspectives", IEEE Trans. Soft. Eng., Vol. SE-8, 354-371, 1982

[Ran75] B. Randell, "System structure for software fault-tolerance", IEEE Trans. Soft. Eng., Vol. SE-1, 220-232, 1975.

[Rap85] S. Rapps and E.J. Weyuker, "Selecting software test data using data flow information", IEEE Trans. Soft. Eng., Vol. SE-11, 367-375, 1985

[Sag86] F. Saglietti and W. Ehrenberger, "Software Diversity -- Some Considerations about Benefits and its Limitations", Proc. IFAC SAFECOMP '86, 27-34, 1986.

[Sco84] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Investigating Version Dependence in Fault-Tolerant Software", AGARD 361, pp. 21.1-21.10, 1984

[Sco87] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault-Tolerant Software Reliability Modeling", IEEE Trans. Software Eng., Vol SE-13 (5), 582-592, 1987.

[Shi85] K.G. Shin and .H. Lee, "Evaluation of error recovery blocks used for cooperating processes", IEEE Trans. Soft. Eng., Vol. SE-10, 692-700, 1985

[ShL88] T.J. Shimeall and N.G. Leveson, "An Empirical Comparison of Software Fault-Tolerance and Fault Elimination," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 180-187, July 1988.

[Spe84] A. Spector, and D. Gifford, "The Space Shuttle Primary Computer System", Comm. of the ACM, Vol. 27(8), 874-900, 1984.

[Str85] L. Strigini and A. Avizienis, "Software Fault-Tolerance and Design Diversity: Past Experience and Future Evolution", Proc. IFAC SAFECOMP '85, 167-172, 1986.

[Tha78] R.A. Thayer, M. Lipow and E.C. Nelson, "Software Reliability", North-Holland, Amsterdam, The Netherlands, 1978

[Tra88] P. Traverse, "AIRBUS and ATR System Architecture and Specification," in [Vog88a], pp 95-104, 1988.

[Tro85] R. Troy and C. Baluteau, "Assessment of Software Quality for the Airbus A310 Automatic Pilot", Proc. FTCS 15, Ann Arbor, USA, (IEEE CS Press), 438-443, June 1985.

[Vog88a] U. Voges (ed.), *Software Diversity in Computerized Control Systems*, Springer-Verlag, Wien, Austria, 1988.

[Vog88b] U. Voges, "Use of Diversity in Experimental Reactor Safety Systems," in [Vog88a], pp 29-49, 1988.

[Vou85] M.A. Vouk, D.F. McAllister, K.C. Tai, "Identification of correlated failures of fault-tolerant software systems", in Proc. COMPSAC 85, 437-444, 1985.

[Vou86a] M.A. Vouk, D.F. McAllister, and K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-tolerant Software", Proc. Workshop on Software Testing, Banff, Canada, IEEE CS Press, 74-81, July 1986.

[Vou86b] M.A. Vouk, M.L. Helsabeck, K.C. Tai, and D.F. McAllister, "On Testing of Functionally Equivalent Components of Fault-Tolerant Software", Proc. COMPSAC 86, 414-419, 1986.

[Vou88a] M.A. Vouk, D. F. McAllister, D.E. Eckhardt, A. Caglayan, and J.P.J. Kelly, "Analysis of Faults Detected in a Multiversion Software Testing Experiment," North Carolina State University, Department of Computer Science, Technical Report, TR-88-10, 1988.

[Vou88b] M.A. Vouk, "On Back-To-Back Testing," Proc. COMPASS '88, pp 84-91, June 1988.

[Vou88c] Vouk, M.A., "On Growing Software Reliability Using Back-To-Back Testing," Proc. 11th Minnowbrook Workshop on Software Reliability, pp, July 1988.

[Vou88d] Vouk M.A., "On the Cost of Back-to-Back Testing," Proc.6th Annual Pacific Northwest Software Quality Conference, Lawrence and Craig, Inc., Portland, OR, pp263-282, September 1988.

[Wey88] E.J. Weyuker, "An Empirical Study of the Complexity of Data-Flow Testing," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 188-195, July 1988.

[Wil83] J.F. Williams, L.J. Yount, and J.B. Flanningan, "Advanced Autopilot Flight Director System Computer Architecture for Boeing 737-300 Aircraft," Proc. 5th Digital Avionics Systems Conference, Seattle, WA, 1983.

Appendix I

BGG: A Testing Coverage Tool[1]

Mladen A. Vouk and Robert. E. Coyle[2]

North Carolina State University
Department of Computer Science, Box 8206
Raleigh, N.C. 27695-8206

Abstract

BGG, Basic Graph Generation and Analysis tool, was developed to help studies of static and dynamic software complexity, and testing coverage metrics. It is composed of several stand-alone modules, it runs in UNIX environment, and currently handles static and dynamic analysis of control and data flow graphs (global, intra-, and inter-procedural data flow) for programs written in full Pascal. Extension to C is planned. We describe the structure of BGG, give details concerning the implementation of different metrics, and discuss the options it provides for treatment of global and inter-procedural data flow. Its capabilities are illustrated through examples.

---

[1]Research supported in part by NASA Grant No. NAG-1-983
[2]Teletec Corporation, Raleigh, N.C.

BGG: A Testing Coverage Tool

Mladen A. Vouk and Robert. E. Coyle[3]

North Carolina State University
Department of Computer Science, Box 8206
Raleigh, N.C. 27695-8206

I. Introduction

Software testing strategies and metrics, and their effectiveness, have been the subject of numerous research efforts (e.g. comparative studies by Nta88, Cla85, Wei85, and references therein). Practical testing of software usually involves a combination of several testing strategies in hope that they will supplement each other. The question of which metrics should be used in practice in order to guide the testing and make it more efficient remains largely unanswered, although several basic coverage measures seem to be generally considered as the minimum that needs to be satisfied during testing.

Structural, or "white-box", approaches use program control and data structures as the basis for generation of test cases. Examples include branch testing, path testing [Hen84, Woo80] and various data flow approaches [Hec77, Las83, Rap83, Fra88]. Functional, or "black-box", strategies rely on program specifications to guide test data selection [e.g. How80,87, Dur84]. Some of the proposed strategies combine features of both functional and structural testing as well as of some other methods such as error driven testing [Nta84].

Statement and branch coverage are regarded by many as one of the minimal testing requirements; A program should be tested until every statement and branch has been executed at least once, or has been identified as unexecutable. If the test data do not provide full statement and branch coverage the effectiveness of the employed testing strategy should be questioned. Of course, there are a number of other metrics which can provide a measure of testing completeness. Many of these are more sophisticated and more sensitive to the program control and data flow structure than statement or branch coverage. They include path coverage, domain testing, required elements testing, TER_n ($n \geq 3$) coverage, etc. [How80, Hen84, Whi80, Nta84 and reference therein].

---

[3]Teletec Corporation, Raleigh, N.C.

