Software Tech News

/awareness/newsletters/listing.shtml/awareness/newsletters/listing.shtml

How Reliable are Requirements for Reliable Software?

by Herbert Hecht and Myron Hecht, SoHaR Inc.

Introduction

Missing, inaccurate or incomplete requirements lead to errors in software development and usually also prevent these errors from being detected during the testing phase. Functional testing is based on the requirements; a missing or inaccurate one will not be detected. Structural testing is based on the developed code; an unstated requirement is unlikely to be implemented and will not be detected. Operational failures due to omissions or inaccuracies cause major economic losses or even casualties, and corrective measures are far more costly than they would be if the defect had been caught earlier. A distinguishing feature of *reliable software* is that it contains fault tolerance provisions, such as alternative exits when the assertions fail, roll-back and re-try, recovery blocks, or multi-version programming. In most cases these provisions prevent or attenuate the effect of hardware and software failures that would have occurred in their absence, but there have also been incidents where the fault tolerance objectives have not been achieved and the reasons for the failure have usually included missing or ill-formulated requirements.

In the body of this paper we first describe *what* is missing in requirements, then why it is missing, and after that we explore corrective measures and test strategies for verification of reliable software.

What is Missing in Requirements for Reliable Software ?

Difficulties in formulating requirements for reliable software frequently arise from inability to identify

. all sequences that invoke fault tolerance provisions and

a. future operational environments. We discuss these in turn.

An analysis of failures in a telephone switching system paper notes that

. the largest cause category (44% of failures) comprised combination hardware/software faults. In most cases it was the inability of the software to recover from hardware faults that it was intended to protect against, and

a. that the faults leading to the most severe consequences "were introduced during the specification period and are therefore difficult to solve." [1] Similarly, a GAO report on serious problems in ten computer-based systems traces these to failure to implement "a process for disciplined, consistent procedures for software requirements management, quality assurance, configuration management, and project tracking. "[2] Requirements management is the key since all the other functions depend on it.

The reasons for the deficiencies in requirements include disbelief that more than one failure can occur during an operating interval, or neglecting the possibility that a single event (e. g., a short power interruption) can trigger several fault responses in the system is frequently overlooked. Even where requirements for fault tolerance provisions are explicit, the designers may misinterpret them unless a specific review or consultation process is provided. This is seen in an experiment sponsored by NASA to investigate the independence of fault responses in redundant software.[3] The specifications for the program were very carefully prepared and then independently validated to avoid introduction of common causes of failure. Each programming team submitted their program only after they had tested it and were satisfied that it was correct. Then all 20 versions were subjected to an intensive third party test program. The objective of the individual programs was to furnish an orthogonal acceleration vector from the output of a non-orthogonal array of six accelerometers after up to three arbitrary accelerometers had failed. Table 1 shows the results of the third party test runs in which an accelerometer failure was simulated.

Table 1. Tests of Redundancy Management Software

No. of Prior Anomalies	Observed Failures	Total Tests	Failure Fraction
0	1,268	134,135	0.01
1	12,921	101,151	0.13
2	83,022	143,509	0.58

The number of rare conditions (anomalies) responsible for failure is one more than the entry in the first column (because a new accelerometer anomaly was simulated during the test run, and it is assumed that the software failure occurred in response to the new anomaly). In slightly over 99% of all tests a single rare event (accelerometer anomaly) could be handled as indicated by the first row of the table. Two rare events produced an increase in the failure fraction by more than a factor of ten, and the majority of test cases involving three rare events resulted in failure. Although the statement of the problem clearly required that up to three anomalies had to be tolerated, the software developers had difficulties in providing for the required response to more than a single malfunction. Also, the developers' own test scenarios did not sufficiently explore multiple failure conditions.

The second difficult area for requirements is the response to changes in the system environment. Computers and operating systems are periodically updated and new models of sensors or actuators may be introduced. The application program may be reviewed and tested for proper operation in the new environment, but safeguards to prevent use of the software in the wrong configuration are frequently missing. Thus, if a problem develops with the most recent release of the operating system and it is decided to revert to the previous one, the need to go back to the old application software may be overlooked. Several crashes of important programs have been attributed to such lapses in configuration management. Providing a version check as part of the initialization should be a mandatory requirement but apparently it is not.

Why Requirements are Incomplete

The primary cause of incomplete requirements is the waterfall model that assumes that requirements can be completely formulated at the outset for systems of any scale. That, coupled with a procurement system that discourages continuous updating of user needs, casts in concrete requirements that were developed under severe time constraints and many months, possibly years, before the development started.

In a large organization, and particularly in branches of the government, at least three entities participate in the formulation of requirements: the user, the funding agency, and the office in charge of the development. The first step in the process is a statement of operational needs generated by the user. This is typically forwarded to the developer for obtaining a budgetary estimate, and then the need and the estimate are submitted for funding. In favorable circumstances the funding will be approved, but usually after considerable delay. Once approval has been obtained, the emphasis is on avoiding further delay. Previously generated requirements are dusted off and only cursorily reviewed to determine that they really represent current needs.

Finally, we want to reiterate the difficulty of conceptualizing and understanding the effect of multiple failures that was already mentioned in the preceding section. The resource-constrained environment of a typical software development provides a further obstacle to evaluating whether the requirements fully cover all required combinations of failures.

