DETECTION OF SOFTWARE FAULTS

Nirvikar Katiyar
Associate Pofessor
NVPEMI Kanpur

Dr. Raghuraj Singh
Prof. CS Deptt.
HBTI Kanpur

Abstract

An important decision in software projects is when to stop testing and based on defect data from an inspection the number of defects present in a software product can be estimated to decide on further development or quality assurance activities. Project managers are primarily interested to determine the number of major defects in the product, which may have a strong impact on product quality as well as on project schedule and cost, if they go undetected. An important aspect of developing models related with number and type of faults in a software system to a set of structural measurement is defining to constitute a fault. By definition, a fault is a structural imperfection in a software system that may lead to the system‘s eventually failing. A precise definition of faults, what are makes it possible to accurately identify and count them, which in turn allows the formulation of models relating fault counts and types to other measurable attributes of a software system. Unfortunately, the most widely used definitions are not measurable. There is no guarantee that two different individuals looking at the same set of failure reports and the same set of fault definitions will count the same number of underlying faults. The incomplete and ambiguous nature of current fault definitions adds a noise component to the inputs used in modeling fault content. If this noise component is sufficiently large, any attempt to develop a fault model will produce invalid results. In this paper, we base our recognition and enumeration of software faults on the grammar of the language of the software system. By tokenizing the differences between a version of the system exhibiting a particular failure behavior, and the version in which changes were made to eliminate that behavior, we are able to

unambiguously count the number of faults associated with that failure. With modern configuration management tools, the identification and counting of software faults can be automated.

Keywords: Software testing, Software quality; Software faults; Defect detection

1. Introduction

Unfortunately there is no particular definition of software fault. In the face of this difficulty, it is rather hard to develop meaningful associative models between faults and code attributes. In calibrating a model, we would like to know, how to count faults in an accurate and repeatable manner, just as we would expect to enumerate statements, lines of code and so on.... In measuring the evolution of a system concerning rates of fault introduction and removal, we measure in units proportional to the way the system changes over time. Changes to the system are visible at the module level, and we attempt to measure at that level of granularity. We will also collect information about faults at the same granularity. A fault is a structural imperfection in a software system that may lead to the system‘s eventually failing. In other words, it is a physical characteristic of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems.

People making errors in their tasks introduce faults into a system. These errors may be errors of commission or errors of omission. There are, of course, differing etiologies for each fault. Faults of commission involve implementing code that is not part of the specification or design. Faults of omission involve lapses wherein a behavior specified in the design was not implemented. In order to count faults, there must be a well-defined method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our static source code measurements.

In a careful examination of software faults over the years, we have observed that the

overwhelming number of faults that are recorded as code faults are really design faults.

Some software faults are really faults in the specification. The design implements the specification and the code implements the design. We must be very careful to distinguish among these fault categories. There may be faults in the specification. The specification may not meet the customer‘s needs. If this problem first manifests itself in the code, it is still not a code fault. It is a fault in the program specification or a specification fault. The software design may not implement the software requirements specification. Again, these design problems tend to be made manifest during software testing. Any such design faults must be identified correctly as design faults. In a small proportion of faults, the problem is actually a code problem. In these isolated cases, the problem should be reported as a code fault. We observed an example of this type of problem in a project on a large embedded software system. The program in question was supposed to interrogate a status register on a particular hardware subsystem for a particular bit setting. The code repeatedly misread this bit. This was reported as a software problem. What really happened was that the hardware engineers had implemented a hardware modification that shifted the position of the status bit in the status register. They had failed to notify the software developers of this material change in the hardware specification. The software system did exactly what it was supposed to do. It is just that this no longer met the hardware requirements. Yet the problem remains on record as a software fault.

2. Related work

Fault constitutes a manifestation of an error in software. A fault, if encountered, may cause a failure‖ [37, 32]. This establishes a fault as a structural defect in a software system that underlies the failure of that system to operate as expected, but does not help in determining the type of failure that was observed, or establish how individual faults may be identified or measured. Some standards address the issue of the type of failure observed by describing schemes for classifying anomalies recorded during software

development and operation. For instance [5] provides details of an anomaly classification process, as well as criteria for classifying the type of anomaly observed, at what point in the development process the anomaly was observed, and the action taken in response to the anomaly. For example, this standard allows classification of the type of behavior exhibited by the anomaly (e.g. precision losses) or the type of defect that led to the anomaly (e.g. _referenced wrong data variable‘).