The simplest data-flow measure is the count of definition-use pairs or tuples [Her76]. There are several variants of this measure. More sophisticated measures are p-uses, all-uses, and du-paths [Fra88, Nta88], ordered data contexts [Las83], required pairs [Nta84,88], and similar. The data-flow based metrics have been under scrutiny for some time now as potentially better measures of the testing quality than control-flow based metrics [e.g. Las83, Rap83, Fra88, Wey88]. However, one recent study [Zei88] indicates that most of the data-flow metrics may not be sufficiently complete for isolated use, and that in practice they should be combined with control-flow based measures.

Over the years a number of software tools for measuring various control and data flow properties and coverage of software code have been reported [e.g. Ost76 (DAVE), Fra86 (ASSET), Kor88]. Unfortunately, in practice these tools are either difficult to obtain, or difficult to adapt to specific languages and research needs, or both. To circumvent that, and also gain better insight into the problematics of building testing coverage tools, we have developed a system for static and dynamic analysis of control and data flow in software.

The system, BGG (Basic Graph Generation and Analysis system), was built as a research tool to help understand, study, and evaluate the many software complexity and testing metrics that have been proposed as aids in producing better quality software in an economical way. BGG allows comparison of coverage metrics and evaluation of complexity metrics. It can also serve as a support tool for planning of testing strategies (e.g. stopping criteria), as well as for active monitoring of the testing process and its quality in terms of the coverage provided by the test cases used. Section II of the paper provides an overview of the BGG system structure and functions. Section III gives details concerning the implementation of various metrics and of handling local, global and inter-procedural data flow. Section IV illustrates the tool capabilities through examples.

II. Structure and Functions

A simplified top level diagram of BGG is shown in Figure 1. BGG is composed of several modules which can be used as an integrated system, or individually given appropriate inputs, to perform static and dynamic analyses of control and data flow in programs written in Pascal. The tool currently handles full Berkeley Pascal[4] with one minor exception. The depth of the "with" statement nesting is limited to one. The extension to greater depth is simple and will be implemented in the next version of the system. BGG runs in UNIX environment. Its

[4]Standard UNIX compiler, pc.

implementation under VM/CMS is planned together with its extension to analysis of programs written in the C language. BGG itself is written in Pascal, C and UNIX C-shell script.

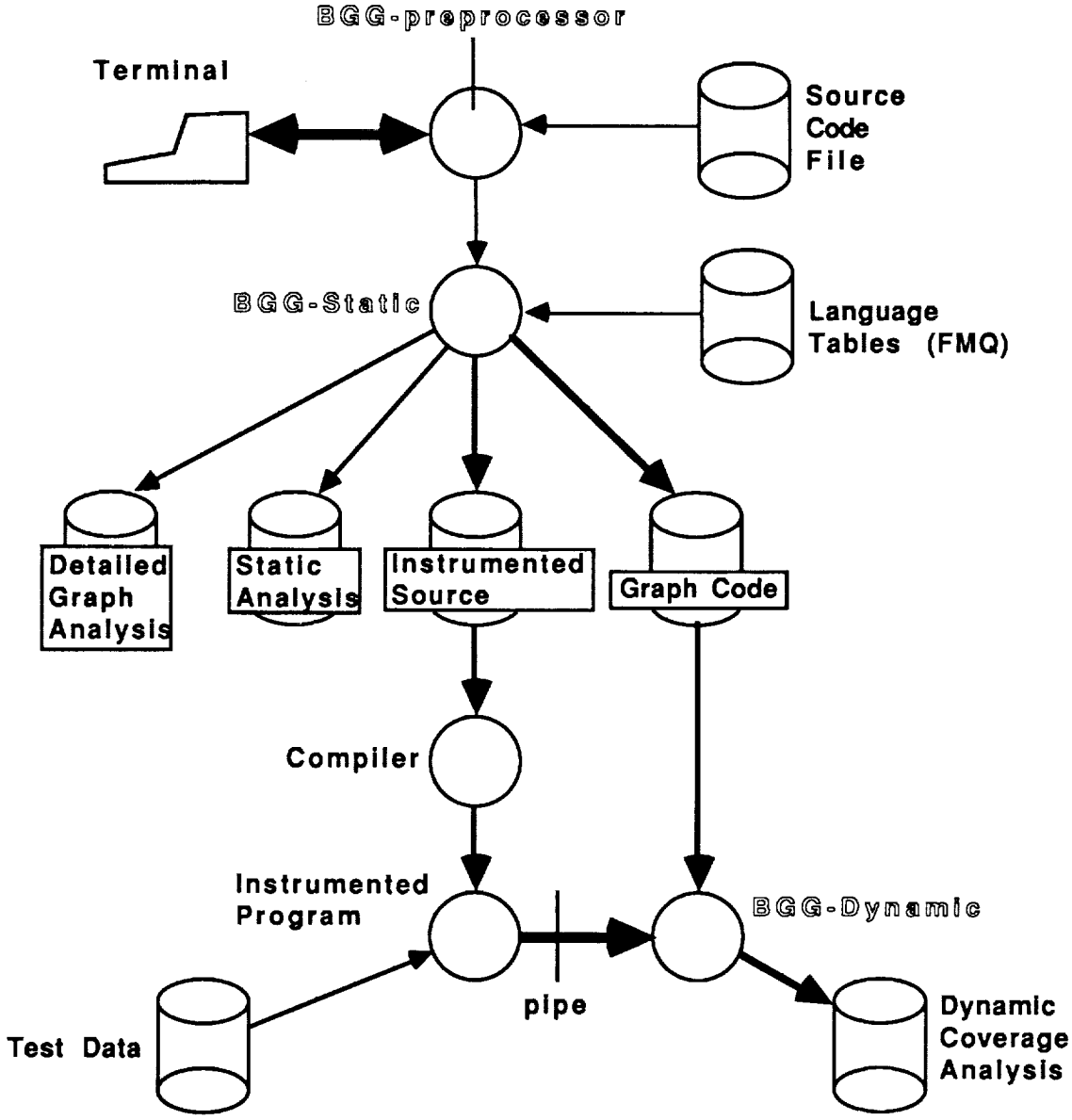

Figure 1. Schematic diagram of the information flow in the BGG system of tools.

BGG pre-processor provides the user interface when the tool is used as an integrated system. It also performs some housekeeping chores (checks for file existence, initializes appropriate language tables and files, etc.), and prepares the code for processing by formatting it and stripping it of comments. The language tables are generated for the system once, during the system installation, and then stored. The front-end parsing is handled through the FMQ generator [Mau81, Fis88].

This facility also allows for relatively simple customization of the system regarding different programming languages and language features. Also, each of the BGG modules has a set of parameters which can be adjusted to allow analyses of problems which may exceed the default values for the number of nodes, identifier lengths, nesting depth, table sizes, etc.

Pre-processed code, various control information and language tables are used as input to the BGG-Static processor. This processor constructs control and data-flow graphs, and performs static analysis of the code. These graphs are the basis for all further analyses. Statistics on various metrics and control-flow and data-flow anomalies, such as variables that are used but never defined etc, are reported. BGG-Static also instruments the code for dynamic execution tracing.

