1996 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM
JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

SOFTWARE RELIABILITY ISSUES CONCERNING LARGE AND SAFETY CRITICAL SOFTWARE SYSTEMS
A Literature Survey of Current Techniques

Dr. Khaled Kamel, Professor and Chairman
Engineering Computer Science Department
University of Louisville
Louisville, Kentucky

KSC Colleague - Barbara Brown
Computer Science

Contract Number NASA-NGT10-52605

July 19, 1996

Software Reliability Issues Concerning
Large and Safety Critical Software Systems

Khaled Kamel and Lawrence Lowe, University of Louisville

ABSTRACT

This research was undertaken to provide the National Aeronautics and Space Administration (NASA) with a survey of state-of-the-art techniques being used in industry and academia to provide safe, reliable, and maintainable software to drive large systems. Such systems need match the complexity and strict safety requirements of NASA's shuttle system. In particular, the Launch Processing System (LPS) is being considered for replacement. The LPS is responsible for monitoring and commanding the shuttle during test, repair, and launch phases. NASA built this system in the 1970's using mostly hardware techniques to provide for increased reliability, but it did so often using custom-built equipment, which has not been able to keep up with current day technologies. This report surveys the major techniques used in industry and academia to ensure reliability in large and critical computer systems.

1. Introduction

After many years of expensive and highly customized maintenance of the Shuttle's Launch Processing System, NASA has decided that the time to redevelop the LPS has come. A few factions exist within the NASA hierarchy, each with slightly different but complementary ideas on how to approach such a giant step. All the involved persons, however, share a mutual goal: to implement the new Launch Processing System taking advantage of today's well-established standards and commercial-off-the-shelf (COTS) equipment. Standards and commercial equipment are necessary to keep operation and development costs minimized. Without them, the customized path becomes increasingly more expensive with time, and obsolescence is quick to come. An added advantage is that most commercial vendors implement the latest technology in their components to provide high speed and high reliability during operation with backwards-compatible upgrades. Since commercial equipment has become widely used in a number of environments, our confidence in its ability to perform correctly is strengthened.

Regardless of how safe and reliable hardware is, it is usually software that drives it, and software is expected to play an even larger role at NASA with upcoming efforts such as Space Station and future vehicles. It is well known that the potential for software to fail is significantly higher than that of hardware. Many will attribute this software characteristic to the high number of states and transitions that a software system can reach. Thoroughly testing all such states and paths is often not a feasible task (Kim 95). The task is complicated even further with the possibility of unforeseen states. As a result, software practices have focused on detecting, tolerating, and recovering from failures related to software rather than preventing them. The key mechanism for tolerating faults is *redundancy,* and it is the primary focus of those techniques presented within this paper. Since the fault handling of both hardware and software within a system are closely related, the architectures and methods presented here often address these reliability concerns from both sides.

2. Terminology of Reliability

Software controls an increasingly large number of systems in operation throughout the world today. Many of these systems either directly or indirectly affect people, and that is why there can be no practical allowance of error. Ideally, software would be designed and implemented without any inherent flaws, but Pham suggests that "we cannot reasonably expect a large system to be error free" (Pham 92). Any person having experience in the development of a large software system knows that exhaustive testing is a difficult undertaking. Despite any amount of effort, there is a point at which further testing produces minimal improvement with exponential increases in time and cost. Thus, the "final release" of any software system is expected to require future maintenance and the uncovering of flaws missed in the development stage.

Since software cannot practically be completely error-free, there is a factor of *reliability* associated with a system. Gersting defines reliability as "the capability of a system to perform over a period of time in adherence to its specifications" This definition places a strong connection between reliability and requirements. Later sections of this paper show that imperfect requirements, and not software faults, are the cause of many system failures.