This type of scheme is helpful in determining the underlying causes of faults and failures, so that the development process may be modified to (1) identify the types of faults on which fault detection and removal resources should be focused for the current development effort, and (2) minimize the introduction of the most common types of faults in future development tasks. However, classification standards do not provide enough information to help count the number of faults in the system. We can also see that some of the anomaly types can readily be traced to a single fault (e.g. _Operator in equation incorrect‘). However, the response an _I/O Timing‘ anomaly may involve changes to many lines of source code spread across multiple source code files. In this case, the standard does not provide enough information to allow counting the number of faults at the module level. Ad-hoc fault taxonomy was developed by the authors in [43, 60] in an attempt to provide an unambiguous set of rules for identifying and counting faults. The rules were based on the types of changes made to source code in response to failures reported in the system. Although the rules provided a way of classifying the faults by type, and attempted to address faults at the level of individual modules, they were not sufficient to enable repeatable and consistent fault counts by different observers to be made. The rules in and of themselves were unreliable. Orthogonal Defect Classification (ODC), initially reported in [1], provides a framework for (1) identifying defect types and the sources of error in a software development effort, (2) determining the effectiveness of the different defect detection techniques and strategies used by the organization, and (3) using the feedback provided by analysis of the defects to help the

organization reduce the number of defects it inserts into its systems like [5].

ODC provides a scheme for classifying defects, which is useful in identifying sources of error at different points in the development process. However, it does not seem possible to use the classification scheme to consistently count faults at the module level. The recognition process for defects is not sufficiently well defined to permit the automatic recognition of these defects. In [27], Frankl et al. develop a model for evaluating test methods by the delivered reliability of the system. The aspect of that work relevant to this paper is the rejection of the traditional notion of faults in favor of the idea of failure regions (_a collection of failure inputs that some change fixes exactly‘). Frankl et al. observe that faults have no unique characterization. A software component fails for some test set and then changed that to get success on that test set.

A simple example illustrates the concept: suppose we have a program composed of two functions, A and B. Function, A computes a real number, and then calls function B to compute the square root of that number. If the value computed by A is less than 0.0, the program will fail. There are two changes that can be made either A can be changed so that it never passes a value less than 0.0 to B (e.g. call B only if the value is not negative), or B can be changed so that negative input values will not cause it to fail (e.g. compute the square root of the input‘s absolute value). We agree that faults do not necessarily have a unique characterization. However, our experience with development efforts indicates that there is often one set of acceptable repair actions that is noticeably less costly than the alternatives, both in terms of the number of affected components and total amount of changes made. Based on our experience, we assume that in a production development environment, developers will seek the least costly alternatives that they believe will affect the required repairs in order to maintain the required delivery schedule. Under this assumption, we can consider the repair actions to be unique, which we can use as the basis of a meaningful fault count their.

3. Identification of software faults

One of the most important considerations in the measurement of software faults is the ability to scale the fault. Not all faults are equal. The software fault size description problem is very similar to that confronted by civil engineers in the construction of a building. When structural concrete is poured to form the columns of a building, some voids will naturally occur in the concrete. Two factors must be considered in the determination of the structural consequences of voids in the pour. First, there is the amount of stress in the vicinity of the void. Second, there is the volume of the void. A small void in a highly stressed location will make the building weak. A large void in the surface of a column may simply create visual problems. Software faults, just like voids in concrete, also are large or small. The term, fault, has a size component just as does a structural void in the concrete pour. Sometimes a simple operator is at fault. The developer used a _C‘ instead of a _K‘. Sometimes two or three statements must be modified, added, or deleted to remedy a single fault.

The subject of this paper is the identification and enumeration of faults that occur in source code. We ought to be able to do this mechanically. That is, it should be possible to develop a tool that could count the faults for us. Further, some program changes to fix faults are substantially larger than are others. We would like our fault count to reflect that fact. If we have accidentally mistyped a relational operator like _!‘ instead of _O‘, this is very different from having messed up an entire predicate clause from an –if statement‖. The actual changes made to a code module are tracked for us in configuration control systems such as –rcs‖ or –cvs‖ [81] as code deltas. We must learn to classify the code deltas that we make as to the origin of the fix. In other words, each change to each module should reflect a specific code fault fix, a design problem, or a specification problem. If we manifestly change any code module, and fail to record each fault as we repaired it, we will pay the price in losing the ability to resolve faults for measurement purposes. We will base our recognition and enumeration of software faults on the grammar of the language of the software system. Specifically, faults are to be found in

statements, executable and non-executable. In the C programming language we will consider the structures shown below to be executable statements.