When requested, BGG executes the instrumented code with provided test cases and analyzes its dynamic execution trace through BGG-Dynamic. The dynamic analysis output contains information (by procedures and variables) about the coverage that the test cases provide under different metrics.

When instrumenting code BGG inserts a call to a special BGG procedure at the beginning of each linear code block. It also adds empty blocks to act as collection points for branches. The instrumentation overhead in executable statements is roughly proportional to the number of linear blocks present in the code. In our experience this can add between 50% and 80% to the number of executable lines of code. The run-time tracing overhead for the instrumented programs is proportional to the number of linear blocks of code times the cost of the call to the BGG tracing procedure. The latter simply outputs information about the block and the procedure being executed.

The raw run-time tracing information may be stored in temporary files, and processed by BGG-Dynamic later. However, often the amount of raw tracing information is so large that that it becomes impractical to store it. BGG-Dynamic can then accept input via a pipe and process it on-the-fly. Because BGG-Dynamic analyses may be very memory and CPU intensive, particularly in the case of data-flow metrics, interactive testing may be a slow process. Part of the problem lies in the fact that BGG is still a research tool and was not optimized. We expect that the next version of BGG will be much faster and more conservative in its use of memory. It will permit splicing of information from several short independent runs, so that progressive coverage can be computed without regression runs on already executed data.

Currently BGG computes the following static measures: counts of local and global symbols, lines of code (with and without comments), total lines in executable control graph nodes, linear blocks of code, control graph edges and graph nodes, branches, decision points, paths (the maximum number of iterations through loops can be set by the user), cyclomatic number, definition and use counts for each variable, definition-use (du) pair counts, definition-use-redefinition (dud) chain counts, count of definition-use paths, average definition-use path lengths, p-uses, c-uses, and all-uses. Dynamic coverage is computed for definition-use pairs, definition-use-redefinition chains, p-uses, c-uses and all-uses. Definition-use path coverage and path coverage for paths that iterate loops k times (where k can be set by user) will be implemented. There are several system switches which allow selective display and printing of the results of the analyses.

III. Graphs and Metrics

Control and data flow graphs

Each linear block of Pascal code is a node in a graph. A linear code sequence is a set of simple Pascal statements (assignments, I/O, and procedure/function calls), or it is a decision statement of an iterative or conditional branching construct. When a linear block is entered during execution all of its statements are guaranteed to be executed. Decision statements are always separated out into single "linear blocks". Procedure/function calls are treated as simple statements which use or define identifiers and/or argument variables. A linear block node has associated with it a set describing variables defined in it, and a set describing variables used in it. Also attached to each node is the node execution information.

In each Pascal statement all identifiers for simple local and global variables, named constants defined using CONST, and all built-in Pascal functions are considered. Built in functions are treated as global identifiers. For the purpose of the definition-use analyses, explicit references to elements of an array are treated as references to the array identifier only. Similarly, references to variables pointed to by pointers are currently reduced to references to the first pointer in the pointer chain. An extension that will differentiate between a stand-alone use of a pointer (e.g. its definition or use in a pointer expression), and use of a pointer, or a pointer chain, for de-referencing of another variable, will be implemented in the next version of the tool. Input/output statement identifiers (function names) are considered used, while their argument variables are used (e.g. write, writeln) or defined (e.g. read, readln). The file identifier is treated as a simple variable (defined for input, used for output).

Calls to functions or procedures are treated as local statements which use the procedure/function[5] identifier. In the case of function calls this use is preceded by one or more definitions of the function identifier in the called function itself. This definition is propagated to the point of call, where a single definition of the function identifier is then followed by its local use. From the point of view of the calling procedure, the actual argument variables are either used once, or defined once, or both used and defined once (in that order), depending on whether the corresponding parameter is used (any number of times), defined (any number of times), or used and defined (in any order) in the procedure that is called. Definitions are returned only if the corresponding parameter is a **var** parameter.

The point of call ordering: used-defined, for **var** parameters used and defined in any order, was chosen as a warning mechanism for programmers that have access to analyses of their own code but may not have access to the analyses, or the actual code, of the procedures they call. The idea is to impress on the programmers that the variable may be used in the invoked unit, and therefore they should be careful about what they send to it because the definition may not mask an undefined argument variable, an illegal value, etc. The way we handle procedure arguments permits a limited form of inter-procedural data flow analysis, and offers a more realistic view of the actual flow of information through the code. It also means that the code for the called procedures must be available for BGG to analyze. An alternative is not to use this option, but use the defensive approach of assuming that every argument variable of a **var** parameter is always used and then defined.

A global variable that has been only used in a called procedure, or used in the procedures called within the called procedure, is reported as used at the point of call. A global variable that has been only defined in the called procedure, or deeper, is reported defined at the point of call. However, a global variable that has been used and defined (in any order) in the called procedure, or in any procedure called within the called procedure to any depth, is reported as defined and then used at the point of call. The reason global variables are treated differently from procedure arguments is to highlight global variable definition in the called procedure(s) by making it visible as a definition-use pair at the point of call. Again, it is a form of warning to the programmers that the underlying procedures have changed a global variable value, may have re-used this value, and in turn may

[5]From here on, we use term "procedure" to mean procedure or function, unless a distinction has to be made between the two.

have (if the definition was erroneous) affected values of some, or all, the parameter values passed back to the point of call.

All procedure parameters are assumed to be defined (pseudo-defined) on entry. Global variables used in a procedure are also pseudo-defined on entry. Parameters and global variables set in a procedure or function are assumed used (pseudo-used) on exit. The actual use and definition of completely global variables, and locally global variables, is fully accounted for in each procedure in which they occur as far as their uses and re-definitions are concerned. On return to the calling procedure, any global variables that have been used or defined in the called procedure are reported as single uses and/or definitions of that global variable at the point of call, however, pseudo-uses enabled within a procedure are not reported back to the point of call. The tool has options that allow different treatment of global variables (e.g. pseudo definitions and uses can be switched off), and selective display of the analyses of only some functions and procedures.

Iteration constructs are treated as linear blocks containing the decision statement followed (while, for), or preceded (repeat), by the subgraphs describing the iteration body. Conditional branching constructs (if, case of) consist of decision nodes followed by two or more branch subgraphs. All decision points are considered to have p-uses (edge associated uses) as defined in [Fra88].

Metrics

Some of the static metrics that BGG currently computes are less common or are new and require further explanations.

By default, path counts are computed so that each loop is traversed once. However, definition-use-redefinition chain counts (see below) force on some loops one more iteration in addition to the first traversal. User may change the default number of iterations through a loop through a switch (one value for all loops). Cyclomatic number is computed in the usual way [McC76]. Implemented data flow visibility of all language constructs and variables is such that full definition-use coverage implies full coverage of executable nodes (and in turn full statement coverage) [e.g. Nta88]. BGG computes c-uses, p-uses, and all-uses according to definitions given in [Fra88].

