exida.com
excellence in dependable automation

Reliability in Control Systems Software

Dr. William M. Goble
Principal Partner
www.exida.com

Abstract

Considering the components used in the current control systems, hardware failure causes have been widely studied. Software failure causes, on the other hand, are rarely studied or understood. In the field studies that have been done, some of the rules for software failure causes have been theorized but even those are not widely known by software engineers or followed. Few practitioners know the rules of software reliability or take the time to study how to create reliable software. Why? This is in part because it appears deceptively easy to create software. Software tool manufacturers work hard to promote this.

We cannot, however, ignore the importance of software reliability. As control systems grow in functionality and complexity, we depend on an increasing amount of software. This paper addresses these issues and includes examples of software failures, the "root causes" of those failures, some rules for avoiding those causes and some guidance in evaluating software reliability in control system products.

Introduction

Powerful new tools have enabled us to develop control systems that are becoming increasingly more complex. Most of this complexity is found in the software-based control systems. Our dependence on this software is extensive and growing. Software reliability, the ability of this software to perform the expected function when needed, is essential. Yet, how often does it fail us? How often do we hear the following comments? "The company network is down." "My computer froze up again." "I didn't save my file and just blew away four hours of work." "How long has this operator station been frozen." Our experience with software is far from perfect.

Copyright exida.com L.L.C. 2000

As our dependency on our software increases, so does our incentive to develop higher levels of software reliability.

Consider the reasons why software fails in the following examples. The console of an industrial operator had been functioning normally for two years. On one of the first shifts of an operator who had just joined the staff, the console stopped updating the CRT screen and would not respond to operator commands shortly after an alarm acknowledgment. The unit was powered down and successfully restarted with no hardware failures. With over 400 units in the field with 8,000,000 operating hours, the manufacturer found it difficult to believe that a significant software fault existed in such a mature product.

An extensive testing procedure produced no further failures. A test engineer visited the site and interviewed the new operator. At this interview, the engineer noted, “this guy is very fast on the keyboard.” That piece of information allowed the problem to be traced. Further testing revealed that if an alarm acknowledgment key was struck within 32 milliseconds of the alarm silence key, a software routine would overwrite a critical area of memory and the computer would fail.

In another example, an operator requested that a data file be displayed on the CRT and the computer failed. This was not a new request as this same data file had been successfully displayed on the system numerous times before. The problem was traced to a software module that did not always append a terminating “null zero” to the end of the file character string. On most occasions the file name was stored in memory that had been cleared by zeros written into all locations. Because of this, the operation was always successful and the software fault was hidden. On the occasion that the dynamic memory allocation algorithm chose memory that had not been cleared, the system failed. This failure occurred only when the software module did not append the zero in combination with a memory allocation in an uncleared area of

Copyright exida.com L.L.C. 2000

memory.

Consider a third example where a computer stopped working after a message was received on its communication network. The message came from an incompatible operating system and while it used the correct "frame" format, the operating system contained different data formats. Because the computer did not check for compatible data format, the data bits within the frame were incorrectly interpreted. The situation caused the computer to fail within a few seconds. Many examples of software failure are documented (Ref. 1). Most seem to contain some combination of events considered unlikely, rare or even impossible.

Stress verses Strength

Reliability engineering provides us with the "stress verses strength" concept. Failures occur when some stress is greater than the corresponding strength. While this concept is taken from mechanical and civil engineering and is most frequently applied to stress as a mechanical force and strength as the physical ability of a structure to resist that force, the same concept is applicable to software reliability.

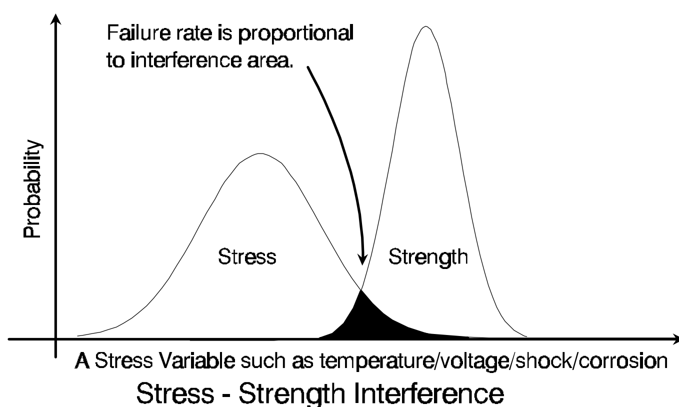

Figure 1.

Dr. A.C. Brombacher associates this concept to electronic hardware reliability. In his

Copyright exida.com L.L.C. 2000

book, “Reliability by Design,” Brombacher points out that failures occur when some stress or combination of stressors exceeds the associated strength (susceptibility) of the system (Ref. 2). See Fig. 1 for a graphic representation of this concept. Stress, or the combination of stressors is represented by a curve that shows the probability of any particular stress value. The strength curve shows the chances of any particular strength value in a collection of products. The area under both curves represents failure conditions. Within a product, strength is the measure of resistance to stress. When the product design produces higher strength levels, the product is much less likely to fail (Figure 2).