Any conceptual error in the way that a unit of software handles a state or set of inputs is referred to as a *fault.* If the state or set of inputs occurs for which the software unit is faulty, the fault will result in an *error.* This error is an internal state that does not relate to the set of allowable states. An error, then, has a potential to cause a system *failure.* The failure is an observable deviation from the specifications. This terminology is important in that software can possess faults that never manifest themselves as errors, and errors that do not trigger failures. Furthermore, we can have software that possesses a number of faults, but is still considered reliable since the faults may not result in failures.

Gersting suggests that knowledge of the *operational profile* of the software unit in question can be used to help estimate the unit's reliability (Gersting 92). The operational profile gives an indication of how often different sub-components within a unit are executed. If a sub-component sees a large amount of traffic, then any faults in that sub-component are more likely to generate failures. It follows then that a fault in a low-traffic sub-component has a lesser impact on the reliability of the unit.

Software faults can be further classified according to *Independence* and *Persistence* (Laprie 92). Under the Independence class, a fault is either *related* or *independent*. Related faults often result from a general misunderstanding of the specifications or from a specification fault. When one fault is encountered, its related faults have a high chance of creating similar errors. This set of similar errors can lead to *common-mode failures*, the set of failures attributable to the same fault. Faults that are independent, however, have no relation and usually do not produce failures under common conditions.

The Persistence class separates faults as either being *solid* or *soft*. A soft fault is a fault synonymous with an intermittent error when discussing hardware faults. That is, once having occurred, it is not likely that the fault conditions will be reproduced. Furthermore, a fault is soft only if it is recoverable. A solid fault is just the complement of the soft fault. It either continues to occur under normal operations or is unrecoverable. This classification of independence and persistence will play a large role when discussing fault-tolerant architectures later in this paper.

3. Redundancy Techniques

Assuming that all software will contain faults, the developer is left with the task of finding a way for software to operate reliably in the presence of such faults. Pham suggests that, "to achieve ultrareliability in computing, it is necessary to adopt the strategy of defensive programming based on redundancy" (Pham 92). Redundancy suggests that a collection of programs, referred to as versions, performing the same function be used in place of a single program. In theory, any faults will be masked by the collection of programs (i.e., the collection will often succeed as a whole where a single program might fail). This technique, allowing faults to exist while still operating reliably, is referred to as *fault-tolerance*. Indeed, fault-tolerance has become the key focus in industry and academia in providing for reliable software systems. The two techniques discussed below are the foremost in implementation and are the main focus of this paper. There are issues, however, that foil the theory. The theory of fault-tolerance depends on faults being independent so that the majority of the collection's versions agree on the correct output. If faults are related, a common-mode failure might cause the majority of the collection to rule in favor of an incorrect output or produce no majority at all. Experiments have shown that, even in the presence of related faults, these redundant systems still give some improvement over their nonredundant counterparts. The results that they give make it clear that redundancy is worth further exploration.

3.1. Recovery Block Method

Using the recovery block approach, as well as other methods, the designer wishes for a system to perform a task or calculation with a high degree of confidence in its results. Under the recovery block method, a system is comprised of three components: (1) a primary module responsible for executing critical software functions and the desired task or calculation, (2) an acceptance test by which the primary module's output can be verified, and (3) a set of alternate modules, each capable of performing the same function as the primary module (Pham 92). According to Stark and Dugan, when the recovery block is entered, a checkpoint is created so that the system's state can be restored. The primary module is then executed to perform the desired task, and the results are verified by the acceptance test. If the acceptance test detects erroneous output, the system is rolled back to the checkpoint, and an alternate module is used to perform the operation. Alternates will continue to be tried by the recovery block until either a module passes the acceptance test or all alternates have been tried. A system failure occurs only in the latter case.