Definition-use-(re)definition, d(u)d, chains are data-flow constructs defined in [Vou84]. It is one of the metrics we are currently evaluating for sensitivity and effectiveness in software fault detection. A d(u)d is a linear chain composed of a definition followed by a number of sequential

uses, and finally by a re-definition of a variable. The basic assumptions behind this metric are a) the longer a d(u)d chain is the more complex is the use of this variable, and b) the more one re-defines a variable the more complex its data-flow properties are. The first property is measured through d(u)d length (see below), the second property is measured by counting d(u)d's. An additional characteristic of d(u)d chains is that they are cycle sensitive and, for those variables where they are present, they force at least two traversals of loops within which a variable is defined. However, full d(u)d coverage does not imply full du-pair coverage. The d(u)d metric is intended as a supplementary measure to other definition-flow metrics.

Definition of a du-path can be found, for example, in [Fra88, Nta88]. A single du-pair may have associated with it one or more du-paths from the definition to that use. We augment the count of du-paths and du-pair counts with measures of du-path lengths. The assumption is that, from the standpoint of complexity (and hence affinity to errors), it is not only the count of du-paths that is important, but also the length of each path. A definition which is used several times, perhaps in physically widely separated places in the program, requires more thought and may be more complex to handle than one that is defined and the used only once, or for the first time. For each du-path we compute length by counting the number of edges covered in reaching the paired use. For every variable we also compute an average length over all du-pairs and du-paths associated with it. In a similar manner we define d(u)d-length as the number of use-nodes between the definition and redefinition points of the chain. Average d(u)d-length is the d(u)d-length accumulated over all d(u)d pairs divided by the number of d(u)d's. We use d(u)d-lengths to augment d(u)d-counts.

We also distinguish between linear (or acyclic) d(u)d's and loop generated, or cyclic, d(u)d's. Cyclic d(u)d's are those where the variable re-defines itself or is re-defined in a cyclic chain. All cyclic constructs are potentially more complex than the linear ones. Comparison is difficult unless the loop count is limited, or looping is avoided, in which case cyclic structures lend themselves to comparison with acyclic ones through unfolding. If iterative constructs are regarded only through du-pairs, many cycles may not be detected since all du-pairs might be generated by going around a loop only once. On the other hand, for a cyclic d(u)d to be generated, a second pass through a loop is always required. However, if there are no definitions of a variable within a loop then the loop would not be registered by d(u)d constructs belonging to that variable. When a variable is only used (or not used at all) within a loop, its value is loop invariant and loop does not add any information that the variable can (legally) transmit to other parts of the graph.

BGG also has facility for computing concentration (or density) of du-paths and d(u)d-paths through graph nodes. We believe that graph (code) sections that show high du-chain and d(u)d-chain node densities may have a higher probability of being associated with software faults than low density regions.

IV. Examples

The examples given in this section derive from an ongoing project where BGG is being used to investigate static and dynamic complexity properties of multi-version software, multi-version software fault profiles, and effectiveness and efficiency of different testing strategies. We are using two sets of functionally equivalent numerical programs for these studies. One set consists of 6 Pascal programs (average size about 500 lines of code) described in [Vou86], the other set consists of 20 Pascal programs (average size about 2,500 lines of code) described in [Kel88].

```
158   function fptrunc(x: real): real;
159
160
161   const
162         largevalue = 1.0e18;
163   var
164         remainder: real;
165         power: real;
166         bigpart: real;
167         term: real;
168   begin
169         remainder := abs(x);
170         if remainder > largevalue then
171               fptrunc := x
172         else begin
173               power := 1.0;
174               while power < remainder do
175                     power := power * 10.0;
176               bigpart := 0.0;
177
178               while remainder > maxint do begin
179
180                     while power > remainder do
181                           power := power / 10.0;
182                     term := power * trunc(remainder / power);
183                     remainder := remainder - term;
184                     bigpart := bigpart + term;
185               end;
186
187               remainder := trunc(remainder) + bigpart;
188
189               if x < 0.0 then
190                     fptrunc := -remainder
191               else
192                     fptrunc := remainder;
193         end;
194   end;
```

Figure 2. Code section for which analysis is shown in Figures 3 and 4

Figure 2 shows a section of the code from program L17.3 of the 6-program suite. Figure 3 illustrates the output that static analysis processor "BGG-Static" offers in the file "Static Analysis" for the same procedure.

Outputs like the one shown in Figure 3 provide summary profile of each local and global symbol found in the code. How many times it was defined (or pseudo-defined), used (or pseudo-used), how many du-pairs it forms, how many d(u)d chains, etc. This static information can be used to judge the complexity of procedures, or the complexity of the use of individual variables. In turn, this information may help in deciding which of the variables and procedures need additional attention on the part of the programmers and testers.

Figure 4 illustrates the detailed node, parameter, and global variable information available in the file labelled "Detailed Graph Analysis" in Figure 1. Figure 4 is annotated (bold text) to facilitate understanding. We see that all parameters (e.g. X), global variables (e.g. TRUNC), and built-in functions (e.g. ABS) are pseudo-defined on entry. The parenthesized number following a capitalized identifier is its number in the symbol table. Note that there are empty nodes, inserted by BGG, which act as collection points for branches (e.g. Block #17). Because FPTRUNC was defined in several places in the code, it is pseudo-used on exit from the function (in Block #18). Note also that built-in function ABS is treated as a global variable, and its parameters are used only (because BGG does not have insight into its code), but the situation is different in the case of locally defined procedures.

For example, Figure 5 shows another section of the code in which procedure ADJUST calls a local function FPMOD (line 285) which, in turn (not shown), calls function FPFLOOR, which then calls function FPTRUNC. The details of the static analysis of the first ADJUST node where the call chain begins are shown in Figure 6. Output lines relevant to the discussion are in bold. Note that FPTRUNC is global from the point of view of ADJUST and is therefore pseudo-defined on entry. The same is true for FPMOD and FPFLOOR. All three are reported as defined and then used in line 285. For two of them the use actually occurs at a deeper level, in function FPMOD for FPFLOOR, and in function FPFLOOR for FPTRUNC. The definitions occur in functions themselves, e.g. for FPTRUNC it occurs in FPTRUNC itself. All these underlying definitions and uses are propagated back to ADJUST.