In very simple terms, these structures will cause our executable statement count, Exec, to change. If any of the tokens change that comprise the statement then each of the change tokens will represent a contribution to a fault count. Non-executable statements are shown below.

<Executable_statement>:: =< labeled_statement>
<Expression>
<Selection_statement>
<Iteration_statement>
<Jump_statement>

We will find faults within these statements.

<declaration>:: = <declaration_specifier>;
<declaration_specifier>= <init_declaration_list>‖;‖

The granularity of measurement for faults will be in terms of tokens that have changed. Thus if one has such type of the following statement in C: a = b + c * d; but it has meant to type a=b+c/d; then there is one incorrect token. In this example, there are eight tokens in each statement. There is one token that has changed. There is one fault. This circumstance is very different when wholesale changes are made to the statement. Consider this statement a = b + c * d; was changed to a = b + (c*x) + sin (z);

We are going to assume, for the moment, that the second statement is a correct implementation of the design and that the first was not. This is clearly not a coding error. In this case there are 7 tokens in the first statement and 14 tokens in the second statement. This is a fairly substantial change in the code. Our fault recording methodology should reflect the degree of the change. The important consideration with this fault measurement strategy is that there must be some indication as to the amount of code that has changed

in resolving a problem in the code. We have regularly witnessed changes to tens or even

hundreds of lines of code recorded as a single _bug' or fault. The only measurable index of the degree of the change is the number of tokens that have changed to ameliorate the original problem.

4. Methodology used for counting the number of faults

Each line of text in each version of a program can be seen as a bag of tokens. That is, there may be multiple tokens of the same kind on each line of the text. When a software developer changes a line of code in response to the detection of a fault, either through normal inspection, code review processes, or as a result of a failure event in a program module, the tokens on that line will change. New tokens may be added, and invalid tokens may be removed. The sequence of tokens may be changed. Enumeration of faults under this definition is simple, straightforward.

Most important of all, this process can be automated. Measurement of faults can be performed very precisely, which will eliminate the errors of observation introduced by existing adhoc fault reporting schemes. An example would be useful to show this fault measurement process. Consider the following line of C code.

(1) a = b + c ;

There are five tokens on this line of code. They are B1= {< a >,< = >,< b >,< + >, < c >} where B1 is the bag representing this token sequence. Now let us suppose that the design, in fact, required that the difference between b and c be computed:

(2) a = b- c ;

There will again be five tokens in the new line of code. This will be the bag B2 = {< a >, < = > , < b > , < - > , < c > } The bag difference is B1 - B2 = {< + >, < - >}. The cardinality of B1 and B2 is the same. There are two tokens in the difference. Clearly, one token has changed from one version of the module to another. There is one fault. Now let

us suppose that the new problem introduced by the code in statement (2) is that the order of the operations is incorrect. It should read:

(3) a = c - b;

The new bag for this new line of code will be B3 = { < a >, < = > , < c > , < - > , < b > }. The bag difference between (2) and (3) is B2 - B3 = { }. The cardinality of B2 and B3 is the same. This is a clear indication that the tokens are the same but the sequence has been changed. There is one fault representing the incorrect sequencing of tokens in the source code. Now, to continue the example above, let us suppose that we are converging on the correct solution however our calculations are off by 1. The new line of code will look like this.

(4) a = 1 + c – b ;

This will yield a new bag B4 = { < a >, < = > , < b > , < + > , < c > , < - > , < b > }. The bag difference between (3) and (4) is B3 - B4 = {< 1 >, < + > }. The cardinality of B3 is five and the cardinality of B4 is seven. Clearly there are two new tokens. By definition, there are two new faults. It is possible that a change will span multiple lines of code. All of the tokens in all of the changed lines so spanned will be included in one bag. This will allow us to determine just how many tokens have changed in the one sequence.

(5) Review Aspects

We have proposed a meaning of software faults that can be applied to source code. The definition allows faults to be unambiguously measured at the level of individual modules. Since faults are measured at the same level at which structural measurement are taken, it becomes more feasible to construct meaningful models relating the number of faults inserted into a software module to the amount of structural change made to that module. Because of the way in which faults are defined, the task of counting faults is easily automated; making it much more practical to analyze large software systems.

In other words, the faults may be quantified by a software tool that can analyze the deltas in code modules maintained by the configuration control system and measure those changes specifically attributable to failure reports. When a fault was initially inserted into a component is based on the ability of the revision control system to identify the version in which each line first appeared in the module. For faults whose repair involves removing or modifying a line, determination is straightforward when the fault was introduced into the module.