The recovery block concept is very similar to the hardware redundancy technique known as *standby sparing*. The

Recovery Block Method

```
Ensure T
     By $M_1$
     Else by $M_2$
       Else by $M_3$
           •
           •
           •
         Else by $M_n$
Else Error
```

Where T denotes the acceptance test, M_1 denotes the primary module, and M_k, $2 \leq k \leq n$, denotes the alternates.

gain in reliability with software redundancy, however, is not as large as with redundant hardware (Stark 94). In addition to the problems of correlated faults already discussed, the redundancy management itself introduces opportunities for further modes of failure.

Gersting notes that a key advantage to the recovery block strategy is that, if the primary module is determined to be successful, no additional resources or time are required to process the alternate modules (Gersting 92). In a sense, the system has performed its function with a near-minimal amount of work. A well designed and implemented system should produce correct results far more often than incorrect ones, so it follows that a recovery block should not seriously degrade average system performance.

A disadvantage to the recovery block method is also observed by Gersting. He states that the overhead in execution time required to perform a system rollback after a failed version can adversely affect system performance in the event of a failure . This is especially true in real-time environments.

The acceptance test is of special concern if the recovery block method is to provide reliable output. As stated earlier, a module will be considered successful and be used to provide the system's output if the module's output passes the acceptance test. This places a heavy responsibility on the acceptance test to accurately decide when an output is or is not a reasonable output. Because knowledge of the exact result expected does not exist, the acceptance test is restricted to items such as limit-checking and past knowledge when making its decisions. If the acceptance test is not well designed, it may pass results that are erroneous. Although this is not flagged as a failure by the system, it is clearly a deviation from the desired mode of operation. At the other extreme, a faulty acceptance test might reject a valid output.

It can be concluded then that a few things are necessary for the recovery block method to be a viable improvement on a system. First, it is desirable that the primary and alternate modules contain as few faults as possible. At a minimum, we would like the existing faults to be independent among versions. Second, the acceptance test must accurately decide the correctness of a module's output. If we are to tolerate faults, we must first detect them correctly. Regardless of the difficulty in meeting these requirements in practice, this technique has been demonstrated to give some improvement in reliability.

3.2. N-Version Programming Method

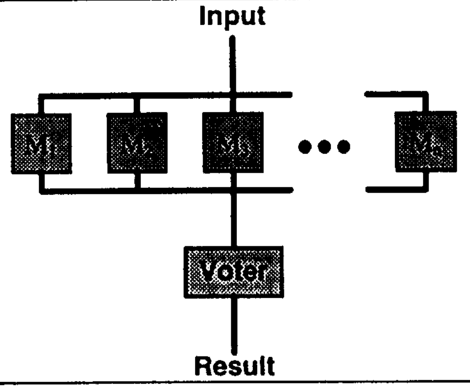
Under N-version programming, N independent programs are executed in parallel on identical input. The system determines results by voting on the outputs provided by the set of programs (Pham 92). With basic N-version programming, this vote is often just the majority agreement.

It is necessary that the N versions involved be as independent as possible. By independent it is meant that the faults contained within one version be independent of those contained within another. One cannot purposely select the faults that will exist within a version, so it is suggested that independence might be attained through diverse development environments. Each version should be implemented using different algorithms, programming languages, staff, and tools . The less in common between versions, the higher the chance of independence. It is important, however, that all versions use the exact same set of requirements and that the requirements be as clear and complete as possible.

The N-version programming method is very similar to the hardware redundancy technique known as N-modular redundancy. As with recovery blocks, the redundant software technique does not provide as significant of a gain in reliability as achieved with hardware. This is true mostly because the N versions often cannot be as thoroughly tested as hardware units. Hardware provides a fixed mapping from a set of inputs to a set of outputs, while software tends to take unanticipated turns during execution. This is not a flaw in software, but in the human's inability to foresee all possibilities.

A strong advantage of N-version programming is that when a version fails, no additional time is required to rollback

N-Version Programming Method



Where M_k, $1 \leq k \leq n$, denotes the set of versions to be executed in parallel. The voter may use any voting logic.

the system and reperform the computation as with recovery blocks . The results will be made available as soon as the outputs from all of the versions have been voted on. This means that the same amount of time is required to "reliably" perform a calculation whether a version fails or not.