Procedure: FPTRUNC

Local Symbol Name	Def Ct	Use Ct	DU Ct	DUD Ct	Cyclic DUD Ct	Acyclic DUD Ct	DU Len	DUD Len	Cyc Len	Acyc Len	Subpath Ct	Puse Ct	Cuse Ct
FPTRUNC	3	1	3	0	0	0	1.00	0.00	0.00	0.00	303		
X	1	3	3	0	0	0	1.88	0.00	0.00	0.00	13	2	2
LARGEVALUE	1	1	1	0	0	0	1.00	0.00	0.00	0.00	13	2	0
REMAINDER	3	9	14	9	2	7	4.14	5.33	4.50	5.57	12	12	8
POWER	3	6	16	6	3	3	2.61	2.83	3.00	2.67	16	10	11
BIGPART	2	2	4	4	2	2	1.00	1.00	1.00	1.00	8	0	4
TERM	1	2	2	2	2	0	1.50	2.00	2.00	0.00	4	0	2
Global Symbol Name													
ABS	1	1	1	0	0	0	1.00	0.00	0.00	0.00	13	0	1
MAXINT	1	1	1	0	0	0	1.67	0.00	0.00	0.00	13	2	0
TRUNC	1	2	2	0	0	0	1.40	0.00	0.00	0.00	13	0	2
Procedure Summary:	17	28	47	21	9	12	2.62	3.48	2.67	4.08	108	28	33

Total local symbols:	7
Total global symbols:	3
Total lines (in executable nodes):	27
Total lines (in non-empty xqt nodes):	25
Total blocks:	18
Total non-empty blocks:	16
Total edges:	22
Total branches:	10
Total decisions:	5
Total paths:	13
Cyclomatic number:	6

Figure 3. Static analysis of the procedure shown in Figure 2.

```
Procedure FPTRUNC (193)
  Parameter X (194)

  Global TRUNC (28) used                                  global variable
  Global MAXINT (55) used                                 global variable
  Global ABS (7) used                                     built in function
  Global FPTRUNC (193) defined
------------------------------ ... ------------------------------------
Procedure # 8 : FPTRUNC

Block # 1                                                 not a for-loop
Nodetype: NOT FOR
Predecessor list: none
Successor list: 2
Block start at line: 168
Block ends at line: 169
Definition-use list for this block:
  ABS (7) is defined in line # 168                        pseudo-def
  MAXINT (55) is defined in line # 168                    pseudo-def
  TRUNC (28) is defined in line # 168                     pseudo-def
  LARGEVALUE (195) is defined in line # 168               constant
  X (194) is defined in line # 168                        pseudo-def
  X (194) is used in line # 169
  ABS (7) is used in line # 169                           (1)
  REMAINDER (196) is defined in line # 169

Block # 2
Nodetype: NOT FOR
Predecessor list: 1
Successor list: 4 3
Block start at line: 170
Block ends at line: 170
Definition-use list for this block:
  LARGEVALUE (195) is used in line # 170
  REMAINDER (196) is used in line # 170

Block # 3
Nodetype: NOT FOR
Predecessor list: 2
Successor list: 18
Block start at line: 170
Block ends at line: 171
Definition-use list for this block:
  X (194) is used in line # 171
  FPTRUNC (193) is defined in line # 171
------------------------------ ... ------------------------------------
                                                          empty collector node
Block # 17
Nodetype: NOT FOR
Predecessor list: 15 16
Successor list: 18
Block start at line: 193
Block ends at line: 193
Definition-use list for this block: empty

                                                          end node
Block # 18
Nodetype: NOT FOR
Predecessor list: 3 17
Successor list: none
Block start at line: 194
Block ends at line: 194
Definition-use list for this block:
  FPTRUNC (193) is used in line # 194                     pseudo-use
```

(1) ABS is a built in function and is treated as a global identifier only

Figure 4. Elements of the detailed node analysis

```
274   procedure adjust(var p: point);
275
276
277   var
278         twopi, piover2: real;
279   begin
280         twopi := pi * 2;
281         piover2 := pi / 2;
282
283         begin
284
285               p.long := fpmod(p.long, twopi);
286
287               p.lat := fpmod(p.lat, twopi);
288               if p.lat > pi then
289                     p.lat := p.lat - twopi;
290
291               if p.lat > piover2 then
292                     p.lat := pi - p.lat
293               else if p.lat < -piover2 then
294                     p.lat := -pi - p.lat;
295         end;
296   end;
```

Figure 5. Code section for which static analysis is shown in Figure 6.

Of course, variables strictly local to FTPRUNC, such as "**remainder**" (see also Figures 2, 3 and 4), do not show at the point of call to FPMOD in ADJUST. It is obvious that global data flow can add considerably to the mass of definitions, uses, and other constructs a programmer has to worry about. Nevertheless, we believe that it is a good practice to make this information available so that the full implication of a call to a procedure can be appreciated.

BGG provides coverage information on the program level, and on the procedure level. Figure 7 illustrates output from the dynamic coverage processor "BGG-Dynamic", delivered in the "Dynamic Coverage Analysis" output file, for function FPTRUNC and a set of 103 test cases. In the example some of the output information normally delivered by BGG has been turned off, e.g. all-uses.

For each procedure BGG-Dynamic first outputs a summary of branch coverage information: the block number, statement numbers encompassed by the block, the number of times the block was executed, and the execution paths taken from the block (node). For example, node 5 in Figure 7 was executed 724 times, 6 times to node 3, and 721 times to node 7. Branches which have not been executed show up as having zero executions.

```
Procedure ADJUST (214)
  Parameter P.LONG (216) used
  Parameter P.LONG (216) defined
  Parameter P.LAT (217) used
  Parameter P.LAT (217) defined

  Global WRITELN (35) used
  Global FPFLOOR (200) used
  Global ABS (7) used
  Global MAXINT (55) used
  Global TRUNC (28) used
  Global FPTRUNC (193) used
  Global FPMOD (203) used
  Global PI (107) used
  Global FPMOD (203) defined
  Global FPFLOOR (200) defined
    Global FPTRUNC (193) defined                    procedure in scope, global variable
------------------------------ ••• ------------------------------
Procedure # 13 : ADJUST

Block # 1
Nodetype: NOT FOR
Predecessor list:
Successor list: 2
Block start at line: 279
Block ends at line: 287
Definition-use list for this block:
  PI (107) is defined in line # 279
  FPMOD (203) is defined in line # 279
    FPTRUNC (193) is defined in line # 279          pseudo-definition
  TRUNC (28) is defined in line # 279
  MAXINT (55) is defined in line # 279
  ABS (7) is defined in line # 279
    FPFLOOR (200) is defined in line # 279          pseudo-definition
  WRITELN (35) is defined in line # 279
  P.LAT (217) is defined in line # 279
  P.LONG (216) is defined in line # 279
  P (215) is defined in line # 279
  PI (107) is used in line # 280
  TWOPI (218) is defined in line # 280
  PI (107) is used in line # 281
  PIOVER2 (219) is defined in line # 281                    Beginning of the list of
  TWOPI (218) is used in line # 285          <----------    visible definitions and uses
  P.LONG (216) is used in line # 285                        for line 285 from Figure 5.
  FPMOD (203) is defined in line # 285          calls FPFLOOR
    FPFLOOR (200) is defined in line # 285      calls FPTRUNC
  FPTRUNC (193) is defined in line # 285
  WRITELN (35) is used in line # 285
  FPFLOOR (200) is used in line # 285           actually used in FPMOD
  ABS (7) is used in line # 285
  MAXINT (55) is used in line # 285
  TRUNC (28) is used in line # 285
    FPTRUNC (193) is used in line # 285         actually used in FPFLOOR
    FPMOD (203) is used in line # 285           actually used in ADJUST
  P.LONG (216) is defined in line # 285         <---------- List for line 285 ends
  TWOPI (218) is used in line # 287
  P.LAT (217) is used in line # 287
  FPMOD (203) is defined in line # 287
  FPFLOOR (200) is defined in line # 287
  FPTRUNC (193) is defined in line # 287
  WRITELN (35) is used in line # 287
  FPFLOOR (200) is used in line # 287
  ABS (7) is used in line # 287
  MAXINT (55) is used in line # 287
  TRUNC (28) is used in line # 287
  FPTRUNC (193) is used in line # 287
  FPMOD (203) is used in line # 287
  P.LAT (217) is defined in line # 287          <---------- Block #1 ends
```

Figure 6. Elements of the detailed static analysis of the procedure shown in Figure 5.