Corrective Measures

In the two preceding sections we have seen that requirements for highly reliable systems may be incomplete, particularly with regard to the reliability related features. Missing or incomplete requirements are not likely to be identified by either functional or structural testing and thus tend to persist into the OPEVAL and usage phases, sometimes constituting safety hazards and always imposing a very high cost for correction in the late lifecycle phases.

Since we have identified the waterfall model as a root cause of incomplete requirements it is appropriate to mention techniques that recognize that requirements evolve during development. Among these are the spiral development model[4] and rapid prototyping.[5] Narrower techniques are summarized in Table 2.

As a baseline (against which corrective measures will be evaluated) let us assume that the software development proceeds in a disciplined manner, and that applicable techniques from the requirements engineering discipline have been used.[6]

Table 2. Techniques for Avoidance of Incomplete Requirements

Technique	Benefits
Formal Methods [7]	Can detect some inconsistencies and instances of incomplete requirements
Condition Tables [8]	Very effective detection of incomplete requirements
Scenario-Based Testing [9] Thread-Based Testing [10] Task-Based Testing [11]	All of these elements introduced into earlier test phases, effectiveness depends on the skill of the implementers
Random Testing [12]	Multiple RN generators for groupings of exception conditions can detect missing requirements for combination events.

The first two entries in the above table address primarily logical gaps or inconsistencies. The three test methods that are grouped together in the next row go beyond the traditional requirements format and recognize the need for more user interaction with the development. Random testing has been shown to provide high coverage in the cited reference, but it needs an oracle to identify the correct test outcome where that is not obvious.

While user involvement during development will help, the typical task-oriented user does not recognize deficiencies in exception handling or the need for automated configuration monitoring. *Requirements elicitation* improves the effectiveness of user interaction but must be directed to areas where deficiencies are likely to exist. This requires knowledge of past failures and better utilization of existing databases for identifying the role of incomplete requirements. Thus collection and analysis of failure data emerges as the key to long term improvements for formulation of reliable requirements for reliable systems.

About the Authors	Author Contact Information
Herbert Hecht founded SoHaR in 1978 and is currently Chairman of the Board. Previously he held engineering management positions at The Aerospace Corporation and at Honeywell Flight Systems. His chief professional interest is the reliability and availability of computer based systems. He has served as a Governor of the IEEE Computer Society and as a visitor in Computer Engineering for ABET. Recently he has been on the National Research Council Committee that evaluated long term use of the International Space Station. He earned BEE and MEE degrees from City College and Polytechnic University of New York, respectively, and received a Ph. D. in Engineering from UCLA.	Herbert Hecht Chairman of the Board SoHaR Incorporated 8421 Wilshire Blvd. #201 Beverly Hills CA 90211 Voice: (323) 653-4717 x110 Fax: (323) 653-3624 herb@sohar.com www.sohar.com
Myron Hecht is co-founder and President of SoHaR Incorporated. His activities in basic research and development at SoHaR have resulted in new architectures for real time distributed systems, methodologies for the development and verification of fault tolerant software, and design techniques for highly reliable distributed systems for process control and C3I. In prior employment he developed and verified computer codes for nuclear power plants at SAIC and Westinghouse. He has an M.B.A, an M.S. in Nuclear Engineering, and a B.S. in Chemistry, all from UCLA. He is a member of the IEEE and has served its standards committees. He has authored or co-authored more than 60 refereed publications in the fields of software quality and metrics, computer dependability, maintenance resource allocation, air traffic control, and nuclear engineering.	Myron Hecht President SoHaR Incorporated 8421 Wilshire Blvd. #201 Beverly Hills CA 90211 Voice: (323) 653-4717 Fax: (323) 653-3624 myron@sohar.com www.sohar.com

References

1. K. Kanoun and T. Sabourin, "Software Dependability of a Telephone Switching System", *Digest, Fault Tolerant Computing Symposium-17*, pp. 236-241, Pittsburgh, Pa., June 1987
2. General Accounting Office, High Risk Series: Information Management and Technology, GAO/HR97-9, Feb 97
3. D. E. Eckhardt, A. K. Caglayan, J. C. Knight, et al., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability", *IEEE Trans. Software Engineering*, vol 17 no 7, July 1991, pp. 692 - 702
4. B. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, May 1988, p.61
5. R. Balzer, N. Goldman and D. Wile, "Operational Specification as the Basis for Rapid Prototyping", *ACM Software Engineering Notes*, Dec 82, pp. 3 - 16
6. Mylopoulos, J (ed.), *Requirements Engineering*, IEEE Computer Society, 1997
7. Susan Gerhart, Dan Craigen and Ted Ralston, "Observations on Industrial Practice Using Formal Methods", *Proc. 15th International Conference on Software Engineering*, IEEE Computer Society Press, Baltimore, May 1993, pp. 24 - 33
8. D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of Safety-Critical Software", *Proc. Ninth Annual Software Reliability Symposium*, Colorado Springs CO, May 1991

9. Jarke, M., and R. Kurki-Suonio, eds. Special Issue on Scenario Management, *IEEE Transactions on Software Engineering*, vol 24 no 12, December 1998

10. Borgia, W. M., and N. J. Hrdlick,, "Thread-Based Integration Testing", *Software Tech News*, vol 3 no 3, (DACS), January 2000

11. Telford, D. G., "Task-Based Software Testing", *Software Tech News*, vol 3 no 3, (DACS), January 2000

12. P. G. Bishop, ed., *Dependability of Critical Computer Systems 3 - Techniques Directory*, Elsevier Applied Science, ISBN 1-85166-544-7, 1990