A disadvantage is that "resources and overhead are required to execute, synchronize, and vote upon multiple versions for every computation" (Gersting 92). This means that even when all version successfully perform the desired computation, the overhead required to execute and decide each version is required. This makes basic N-version programming a poor choice for systems that are either resource-limited or time-critical.

Another disadvantage is that straight-forward comparison of version outputs is often not possible when voting . This is true in cases when outputs are floating-point types, where correct answers are expected to fall within a tolerance range. This complicates the voting logic. In fact, some studies have shown that more reliable results may be achieved by voting strategies other than a direct majority rule.

3.2.1. Voting Techniques

An important situation arises when basic N-version programming provides a set of outputs on which no correct majority exists (e.g., either N different outputs occur or the majority of the versions output an identical incorrect result). Under basic N-version programming, the system has no choice but to produce a system failure. A study done by Gersting compares different voting strategies by which the probability of success in such a situation is improved.

The computation performed in Gersting's study outputs a one-dimensional vector. Three redundant versions of the software were used, each being numbered according to decreasing confidence in reliability (i.e., the most reliable version is labeled 1, the next most reliable is labeled 2, and so on). Each of these versions produces an output vector for each iteration. The elements in a vector are fields of data that have some meaning to the recipient of the vector. The need for a composite result such as this may not always be the case. Some of the voting algorithms presented here may not apply in those cases.

3.2.2. Composite/Version

With this algorithm, a counter is maintained for each of the versions. Each field in the vector is compared among the three versions. The majority answer for a field is taken as the correct answer for that field. Those versions whose vectors agree with the chosen field result have their counters incremented. In the case where no majority exists for a field, no version is allowed to increment its counter. The overall result vector is the vector belonging to the version with the highest counter value after all fields have been voted on. In other words, a "correct" value for each field is decided using the majority for that field. The winner is the vector that most often agrees with the correct field values. Note that the result vector is always taken exactly from one of the version outputs. Had the result vector been built from each field majority, inconsistencies between fields may have been introduced.

3.2.3. Weighted Composite/Vector

This algorithm is identical to the Composite/Version algorithm except that a version's counter is not always incremented by one when a field within its output vector agrees with the majority. Instead, weights are assigned to different fields based on importance. The weight is the value added to a version's counter on successful field comparisons. Gersting investigated assigning weights based on how often a field changes. If a field's value does not change often, it received a lower weight since a version probably had a higher chance of getting it right. The end result was that a version was chosen that most often got the critical fields correct. Of course, this requires that knowledge of the application be known in order to assign meaningful weights.

3.2.4. Composite

The Composite algorithm does not require a counter for each version. Instead, the majority of each field is taken as the correct result for that field in the result vector. When a field has no majority, that field in the result vector will be filled by the field value from Version 1. This means that the result vector may very likely not match any of the version outputs. This method increases the chance of internal inconsistencies that the first two algorithms were trying to avoid.

3.2.5. History

For this algorithm, the voter maintains a count for each version that represents the number of times that a version agreed with the majority during past computations. For a computation, the algorithm selects the result vector from the version that both agrees with the majority and has the highest count.. If no majority exists, the agreeing version with the highest confidence is chosen.

3.2.6. Acceptance Test

This algorithm first attempts to produce a result vector by taking a majority vote. If such a vote fails to produce a majority, the algorithm then submits each version's output vector to an acceptance test in order of decreasing reliability. The result in such a case is the first vector to pass the acceptance test. As with the Weighted Composite/Vector algorithm, this algorithm requires knowledge of the application in order to define the acceptance test criterion.