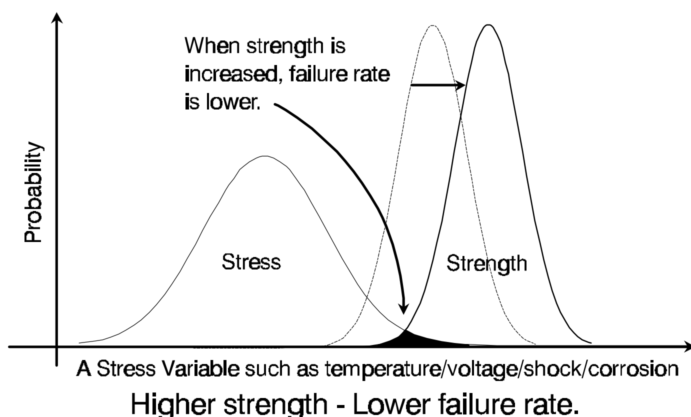

Figure 2. Failure Probability versus Stress-Strength

Electronic hardware has many potential stressors. In the environmental and physical arena, stressors include heat, humidity, chemicals, shock, vibration, electrical surge, electro-static discharge, radio waves, and others. Operational stressors include incorrect commands from an operator, incorrect maintenance procedures, bad calibration, improper grounding, and others. OK that makes sense for hardare but what are the applications for this concept with software? What are the stressors on a software system?

Just as in hardware, a software failure occurs when the stress is greater than the

Copyright exida.com L.L.C. 2000

strength. The strength of a software system can be measured by the number of software faults or design errors (bugs) present, the testability of the system, the amount of software error checking and on-line data validation. The stress of a software system is dependent on the combination of inputs, input timing and stored data seen by the CPU. Inputs and the timing of inputs may be a function of other computer systems, operators, or both.

Improving Software Strength and Reliability

Higher levels of software reliability are obtained by improving software strength. Most of the efforts in this area to date have focused on software process improvement through removing software faults. Because human beings create software and humans make mistakes, the design process is not perfect. Many company resources are expended to establish and monitor the software development process (Ref. 3,4). These efforts are intended to increase software strength by reducing the number of faults in the software. This approach can be very successful, depending on implementation. Attempts are being made to audit the effectiveness of the software process. The ISO9000-3 standard establishes required practices. The software engineering institute has created a five level software maturity model (Ref. 3). In this model, level 5 represents the best process. In addition, companies in regulated industries (e.g. pharmaceutical) audit software vendors to insure compliance to internal software standards. By reducing the number of faults in software and thereby increasing its strength, these efforts do improve software reliability.

Another factor that influences the number of faults in a software system is the testability of a system. Software testing may or may not be effective, depending on the variability of execution. A test program cannot be complete, for example, when software executes differently each time it is loaded. In this situation, the number of test cases explodes to “virtual infinity” (Chapter 7, Ref. 5). Execution variability is increased with dynamic memory allocation, number of CPU interrupts, number of tasks in a multitasking environment, etc. All of these factors need to be considered when the potential reliability of software is evaluated. Increasing the amount of variability such as multitasking,

Copyright exida.com L.L.C. 2000

dynamic memory allocation, and others, decreases testability and indicates lower strength and reliability.

Software strength also increases based on several other important factors. These factors are referred to as "software diagnostics" and increase software strength in primarily two ways. First, software diagnostics can reject potentially fatal data and second, software diagnostics can do online verification of proper software execution. In fact, international standards for safety critical software (DIN V VDE 0801, IEC61508) specify software diagnostic techniques like "program flow control" and "plausibility assertions." These diagnostics are required in safety critical software approved by recognized third parties such as FM in the United States and TUV in Germany.

One software diagnostic technique is program flow control. In this procedure, each software component that must execute in sequence writes an indicator to memory. Subsequently, each software component verifies that necessary predecessors have done their job. When a given sequence completes, verification is done to confirm that all necessary software components have run and have run in the proper sequence. When the operations are time critical, the indicators can include time stamps. The times are checked to verify that maximum times have not been exceeded.

Plausibility assertions verification is another software diagnostic technique. While program flow control verifies that software components have executed properly in the area of sequence and timing, plausibility assertions check inputs and stored data. The proper format and content of communication messages are checked before commands are executed. Data is checked to verify that it is within a reasonable range. Pointers to memory arrays must also be within a valid range for the particular array.

All of these techniques are considered online software testing. When a software

diagnostic finds something that is not within the predetermined valid range, there is usually an associated software fault. A software diagnostic reporting system records program execution data and fault data. This provides the means for software developers to identify and repair software faults rapidly. This increases the effectiveness of testing before a product is released. Assuming the data is effectively stored, these methods also allow more effective resolution of field failures when they occur.

Conclusions

Control systems depend on software and this dependency is increasing. There is an important need to evaluate software reliability but very little is now being done. The stress versus strength concept helps identify the important factors in software reliability. Depending on the required level of software reliability, the following relevant areas and questions need to be considered:

Software Process

1. Does a formal process exist for software creation?
2. Do all software developers follow this software process?
3. Has the process been audited by a third party such as FM or TUV?
4. Does the process conform to applicable standards such as ISO 9000-3, VDE0801/A1, IEC61508?

Testability

1. What execution variability factors exist in the system?
2. Does the system involve multitasking?
3. Is there fixed or dynamic memory allocation?

Software Diagnostics

Copyright exida.com L.L.C. 2000

1. Does the design include program flow control?
2. Is the program flow timed?
3. How are the communication messages checked?
4. How often is the data integrity checked?

Software reliability requires that we consider these questions carefully and frequently.

References

[1] Avizienis, A., "Systematic Design of Fault Tolerant Computers," SAFECOMP '96, Proceedings of the 15th International Conference on Computer Safety, Reliability and Security, Springer-Verlag, New York, NY, October 1996.

[2] Brombacher, A. C., Reliability by Design, John Wiley and Sons, New York, NY, 1992.

[3] Humphrey, W. S., Managing the Software Process, Addison-Wesley, Reading, MA, 1989.

[4] Nuefelder, A. M., Ensuring Software Reliability, Marcel Dekker, New York, NY, 1993.

[5] Goble, W. M., Control Systems Safety Evaluation and Reliability, ISA, Research Triangle Park, NC, 1998.

Copyright exida.com L.L.C. 2000