HOW RELIABLE ARE REQUIREMENTS FOR RELIABLE SOFTWARE?

HERBERT HECHT MYRON HECHT
SoHaR Incorporated
Beverly Hills, California

Track 3, Thursday, May 4

Key Words: Requirements, Waterfall Model, Reliable Software, Failure Data Analysis, Software Failures

Abstract

Incomplete requirements lead to errors in software development and also prevent these errors from being detected during the testing phase. The resulting operational failures may cause major economic losses or even casualties, and correction is far more costly than if the defect had been caught earlier. We describe *what* is missing in requirements, then *why* it is missing, and then explore remedial measures and test strategies for verification of reliable software. Failure data analysis is recommended to facilitate better elicitation of requirements.

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 therefore not be detected. Structural testing is based on the developed code; an unstated requirement is unlikely to be implemented and will therefore not be detected. Operational readiness tests (OPEVAL in DoD) sometimes detect the omissions or inaccuracies, but more frequently it is only through failures in actual operation that these defects are made manifest. At that stage the failure will have caused major economic losses or even casualties, and corrective measures are far more costly than they would be if the defect had been caught earlier. Thus there is much motivation for getting the requirements right and complete. We are not the first ones to present this insight, but we hope to shed some new light on the difficulties in achieving complete and accurate requirements, particularly for *reliable software* which we use as a shorthand for "software for systems that must meet stringent reliability requirements"

A distinguishing feature of reliable software, as identified above, 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 remedial measures and test strategies for verification of reliable software.

What is Missing in Requirements for Reliable Software?

Specific difficulties in formulating requirements for reliable software arise from inability to identify (a) all sequences that invoke fault tolerance provisions and (b) future operational environments. We discuss these in turn.

That fault tolerance code (also called *defensive code*) contributed much more than it proportionate share to operational failures in a telephone switching system was documented almost 15 years ago[1]. [KANO87] The paper notes that (a) the largest cause category (44% of failures) comprised combination hardware/software faults, in most cases inability of the software to recover from hardware faults that it was intended to protect against, and (b) that the faults leading to the most severe consequences "were introduced during the specification period and are therefore difficult to solve." The requirements for the defensive code did not completely and accurately identify the fault conditions that were to be defended against.

A GAO report on serious problems in ten computer-based systems traces these to failure to implement "a process for selecting … 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, and consequent omission of a requirement for coverage of multiple failures
- recognition that the occurrence of more than one failure is possible, but considering it so unlikely that only token requirements for dealing with this condition are provided
- overlooking the possibility that a single event (e. g., a short power interruption) can trigger several fault responses in the system, and making no provision for sequencing of these.

A typical example is the requirement for a feedwater control for a steam turbine. The feedwater pumps are started when the water level drops below a set point. For reliability reasons triplicated water level sensors were specified. If the sensors disagree by more than a threshold, an algorithm determines which sensor is to be disregarded. After a sensor has been disregarded for a number of readings it is declared failed. The requirement does not consider what happens beyond that point.

Because the level sensors cannot be replaced while the plant is operating, failure of a single sensor forces an almost immediate shutdown for replacement. An extension of the algorithm to deal with a second failure would in most cases have permitted the replacement to be accomplished during a scheduled maintenance period.

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.] Twenty versions of a redundancy management program, written in Pascal, were developed at four universities (five versions at each) from the same requirements, and the versions were then tested individually to establish the probability of correlated errors that would defeat the benefits of N-version fault tolerant software. 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. The software failure statistics presented below were computed from Table 1 of the reference.

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.

We see that one weakness in the requirements for reliable software tends to be associated with the reliability and fault tolerance provisions, and that the response to multiple failures is a particularly suspect area. The second troublesome area 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 a paradigm of system development that might have been valid for single processor control systems and for batch data processing but that is unrealistic for today's distributed and real-time environments. The waterfall model 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 and feedback of development costs 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. Although the requirements generated at this time are expected to be refined by a contractor, they usually contain key reliability and fault tolerance requirements. Typical clauses are: "no single failure shall prevent ….", "the maintenance free period shall be at least ….", or "the availability of the service shall be at least ….." These statements are easily interpreted as representing the full reliability requirements. Elaboration at that level may be felt unnecessary or even inadmissible. Thus, the implementation only addresses the minimum interpretation of the top-level requirements and this leads to the deficiencies described in the previous section.