Proc. #: 8 Proc/Func Name: FPTRUNC

Block	Statements	Executions	Branches(Executions)
1	168 - 169	721	2(721)
2	170 - 170	721	3(0) 4(721)
3	170 - 171	0	18(0)
4	172 - 173	721	5(721)
5	174 - 174	724	6(3) 7(721)
6	174 - 175	3	5(3)
7	176 - 177	721	8(721)
8	178 - 178	721	9(0) 13(721)
9	178 - 179	0	10(0)
10	180 - 180	0	11(0) 12(0)
11	180 - 181	0	10(0)
12	182 - 185	0	8(0)
13	186 - 188	721	14(721)
14	189 - 189	721	15(66) 16(655)
15	189 - 190	66	17(66)
16	191 - 192	655	17(655)
17	193 - 193	721	18(721)
18	194 - 194	721	

Non-empty blks exec/total non-empty blks = 12 / 16 = 75.00%
Lines in executed nodes/total lines in executable nodes = 18 / 25 = 72.00%
Branches exec/total branches = 6 / 10 = 60.00%
Legend: Ct - Total static count, Def - Definitions, Use - Uses, DU - Definition-Use Pairs, DUD - Definition-Use-(Re)Definition Chains
PecentX - percentage of the static count executed with the current set of test cases, Puse - Predicate-Uses, Cuse - Computational Uses

Local Symbol Name	Def Ct	Use Ct	DU Ct	PercentX	DUD Ct	PercentX	Puse Ct	PercentX	Cuse Ct	PercentX
FPTRUNC	3	1	3	66.67	0	0.00	0	0.00	3	66.67
DU pairs not executed:										
	Defined: 171		Used: 194							
X	1	3	3	66.67	0	0.00	2	100.00	2	50.00
DU pairs not executed:										
	Defined: 168	Used: 171								
LARGEVALUE	1	1	1	100.00	0	0.00	2	50.00	0	0.00
P-uses not executed:										
	Defined: 168		Used: 170		Successor: 3					
REMAINDER	3	9	14	42.86	9	22.22	12	33.33	8	37.50
DU pairs not executed:										
	Defined: 183	Used: 183								
	Defined: 183	Used: 182								

Figure 7. Dynamic analysis of the procedure in Figure 2 based on 103 test cases.

Defined: 183 Used: 180
Defined: 183 Used: 187
Defined: 183 Used: 178
Defined: 169 Used: 183
Defined: 169 Used: 182
Defined: 169 Used: 180

DUD chains not executed:

Defined: 183 Used: 178 Used: 180 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 183 Used: 178 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 183 Used: 178 Used: 187 Defined: 187
Defined: 169 Used: 170 Used: 174 Used: 174 Used: 178 Used: 180 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 169 Used: 170 Used: 174 Used: 174 Used: 178 Used: 180 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 169 Used: 170 Used: 174 Used: 178 Used: 180 Used: 180 Used: 182 Used: 183 Defined: 183
Defined: 169 Used: 170 Used: 174 Used: 178 Used: 180 Used: 182 Used: 183 Defined: 183

P-uses not executed:

Defined: 183 Used: 180 Successor: 12
Defined: 183 Used: 180 Successor: 11
Defined: 183 Used: 178 Successor: 13
Defined: 183 Used: 178 Successor: 9
Defined: 169 Used: 180 Successor: 12
Defined: 169 Used: 180 Successor: 11
Defined: 169 Used: 178 Successor: 9
Defined: 169 Used: 170 Successor: 3

POWER 3 6 16 18.75 6 16.67 10 30.00 11 9.09

DU pairs not executed:

Defined: 181 Used: 181
Defined: 181 Used: 182
Defined: 181 Used: 182
Defined: 181 Used: 180
Defined: 175 Used: 175
Defined: 175 Used: 181
Defined: 175 Used: 182
Defined: 175 Used: 182
Defined: 175 Used: 180
Defined: 173 Used: 181
Defined: 173 Used: 182
Defined: 173 Used: 182
Defined: 173 Used: 180

DUD chains not executed:

Defined: 181 Used: 180 Used: 181 Defined: 181
Defined: 181 Used: 180 Used: 182 Used: 182 Used: 180 Used: 181 Defined: 181
Defined: 175 Used: 174 Used: 175 Defined: 175

Figure 7 (continued 1). Dynamic analysis of the procedure in Figure 2 based on 103 test cases.

Defined: 175 Used: 174 Used: 180 Used: 181 Defined: 181
Defined: 173 Used: 174 Used: 180 Used: 181 Defined: 181
P-uses not executed:
Defined: 181 Used: 180 Successor: 12
Defined: 181 Used: 180 Successor: 11
Defined: 175 Used: 180 Successor: 12
Defined: 175 Used: 180 Successor: 11
Defined: 175 Used: 174 Successor: 6
Defined: 173 Used: 180 Successor: 12
Defined: 173 Used: 180 Successor: 11

BIGPART		2	2	4	25.00	2	0.00	0	0.00	4	25.00

DU pairs not executed:
Defined: 184 Used: 184
Defined: 184 Used: 187
Defined: 176 Used: 184
DUD chains not executed:
Defined: 184 Used: 184 Defined: 184
Defined: 176 Used: 184 Defined: 184

TERM		1	2	2	0.00	1	0.00	0	0.00	2	0.00

DU pairs not executed:
Defined: 182 Used: 184
Defined: 182 Used: 183
DUD chains not executed:
Defined: 182 Used: 183 Used: 184 Defined: 182

Global Symbol Name		Def Ct	Use Ct	DU Ct	PercentX	DUD Ct	PercentX	Puse Ct	PercentX	Cuse Ct	PercentX
ABS		1	1	1	100.00	0	0.00	0	0.00	1	100.00
TRUNC		1	2	2	50.00	0	0.00	0	0.00	2	50.00
	DU pairs not executed:										
		Defined: 168	Used: 182								
MAXINT		1	1	1	100.00	0	0.00	2	50.00	0	0.00
	P-uses not executed:										
		Defined: 168		Used: 178		Successor: 9					
Procedure Summary:		17	28	47	38.30	18	16.67	28	39.29	33	30.30

Figure 7 (continued 2). Dynamic analysis of the procedure in Figure 2 based on 103 test cases.