Gersting made a few noteworthy conclusions about his experiment. First, he found each algorithm to reduce the failure rate by roughly 50%. There was little difference in the actual improvement between versions. He also found that success was a bit higher when only allowing the result vectors to be one of the versions' outputs (i.e., do not create a composite vector in a piecewise manner from the field majorities). However, there were times when such a technique provided correct results in the presence of three failed versions. The History algorithm proved to be least successful of all. The most successful algorithm was the Acceptance Test. Gersting ends with a useful rule of thumb: "if an output must be forced in the no-majority case, do almost anything reasonable - preferably simple - and you'll often be right" .

Conclusion

Large and safety critical computer systems similar to NASA LPS are required to have reliable hardware. Software that derive the hardware is continuing to grow and play major role in the system operation. This report detailed the most commonly used techniques in implementing reliability and safety requirements in software systems. The replacement of the LPS should carefully consider the overall system reliability including hardware and software during all phases of system operation, system analysis, system evaluation, final implementation, and throughout the system life cycle.

Works Cited

Babcock IV, Philip S. "The Application of Analytic Tools to the Design and Verification of Robust Systems." 1994 Proceedings Annual Reliability and Maintainability Symposium 1994.

- Dugan, Joanne Bechta, and Stacey A. Doyle. "Simple Models of Hardware and Software Fault Tolerance." 1994 Proceedings Annual Reliability and Maintainability Symposium 1994: 124-29.

Dunham, Janet R., and John L. Pierce. An Experiment In Software Reliability. NASA Contractor Report 172553. Research Triangle Park: Software Research and Development Center for Digital Systems Research, 1986.

Eagle, Kenneth H., and Ajay S. Agarwala. "Redundancy Design Philosophy for Catastrophic Loss Protection." 1992 Proceedings Annual Reliability and Maintainability Symposium 1992: 1-4.

Eckhardt Jr., Dave E., et al. An Experimental Evaluation of Software Redundancy As a Strategy for Improving Reliability. NASA Technical Memorandum 102613. Hampton: Langley Research Center, 1990.

France. North Atlantic Treaty Organization. Advisory Group for Aerospace Research and Development. Fault Tolerance Design and Redundancy Management Techniques. Lecture Series No. 109. Neuilly Sur Seine: 1980.

Gersting, Judith L., et al. "A Comparison of Voting Algorithms for N-Version Programming." In Fault-Tolerant Software Systems: Techniques and Applications. Proceedings 24th Annual Hawaii International Conference on Systems Sciences. January 1991. Los Almitos: IEEE Computer Society Press, 1992.

Kim, K. H., and Howard O. Welch. "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications." In Fault-Tolerant Software Systems: Techniques and Applications. IEEE Transactions on Computers. Vol. 38 Number 5. Los Almitos: IEEE Computer Society Press, 1992.

Lala, Jaynarayan, et al. Study of a Unified Hardware and Software Fault Tolerant Architecture. NASA Contractor Report 181759. Cambridge: The Charles Stark Draper Laboratory, Inc., 1989.

Laprie, Jean-Claude., et al. "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures." In Fault-Tolerant Software Systems: Techniques and Applications. IEEE Computer. Vol. 23 Number 7. Los Almitos: IEEE Computer Society Press, 1992.

National Aeronautics and Space Administration. Launch Processing System Description. Cape Canaveral: John F. Kennedy Space Center, 1984.

Pham, Hoang. Fault-Tolerant Software Systems: Techniques and Applications. Los Almitos: IEEE Computer Society Press, 1992.

Stark, George E. "Technologies For Improving the Dependability of Software-Intensive Systems: Review of NASA Experience." 1994 Proceedings Annual Reliability and Maintainability Symposium 1994.

Vouk, Mladen A., Amitkumar M. Paradkar, and David F. McAllister. "Modeling Execution Time of Multi-Stage N-Version Fault-Tolerant Software." In Fault-Tolerant Software Systems: Techniques and Applications. Proceedings IEEE Computer Software and Applications Conference. 1990. Los Almitos: IEEE Computer Society Press, 1992.