However, if the repair activity involves adding a line, determining the version into which the fault was inserted is more complicated. We need to examine the context in which the repair is made to determine the first version of the module in which the absence of the line would have constituted a fault. As an approximation, we can determine when the lines bounding the repair first appeared in the module.

References:

[1] Chillarege R, Bhandari I, Chaar J, Halliday M, Moebus D, Ray B, Wong M-Y. Orthogonal defect classification—a concept for inprocess measurement. IEEE Trans Softw Eng 1992;943–6.

[2] Ackerman AF, Buchwald LS, Lewski FH. Software inspections: an effective verification process. IEEE Softw 1989;31–6.

[3] Basili VR, Perricone B. Software errors and complexity: an empirical investigation. ACM Commun 1984;27(1):45–52.

[4] Bisant David B, Lyle James R. A two-person inspection method to improve programming productivity. IEEE Trans Softw Eng 1989; 15(10):1294–304.

[5] IEEE Std 1044-1993. IEEE standard classification for software anomalies. Institute of Electrical and Electronics Engineers; 1994.

[6] Britcher Robert N. Using inspections to investigate program correctness. IEEE Comput 1988;38–44.

[7] Card DN, Glass R. Measuring software design quality. New York, NY: Addison-Wesley; 1990.

[8] Card DN. Learning from our mistakes with defect causal analysis. IEEE Softw 1998;September.

[9] Conley JM. Tort theories of recovery against vendors of defective software. Rutgers Comput Technol Law J 1987;13:1–32.

[10] Daam A. The effectiveness of software error detection mechanisms in real-time operating systems. Proceedings of international symposium on fault tolerant computing; 1986. p. 171–176.

[11] Deswarte Y, Kanoun K, Laprie JC. Diversity against accidental and deliberate faults. Proceedings of computer security, dependability and assurance, York, England; 1998.

[12] Dobbins, James H. Inspections as an up-front quality technique. In: Handbook of software quality assurance. Van Nostrand Reinhold; 1987. p. 137–177.

[13] Ebenau RobertG, Strauss SusanH. Software inspection process. New York, NY: McGraw-Hill; 1994.

[14] Eick Stephen, Loader Clive R, Long M David, Votta Lawrence G, Vander Wiel Scott. Estimating software fault content before coding. Proceedings of the fourteenth international conference on software engineering, Melbourne, Australia; May 1992. p. 59–65.

[15] Eick, Stephen, Clive R. Loader, Scott Vander Wiel, Lawrence G. Votta. How many errors remain in a software design document after inspection? In: Interface ‘93 conference proceedings, San Diego, California, March; 1993. p. 195–202.

[16] Endres A. An analysis of errors and their causes in systems programs. IEEE Trans S/E 1975;1:140–9.

[17] Franz Louis A, Jonathan C Shih. Estimating the value of inspections and early testing for software projects. Hewlett-Packard J 1994; December:60–7.

[18] Robinson SH. Finite-state machine synthesis for continuous, concurrent error detection using signature-invariant monitoring. Research Report CMUCAD-92-36, Carnegie Mellon

[19] Glass R. Persistent software errors: 10 years later. Proceedings of first international S/W test, analysis and review conference, Jacksonville, FL; 1992.

[20] Gould JD, Drongowski P. An exploratory study of computer program debugging. Hum Factors 1974;16(3):258–77.

[21] Gould JD. Some psychological evidence on how people debug computer programs. Int J Man–Machine Stud 1975;7:151–82.

[22] Graham D. Test is a four letter word: the psychology of defects and detection. Proceedings of first international S/W test, analysis and review Conference, Jacksonville, FL; 1992.

[23] Hendrickson E. Bug hunting: going on a software safari. Proceedings of software testing analysis and review conference, Orlando, FL; 2001.

[24] Wakerly J, Wakerly J. Error detecting codes, self checking circuits and applications. Englewood Cliffs: Prentice-Hall; 1982.

[25] Weiser M. Programmers use slices when debugging. Commun ACM 1982;25:446–52.

[26] Holmquist LP, Kinney LL. Concurrent error detection for restricted fault sets in sequential circuits and microprogrammed control units using convolutional codes. Proceedings of international test conference; 1991. p. 926–935. J.C. Munson et al. / Advances in Engineering Software 37 (2006) 327–333

[27] Frankl P, Hamlet D, Littlewood B, Stringini L. Evaluating testing methods by delivered reliability. IEEE Trans Softw Eng 1998;24(8): 586–601.