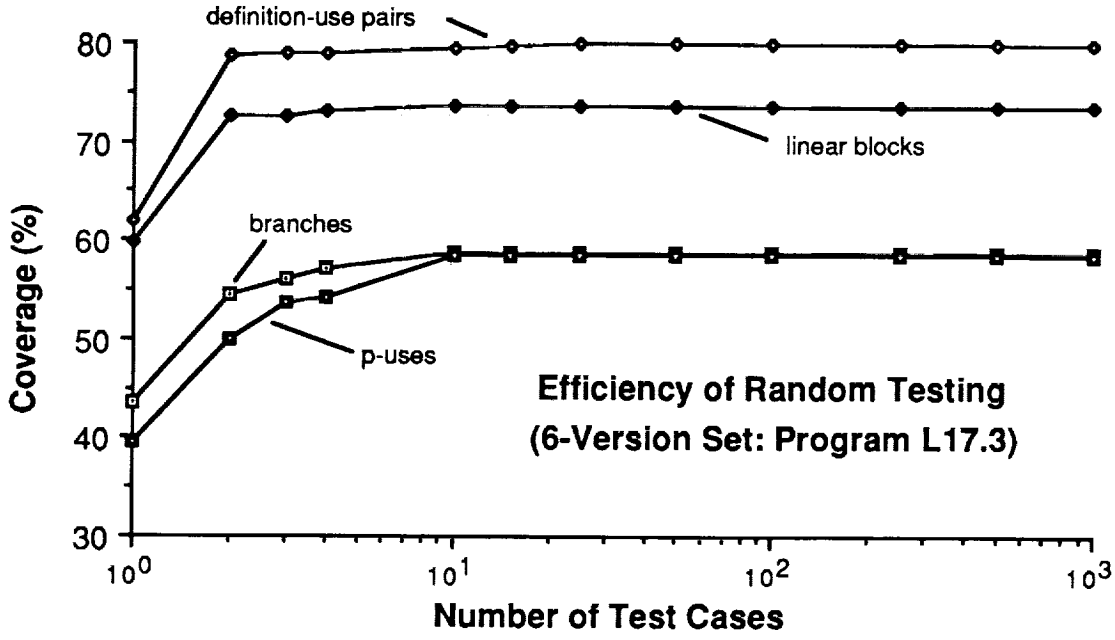

Figure 8. Coverage observed during random testing of a program from the 6-version set.

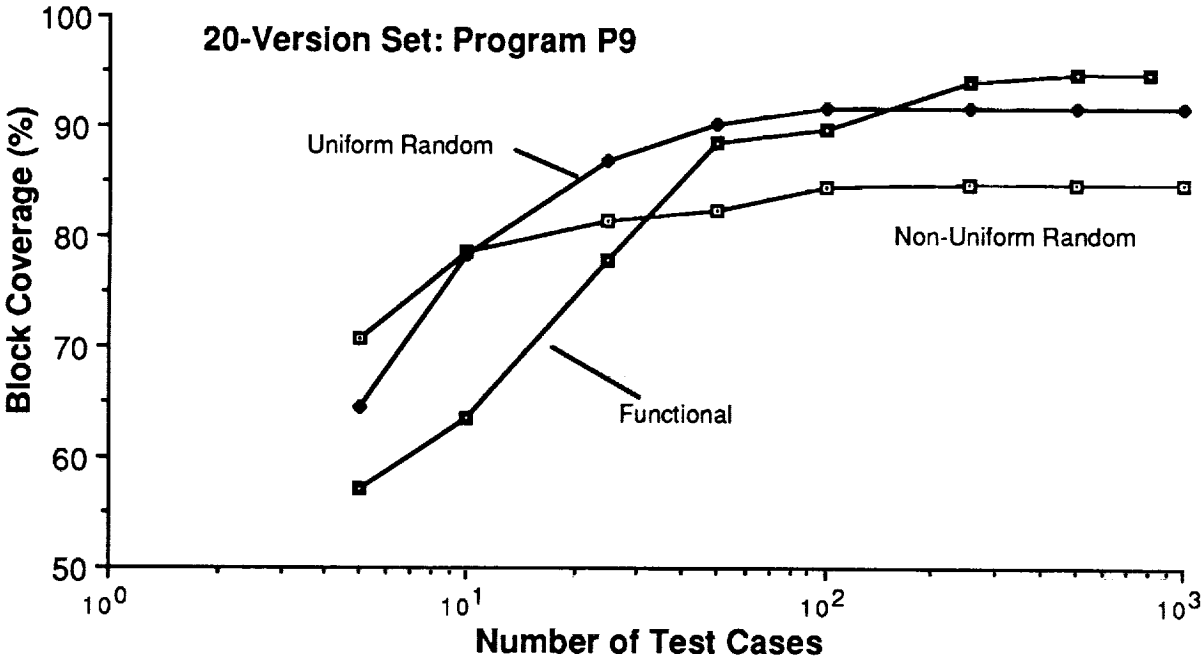

Figure 9. Comparison of linear block coverage observed for two random testing profiles and a functional data for a program out of the 20-version set.

The branch table is followed by a summary of coverage by metrics: coverage for non-empty blocks (blocks that have not been inserted by BGG), lines of code within executable nodes, and branch coverage. This is followed by coverage for data flow metrics by symbol name. The static definition, use, du-pair, d(u)d, p-use, etc. counts for a variable are printed along with the information on its the dynamic coverage expressed as percentage of the executed static constructs. For each identifier, this is followed by a detailed list and description of constructs that have not been executed (e.g. du-pairs or p-uses). Execution coverage output tables can be printed in different formats (e.g. counts of executed constructs, rather than percentages), and with different content (e.g. all-uses).

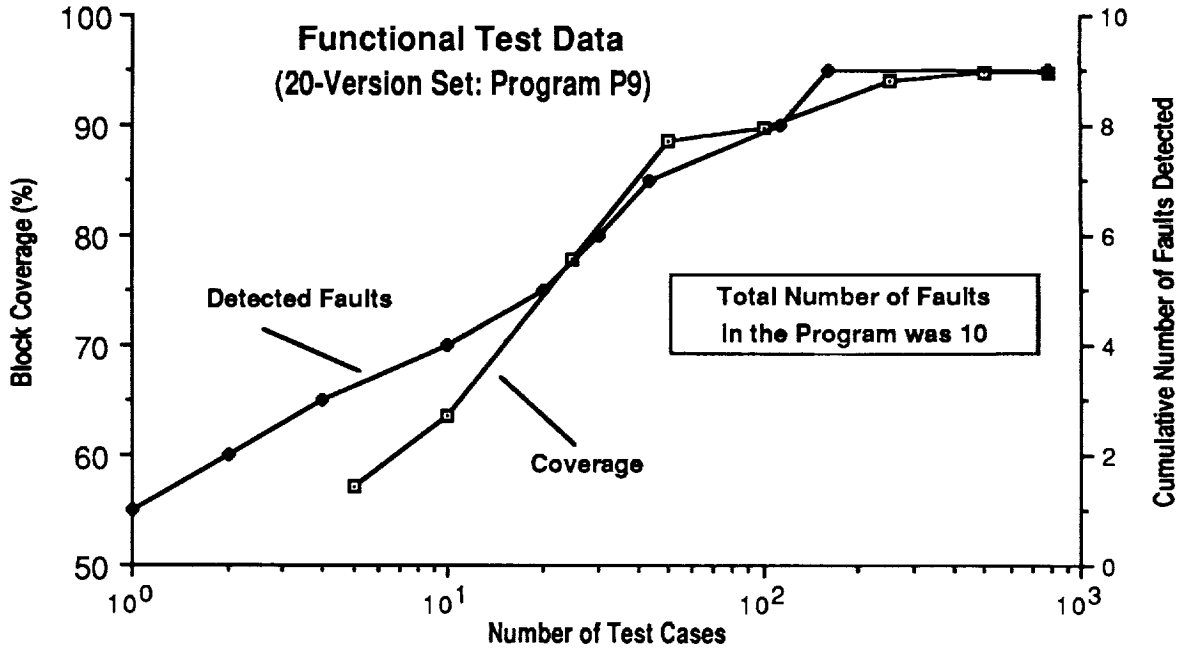

Figure 10. Linear block coverage and fault detection efficiency observed for program P9 of the 20-version set with functional test cases.

