Software Fault Avoidance

Goutam Kumar Saha

Senior Member IEEE

gksaha@ieee.org

All software faults are basically design faults. Correct specification and correct implementation are must in order to produce correct software. Software fault avoidance aims to produce fault free software through various approaches having the common objective of reducing the number of latent defects in software programs. Software fault avoidance approaches include: formal or precise specification practices, programming disciplines like information hiding and encapsulation, extensive and repetitive reviews and formal analyses during the development process, and of course, rigorous testing. In other words, *software fault avoidance* approaches include verification & validation, software testing, and proof methodology. Rigorous development process (standard development processes, capability maturity model), strongly typed languages, comprehensive standards, support tools and highly trained manpower and formal methods are the key factors to software fault avoidance.

Formal methods are fault avoidance techniques that aim to increase dependability by eliminating errors at the requirements specification and design stages of development. Formal specifications use formal language with mathematical semantics. Mathematical semantics make analysis related to syntax checking, type checking possible. Formal specifications help in software design, code refinement, and proof correctness by construction. Formal or semi-formal specifications and programming are useful to show how the codes agree to the specifications and they force us to program more simply and more clearly. As a result, many defects are eliminated. Verification uncovers additional defects and encourages careful examination of the program for efficiency and other quality aspects.

Software testing aims to compensate for human fallibility and to unveil program bugs. A *test* normally shows the presence of faults, not their absence. Bernstein points out, "Typically, testing alone cannot fully verify that software is complete and correct. In addition to testing, other verification techniques and a structured and documented development process must be combined to assure a comprehensive validation approach". IBM's Cleanroom Software Engineering methods aim toward Zero-Defect Programming and these methods are also applicable to three key areas of software development: software specification, verification and testing.

Fault avoidance aims to prevent faults from occurring in the operational system. It limits introduction of faults during system construction. It includes fault prevention, fault removal, and fault forecasting. Fault prevention attempts to eliminate any possibility of faults creeping into a system before it goes operational. Fault removal attempts to find and remove the causes of errors.

Thus, fault avoidance helps to improve the quality of both the components and the systems. Approaches for software fault avoidance include a set of methods and techniques intended both to reduce the presence and to avoid the introduction of faults (in number and severity). When designing dependable systems we must deal with dependability issues from the beginning by addressing fault-tolerance mechanisms within the system design and by employing appropriate fault-avoidance approaches in the design process. Adding dependability later on could be both expensive and might be not as effective as designing it in from the beginning.

Issues in fault-avoidance research are inseparable from considerations of fault-tolerance research. Primary objective of fault avoidance is to limit introduction of faults during system construction. In other words, *Fault avoidance* technique tries to reduce the probability of fault occurrence, while *fault tolerance* technique tries to keep the system operational despite the presence of faults. Because complete fault avoidance or elimination is not possible, a critical system always employs fault tolerance techniques to guarantee high system reliability and availability as fault tolerance tries to compensate for, and to protect against, the impacts of faults during system operation. Though software does not deteriorate (by itself) with use but often much more complex than hardware counter parts and at the same time, it is virtually impossible to design fault free software.

We may think it as *banana software approach*, which ripens at the customer. For real time system, software fault avoidance is not an option. We can improve software by rigorous (if not formal) specification of requirements and by using proven design methodologies along with the use of languages with data abstraction & modularity. At the same time, we must use software engineering environments in order to manage complexity.

Software fault tolerance methods include: exception handling, watchdog timers, assertions, acceptability checks, reasonableness checks, design diversity, and data diversity. Researchers agree that all software faults are design faults. Fault elimination and fault prevention are parts of fault avoidance. *Fault forecasting* includes a set of methods and techniques that intend to estimate the presence, the creation, and the consequences of faults. *Fault prevention* attempts to eliminate any possibility of faults creeping into a system before it goes operational. Fault removal aims to find and remove the causes of errors. *Fault prevention* can be attained by quality control techniques employed during the design and manufacturing of hardware and software. They include structured programming, information hiding, modularization, etc., for software, and rigorous design rules for hardware. Shielding, radiation hardening etc, are useful to prevent operational physical faults. Training, rigorous procedures for maintenance, "foolproof" packages prevent interaction faults. *Firewalls* and similar defenses prevent malicious faults.

We understand that fault avoidance, fault removal and fault tolerance represent three successive lines of defense against the contingency of faults in software systems and their impact on system reliability.