[28] Hops JM, Sherif JS. Development and application of composite complexity models and a relative complexity metric in a software maintenance environment. J Syst Softw 1995;31:157–69.

[29] Kelly J, Sherif Joseph S, Hops Jonathan. An analysis of defect densities found during software inspections. J Syst Softw 1992;17: 111–7.

[30] Khoshgoftaar TM, Allen EB. Predicting fault-prone software modules in embedded systems with classification trees. Int J Reliab Qual Saf Eng 2002;9(1):1–16.

[31] Houlihan P. Targeted software fault insertion. Proceedings of software testing analysis and review conference, software quality engineering, Orlando, FL; May 2001.

[32] IEEE Std 729-1983. IEEE standard glossary of software engineering terminology. Institute of Electrical and Electronics Engineers; 1983.

[33] Iyengar VS, Kinley LL. Concurrent fault detection in microprogrammed control units. IEEE Trans Comput 1985;34(9):810–21.

[34] Jackson Daniel. Aspect: an economical bug-detector. Proceedings of he thirteenth international conference on software engineering; 1991. p. 13–22.

[35] Basili VR, Boehm Barry. Software defect reduction top 10 list. Computer 2001;34(1):135–7.

[36] Namjoo N. Cerberus-16: an architecture of a general-purpose watchdog processor. Proc. 13th int. symposium on fault tolerant computing; 1983. p. 216-19.

[37] IEEE Std 982.1-1988. IEEE standard dictionary of measures to produce reliable software. Institute of Electrical and Electronics Engineers; 1989.

[38] Beahan JJ, Edmonds L, Ferraro RD, Johnston A, Katz D, Some RR. Detailed radiation fault modeling of the remote exploration and experimentation (REE) first generation testbed architecture. Aerospace conference proceedings, vol. 5; 2000. p. 279–291.

[39] Johnson Philip M. An instrumented approach to improving software quality through formal technical review. Proceedings of the sixteenth international conference on software engineering, Sorrento, Italy; May 1994. p. 113–122.

[40] Johnson WL, Draper S, Soloway E. An effective bug classification scheme must take the programmer into account. Proceedings of high level debugging, Palo Alto, CA; 1983.

[41] Jones WK. Product defect causing commercial loss: the ascendancy of contact over tort, vol. 44. University of Miami Law Review; 1990. p. 731.

[42] Kane JR, Yau SS. Concurrent software fault detection. IEEE Trans Softw Eng 1975;1(1):87–99.

[43] Nikora A, Munson J. Finding fault with faults: a case study, with J. Munson. Proceedings of the annual oregon workshop on software metrics, Coeur d'Alene, ID; 1997, May, 11-3.

[44] Kaner C, Bach J. Exploratory testing in pairs. Proceedings of software testing analysis and review conference, software quality engineering, Orlando, FL; May 2001. p. 622–628.

[45] Khoshgoftaar TM, Munson IC. Predicting software development errors using complexity metrics. IEEE J Sel Areas Comm 1990; 253–61.

[46] Madeira H, Camoes J, Silva JG. A watchdog processor for concurrent error detection in multiple processor systems. Microprocessors Microsyst 1991;15(3):123–31.

[47] Levy LB, Bell SY. Software product liability: understanding and minimizing the risks. High Tech Law J 1990;5:1–27.

[48] MacDonald F, Miller J, Brooks A, Roper M, Wood M. A review of tool support for software inspection. Technical Report RR-95-181. Glasgow, Scotland: University of Strathclyde; January 1995.

[49] Madeira H, Quadros G, Silva JG. Experimental evaluation of a set of simple error detection mechanisms. Microprocess Microprog 1990; 30:513–20.

[50] Mahmood A, McCluskey EJ. Concurrent error detection using watchdog processors—a survey. IEEE Trans Comput 1988;37(2): 160–74.

[51] Mahmood A, McCluskey EJ, Lu DJ. Concurrent fault detection using a watchdog processor and assertions. Proceedings of international test conference; 1983. p. 622–628.

[52] Marik B. Using ring buffer logging to help find bugs. http:// visibleworkings.com/trace/Documentation/Ring-buffer.

[53] Massingale CS, Borthick AF. Risk Allocation for Injury due to Defective Medical Software. J Prod Liab 1988;11:181–98.

[54] Mays RG. Experiences with defect prevention. IBM Syst J 1990;29.