BGG can also be used to obtain coverage growth curves for a particular test data set. Figures 8 and 9 illustrate this. They show some of the coverage growth curves we have observed with random and functional (designed) test cases for the program L17.3 of the 6-version set using BGG described here, and for a program P9 from the 20 version set using an early version of the system.

It is interesting to note that both figures show coverage that follows an exponential growth curve and reaches a plateau extremely quickly. For the smaller program (Figure 8, about 600 lines of code) metrics reach saturation already after about 10 cases, while for the larger program (20-version set, about 2,500 lines of code) this happens after about 100 cases. There is also a marked difference in the initial slope and the plateau level obtained with different testing profiles.

Once the coverage is close to saturation for a particular testing profile, its fault detection efficiency drops sharply. This is illustrated in Figure 10 where we plot the coverage provided by the functional testing profile shown in Figure 9, and the cumulative number of different faults detected using these test cases. Out of the 10 faults that the code contained, 9 were detected with the functional data set used within the first 160 cases.

It is clear that apart from providing static information on the code complexity, and dynamic information on the quality of test data in terms of a particular metric, BGG can also be used to determine the point of diminishing returns for a given data set, and help in making the decisions on when to switch to another testing profile or strategy.

V. Summary

We have described a research tool that computes and analyses control and data flow in Pascal programs. We plan to extend the tool into C language. We have found BGG to be very useful in providing information for code complexity studies, in directing execution testing by measuring coverage, and as a general unit testing tool that provides programmers with information and insight that is not available through standard UNIX tools such as the pc compiler, or the pxp processor. We are currently using BGG in an attempt to formulate coverage based software reliability models by relating code complexity, testing quality (expressed through coverage), and the number of faults that have been discovered in the code.

References

[Cla85] L.A. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A comparison of data flow path selection criteria," in Proc. 8th ICSE, pp 244-251, 1985.

[Dur84] J.W. Duran and S.C. Ntafos, "An Evaluation of Random Testing," IEEE Trans. Software Eng., vol. SE-10, pp. 438-444, 1984.

[Fis88] C.N. Fisher and R.J. LeBlanc, *Crafting a compiler*, The Benjamin/Cummings Co., 1988.

[Fra86] P.G. Frankl and E.J. Weyuker, "A data flow testing tool," in Proc. SoftFair II, San Francisco, CA, pp 46-53, 1985.

[Fra88] P.G. Frankl and E.J. Weyuker, "An applicable family of data flow testing criteria," IEEE Trans. Soft. Eng., Vol. 14 (10), pp 1483-1498, 1988.

[Hen84] M.A. Hennell, D. Hedley and I.J. Riddell, "Assessing a Class of Software Tools", Proc. 7th Int. Conf. Soft. Eng., Orlando, Fl, USA,pp 266-277, 1984.

[Hec77] M.S. Hecht, *Flow Analysis of Computer Programs*, Amsterdam, The Netherlands: North-Holland, 1977.

[Her76] P.M. Herman, "A data flow analysis approach to program testing," Australian Comput. J., Vol 8(3), pp 92-96, 1976.

[How80] W. E. Howden, "Functional Program Testing," IEEE Trans. Software Eng., Vol. SE-6, pp.162-169,1980.

[How87] W.E. Howden, "Functional Program Testing and Analysis", McGraw-Hill Book Co., 1987.

[Kel88] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS 18, pp 9-14, June 1988.

[Kor88] B. Koren and J. Laski, "STAD - A system for testing and debugging: user perspective," Proc. Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, Computer Society Press, pp 13 - 20, 1988.

[Las83] J.W. Laski and B. Korel, "A Data-Flow Oriented Program Testing Strategy", IEEE Trans. Soft. Eng., Vol. SE-9, pp 347-354, 1983.

[Mau81] J. Mauney and C.N. Fischer, "FMQ -- An LL(1) Error-Correcting-Parser Generator", User Guide, University of Wisconsin-Madison, Computer Sciences Technical Report #449, Nov. 1981.

[McC76] T. McCabe, "A Complexity Measure," IEEE Trans. Soft. Eng., Vol. SE-2, 308-320, 1976.

[Nta84] S.C. Ntafos, "On Required Element Testing", IEEE Trans. Soft. Eng., Vol. SE-10, pp 793-803, 1984.

[Nta88] S.C. Ntafos, "A Comparison of Some Structural Testing Strategies", IEEE Trans. Soft. Eng., Vol. SE-14 (6), pp 868-874, 1988.

[Ost87] L.J. Osterweil and L.D. Fosdick, "DAVE - a validation, error detection and documentation system for FORTRAN programs," Software - Practice and Experience, Vol. 6, 473-486, 1976.

[Rap85] S. Rapps and E.J. Weyuker, "Selecting software test data using data flow information," IEEE Trans. Soft. Eng., Vol. SE-11(4), pp 367-375, 1985.

[Vou84] M.A. Vouk and K.C. Tai, "Sensitivity of definition-use data-flow metrics to control structures", North Carolina State University, Department of Computer Science, Technical Report: TR-85-01, 1985.

[Vou86] M.A. Vouk, D.F. McAllister, and K.C. Tai, "An Experimental Evaluation of the Effectiveness of Random Testing of Fault-tolerant Software", Proc. Workshop on Software Testing, Banff, Canada, IEEE CS Press, 74-81, July 1986.

[Wei85] M.D. Weiser, J.D. Gannon, and P.R. McMullin, "Comparison of structured test coverage metrics," IEEE Software, Vol 2(2), pp 80-85, 1985.

[Wey88] E.J. Weyuker, "An empirical study of the complexity of data flow testing", Proc. Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, Computer Society Press, pp 188-195, 1988.

[Whi80] L.J. White and E.J. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Trans. Soft. Eng., Vol. SE-6, pp 247-257, 1980.

[Woo80] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience With Path Analysis and Testing of Programs," IEEE Trans. Software Eng., vol.SE-6, pp.278-286, 1980.

[Zei88] S.J. Zeil, "Selectivity of data flow and control flow path criteria", Proc. Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, Computer Society Press, pp 216-222, 1988.