Software Fault Avoidance Rules: The following software fault avoidance rules, as suggested by Lyu, should be followed regardless of the type of installed software- structure: All requirements should be specified and analyzed with

formal methods, Specification- document should be debugged and stabilized before the development of any components (for example by developing final code prototypes), A protocol should exist in order to know and solve the problems. This protocol should contain measures ensuring independence in development and should not introduce correlated faults such as, e.g., communication errors, common lack of knowledge, or exchanges of erroneous information among the various development teams, All the verification, validation and the test (VVT) should be formalized and should show absence of correlated faults, and All the specifications, design and the code should be tested thoroughly. Lambert et al, [1993] points out, "In practice, the software development process is error prone, i.e., software fault avoidance and software fault removal methods are far from perfect. Development faults can be avoided using formal methods (particularly methods with a mathematical basis) during the various phases of the *software* life cycle. However, the application of mathematical methods for a complete operational telecommunication system is not feasible within the next five to ten years. It is expected however that most faults can be avoided using specification languages and modeling and simulation during the requirements and specification phases.

In spite of all formal specifications, testing and verification techniques of fault avoidance approaches, we often observe that a system fails when hardware components fail or environment changes or because of latent defects. In order to avoid faults caused by environment changes or to avoid failure due to latent defects, we need to employ robust design concepts along with the fault avoidance methods while designing a dependable software system. We find that fault avoidance approaches rarely treat various environmental and other faults. It is also true that design for reliability is rarely taught to Computer Science majors. Bernstein points out, "Software faults are common for the simple reason that the complexity in modern systems is often pushed into the software part of the system. Then the software is pushed to and beyond its limits. It is estimated that 60-90% of current computer errors are from software faults. [Gray91] Software faults may also be triggered from hardware; these faults are usually transitory in nature, and can be masked using a combination of current software and hardware fault tolerance techniques."

As software fault tolerance is often measured in terms of system availability, which is a function of reliability, we should include various single version (SV) software- based approaches of fault tolerance for more effective software fault avoidance in order to combat latent defects, environment and operational faults for attaining higher system dependability. Software based approaches often rely on either static redundancy or dynamic redundancy. By static redundancy, we mean the redundancy inside a system for hiding effects of faults whereas, by dynamic redundancy, we mean the redundancy supplied inside a module that catches erroneous output or that provides an error detection facility along with possibly an error recovery module. We also need voting modules. Voting is the process to merge the outputs of redundant modules. Masking redundancy is also useful for masking errors from application software. In masking redundancy, a few processors run the same program and vote to identify errors in any single processor. No software rollback is needed here to fix errors. We might use software implemented fault tolerance (SIFT) or an approach to fault tolerant multi-processor. Again, the algorithm-based fault tolerance (ABFT) approach that refers to a self-contained method for detecting, locating, and correcting

errors with a software procedure, is also useful. The single version software-based approaches include software implemented control flow error checking, error masking, fault recovery, error detection and correction and so on by using necessary replicated data or code, assertions, time or space redundancy etc. Such techniques normally rely on enhanced single version programming (ESVP) schemes that are based on single robust design only. *ESVP* is a low-cost solution.

Whereas, an N-version programming (NVP) scheme that relies on design diversity is suitable for tolerating software design bugs. For higher system dependability, we might go for a hybrid approach that relies on both the NVP and ESVP approaches. In this hybrid approach, each software version of an NVP application is based on an appropriate single version programming (SVP) or ESVP scheme. Such hybrid software design approach would be a useful tool toward better software fault avoidance and this technique aims designing a system with high reliability.

Further Reading:

- Goutam Kumar Saha, "Software Based Fault Tolerance – a Survey," ACM Ubiquity, Vol. 7(25), July, 2006, ACM Press, USA.
  URL: http://www.acm.org/ubiquity/views/v7i25_survey.html
- Goutam Kumar Saha, "Software Fault Avoidance Issues," ACM Ubiquity, Vol. 7(46), November 2006, ACM Press, USA.
- Goutam Kumar Saha, "Control Flow Check – Based Fault Tolerant Computing," International Journal of Computing and Information Technology, Vol.3(1), 2011.
- Reliability, Maintainability, and Availability (RMA) Handbook, FAA-HDBK-006, 2006.
- Charles B. Weinstock, David P. Gluch, "A Perspective on the State of Research in Fault -Tolerant Systems," June 1997, CMU/SEI-97-SR-008.
- Philippe Charpentier, "Final Report of WP 1.2," INRS, European Project STSARCES Contract SMT 4CT97-2191.
- Jie Xu, Brian Randell, "Software Fault Tolerance: *t* / *(n* - 1)-Variant Programming," IEEE Transactions on Reliability, Vol. 46, No. 1, 1997 March.
- J. P. Bowen, V. Stavridou, "Formal Methods and Software Safety," 1992.
- Allan M. Stavely, Toward Zero-Defect Programming, Addison Wesley Longman, 1999.
- Michael Huth and Mark Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press, 2000.
- P.P. Shirvani, E.J. McCluskey, "Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project," CRCTR 98-2, Stanford University, Stanford, CA, Dec. 1998.
- John C. Knight, "Software Fault," (Lecture series), 2004.
- R.V. Hamxleden, "Achieving Dependability," (Lecture series on Distributed Real – Time Systems), 2001.
- Mili, B. Cukic, T. Xia, R. Ben Ayed, "Combining Fault Avoidance, Fault Removal and Fault Tolerance: An Integrated Model."
  URL: http://ieeexplore.ieee.org/iel5/6516/17400/00802168.pdf
- Lambert J.M. Nieuwenhuis, Howard Sewberath Misser, Ian Hawker, Stephen S. Donachie, Mauro Ravera and Stefan Balzaretti, "Reliability Engineering for Future Telecommunication Networks and Services," IEEE, 1993.
  URL: ieeexplore.ieee.org/iel2/1047/7666/00318170.pdf?arnumber=318170
- Algirdas Aviziienis, Jean-Claude Laprie and Brian Randell, "Fundamental Concepts of Dependability," Research Report N01145, LAAS-CNRS, April 2001.

- Larry Bernstein, "Software Fault Tolerance Forestalls Crashes: to Err is Human; to Forgive is Fault Tolerant," Advances in Computers, Vol. 58, Highly Dependable Software, edited by Marvin Zelkowitz, Academic Press, ISBN 0-012-012158-1, pp. 240- 285, 2003.
- M. R. Lyu, Software Fault Tolerance, Chichester, England: John Wiley and Sons, Inc., 1995.
- M. R. Lyu, Handbook of Software Fault Tolerance, ISBN 0-471-93784-3, 2000.
- E. M. Gray and R. H. Thayer, "Requirements in Aerospace Software Engineering, A Collection of Concepts," Ed. C. Anderson and M. Dorfman. Washington: AIAA, 1991.
- J.C. Laprie, "Dependable computing: concepts, limits, challenges," In Proc. 25th IEEE Int. Symp. Fault-Tolerant Computing - Special Issue, Pasadena, CA, 1995.
- Goutam Kumar Saha, "Software Implemented Fault Tolerance Through Data Error Recovery," ACM Ubiquity, Vol. 6(35), September 2005, ACM Press, USA. URL: http://www.acm.org/ubiquity/views/v6i35_kumar.html
- Goutam Kumar Saha, "Software Based Fault Tolerant Computing," ACM Ubiquity, Vol. 6(40), November 2005, ACM Press, USA.
  URL: http://www.acm.org/ubiquity/views/v6i40_saha.html
- Goutam Kumar Saha, "Low-Cost, Fault Tolerance Applications," IEEE Potentials, Vol. 24(4), pp.35-39, 2005, IEEE Press, USA.
- Goutam Kumar Saha, "Software Based Fault Tolerant Array," IEEE Potentials, Vol. 25(1), pp.41-45, Jan-Feb, 2006, IEEE Press, USA.
- Goutam Kumar Saha, "Transient Fault Tolerance Through Algorithms," IEEE Potentials, Vol. 25(5), pp. 25-30, Sep-Oct 2006, IEEE Press, USA.
- Goutam Kumar Saha, "Software Implemented Fault Tolerance - The ESVP Approach," ACM Ubiquity, Vol. 7 (31), August 2006, ACM Press, USA. URL: http://www.acm.org/ubiquity/views/pf/v7i31_esvp.pdf
- R.V. Hanxleden, "WS 2001/02 – Distributed Real-Time Systems- Approaches to Achieving Reliable Systems," Lecture-15, 2001.
- F. R. Harnden, F.A. Primini, H.E. Payne eds. "Software Fault Tolerance for Low – to – Moderate Radiation Environments," Proc. Astronomical Data Analysis Software and Systems, Vol.238, 2001.