[55] Mellor P, Mellor P. Failures, faults, and changes in dependability measurement. J Inf Softw Technol 1992;34(10):640–54.

[56] Michel E, Michel E, Hohl W. Concurrent error detection using watchdog processors in the multiprocessor system MEMSY. In: Fault tolerant computing systems. Informatik Fachberichte, 283. Springer; 1991. p. 54–64.

[57] Mahmood A, Ersoz A, McCluskey EJ. Concurrent system-level error detection using a watchdog processor. Proceedings of fifteenth international symposium on fault tolerant computing; 1985. p. 145–152.

[58] Munson J, Nikora A. Toward a Quantifiable Definition of Software Faults. Proceedings of the 13th IEEE international symposium on software reliability engineering, IEEE Press.

[59] Namjoo M, McCluskey EJ. Watchdog processors and capability checking. Proc. 12th int. symposium on fault tolerant computing; 1982. p. 245-48.

[60] Nikora A. Software system defect content prediction from development process and product Characteristics. Doctoral Dissertation, Department of Computer Science, University of Southern California; May 1998.

[61] Nikora A, Munson J. A practical software fault measurement and estimation framework. Proceedings of the 12th international symposium on software reliability engineering, Hong Kong; 2001, Nov. 27-30.

[62] Nimmer R, Krauthaus PA. Computer error and user liability risk. Jurimetrics J 1986;121–37.

[63] O'Neill D. Software inspections: more than a hunt for errors. Crosstalk. J Defense Softw Eng 1992;30:8–10.

[64] Ohlsson J, Rimen M, Gunneflo U. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog. Proc. 22nd int. symposium on fault tolerant computing; 1992. p. 316-25.

[65] Perry DE, Stieg CS. Software faults in evolving a large, real-time system: a case study. 4th European software engineering conference- ESEC93; Sept. 1993. p. 48–67.

[66] Perry DE, Evangelist WM. An empirical study of software interface faults-an update. Proceedings of the 20th annual Hawaii international conference on systems sciences, vol. II. p. 113–126; Jan. 1987.

[67] Pettichord B. Beyond the bug battle. Proc. software testing analysis and review conf., Orlando, FL; 2000.

[68] Rifkin S, Deimel L. Applying program comprehension techniques to improve software inspections. Proceedings of the 19th annual NASA software engineering laboratory workshop, Greenbelt, MD; Nov. 1994.

[69] Rumelhart D, Hinton G, Williams R. In: Learning internal representations by error propagation. Parallel distributed processing, vol. 1. MA: MIT Press, Cambridge; 1986. p. 318–62.

[70] Schneider GM, Martin J, Tsai W-T. An experimental study of fault detection in user requirements documents. ACM Trans. Softw Eng Meth 1992;1(2):188–204.

[71] Schneidewind NF. Analysis of error processes in computer software. Proc. int. conf. reliable software, Los Alamitos, CA; 1975. p. 337–46.

[72] Senders JW, Moray NP. Human error: cause, prediction, and reduction. NJ: Lawrence Erlbaum; 1991.

[73] Shen JP, Schuette MA. On-line self-monitoring using signature instruction streams. Proc Int Test Conf 1983;275–82.

[74] Shen JP, Tomas SP. A roving monitoring processor for detection of control flow errors in multiple processor systems. Microprocess Microprogramm 1987;20:249–69.

[75] Shen VY, Yu T, Thebault SM, Paulsen LR. Identifying Error Prone Software: An Empirical Study. IEEE Trans Software Eng 1985;11: 317–23.

[76] Sherif JS, Sherif JS, Ng E, Steinbacher J. Computer software development: quality attributes, measurements and metrics. Nav Res Logist 1988;35:425–36.

[77] Sosnowski J, Sosnowski J. Concurrent error detection using signature monitors. In: Fault Tolerant Computing Systems. Informatik Fachberichte, vol. 214. Berlin: Springer Verlag; 1989. p. 341-355.

[78] Stott DT, Floering B, Burke D, Kalbarcysk Z, Iyer RK, NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. Proceeding Computer Performance and Dependability Symposium; 2000. p. 91–100.

[79] Telles M, Telles M, Hsieh Y. The science of debugging. Arizona: Scottsdale; 2001.

[80] Voas J, Voas J, McGraw G. Software fault injection: inoculating programs against errors. New York: Wiley; 1998. University; 1992.

[81] Cederqvist P. Version management with CVS for CVS 1.11.1p1, available at: http://www.cvshome.org/docs/manual/.