Another reason for incomplete requirements is the difficulty of measuring the effect of design decisions on reliability. Whereas models and simulators are widely employed for evaluating the effects of changes in requirements on performance, there is very little use of similar tools for continuous reliability or availability evaluation. Thus, even where the original requirements for reliability features were complete, there is a possibility that they may be incomplete or inaccurate as a result of changes made during the development.

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 there is considerable theoretical and empirical evidence 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]. The latter provides benefits in some applications of modest complexity[6] but falls short of being a systematic way of dealing with incomplete requirements.

Thus, there is ample motivation to correct these deficiencies, and we mention now a number of techniques that are claimed to be helpful. It is assumed that the software development proceeds in a disciplined manner, and that applicable techniques from the requirements engineering discipline have been used[7] Thus, outright failure to implement a requirement, or omission of a requirement essential for an implemented function, is ruled out.

Some techniques that go beyond this baseline are summarized in Table 2. But none of these represents a complete solution by itself, and it is not known to what extent even a combination of the techniques will provide a really significant improvement.

Table 2. Techniques for Avoidance of Incomplete Requirements

TECHNIQUE	BENEFITS	LIMITATIONS
Formal Methods[8]	Can detect some inconsistencies and instances of incomplete requirements	Unlikely to detect a completely missing requirement, e. g., to enforce configuration control
Condition Tables[9]	Very effective detection of incomplete requirements	Same as above
Scenario-Based Testing[10] Thread-Based Testing[11] Task-Based Testing[12]	All of these introduce elements of OPEVAL into the earlier test phases. Effectiveness depends on skill of the implementers.	No assurance of detecting completely missing requirements
Random Testing[13]	Multiple RN generators for groupings of exception conditions can detect missing requirements for combination events.	Difficult to establish effectiveness of testing. Unlikely to detect completely missing requirements

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 it is encouraging to see that the need for more user interaction with the development is being recognized. That requirements are stated at the beginning of a project and that the developer then simply turns the crank to implement them has always been a myth, and all steps to overcome that misconception will be beneficial. 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.

The most promising approaches appear to be those that increase user involvement during development but that by itself will not necessarily remove all the deficiencies described in the preceding sections. The typical task-oriented user will usually not recognize deficiencies in exception handling or the need for automated configuration monitoring. Research in *requirements elicitation* aims to improve the effectiveness of user interaction in the requirements

formulation. But in order for the systems or software engineer to elicit complete requirements, he or she has to know in what areas 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.

Conclusions

Systems that are essential for our daily activities, and particularly for defense, require dependable operation of computers and software. These systems are complex, typically involving multiple computers and programs. Development practices based on the waterfall model that assumes complete requirements are stated at the beginning, with further steps only representing increasingly detailed implementation, are not adequate for the environment in which these systems are developed and used.

We have presented a number of techniques that can improve communication between the user and the developer, and that can partly overcome the limitations of the conventional requirements formulation. But we must be mindful that non-functional requirements (that may be essential for reliability) may not surface unless they are specifically elicited. What must be elicited? The subjects most likely to cause failures. At present we can only guess at what they are, but with better data we will know.

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] V. Scott Gordon and James Bieman, "Rapid Prototyping: Lessons learned", *IEEE Software*, Jan1998, pp. 85-94

[7] Mylopoulos, J (ed.), *Requirements Engineering*, : IEEE Computer Society, 1997

[8] 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

[9] 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

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

[11] Borgia, W. M., and N. J. Hrdlick,, "Thread-Based Integration Testing*", Software Tech News,*vol 3 no 3, Data and Analysis Center for Software (DACS), January 2000

[12] Telford, D. G., "Task-Based Software Testing", *", Software Tech News,*vol 3 no 3, Data and Analysis Center for Software (DACS), January 2000

[13] P. G. Bishop, ed., *Dependability of critical computer systems 3 -- Techniques directory*, Elsevier Applied Science, ISBN 1-85166-544-7, 1990