# Condition Based Management Augmenting Software and System Reliability

Dr Samuel J Keene, Jr, FIEEE

Introduction

Reliability can be said to be the probability of a system performing the desired task satisfactorily for a given time period while operating under a specified environmental conditions. It was thought in the 70's that software once software was tested and worked, that it would work perfectly thereafter. A reliability of "1" was assigned to the software. The thought was once software was working; it would not change and therefore would continue operating in its operating state. By the 1990's software reliability problems were considered the "long pole in the test", or the main driver of software problems. The importance of software in the total system reliability had changed 180 degrees.

The author was part of a broadcast panel on "Software Safety". One important part of that broadcast was that safety is not a characteristic parameter of software. It is a system characteristic. What I believe was meant here is that software does not operate alone: the hardware operationalizes the software. Therein the safety exposure can materialize and be brought to bear on the hardware, person in the loop, or system mission. There are many variables acting on the system and the apparent software reliability. Dr Jeffrey Voas has depicted the nesting and interaction of software and system components. See below in Figure 1:

**Figure 1: The System Makeup**

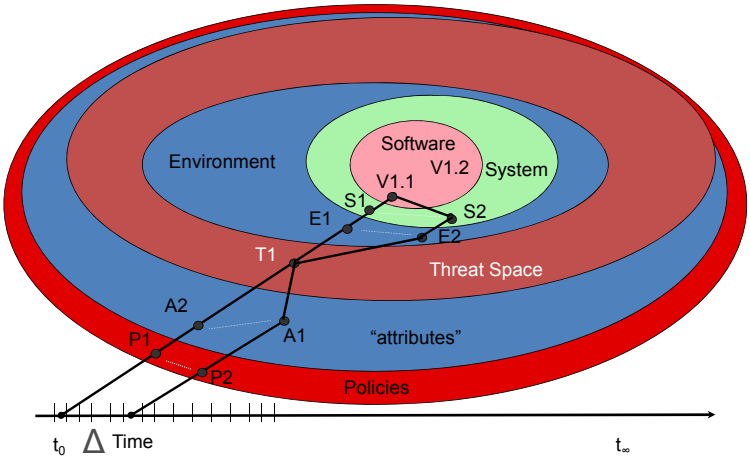



Figure 1 shows the "software" brain at the center. It is only limited by the programmer requirements mapped into it (requirements are a major challenge for reliability). The "software" is the Application Code embedded on top of the Operating System. Typically the software is evolving by going through version and point releases. These new releases fix function, safety, and reliability exposures. This "software" core then resides on hardware components. The hardware is also subject to updates over

time. The “environment” contains weather related components, as the job flow, internal components status. This is all the things that the “software” must handle besides doing its main functional tasks.

Note the world of systems and software is dynamic and ever changing. The particular challenge is to assure the system (and software) behave appropriately across all system and environment variations. In the end, it should be noted that it is usually the software which is changed to adapt to unwanted variations through other parts of the system or the environment. This means that software often gets the rap when all it is doing is making up for some other system component deficiency.

Sometimes we hear in the newspapers of some software fault being found years after the software was released. The disparaging feeling these announcements give is” How come it has taken so long to find this bug?” A part of the answer is that it may have taken that long a period of time for that failure to be manifest. Time or aging can be a failure driver. Also the system is evolving. Typically there may changes in: the hardware, the user application, the system software and application software or even the way the user uses the system.

Aging signals

A colleague of mine, who was a plant manager for 3M shared an experience he had touring an automobile factory in Detroit as a young man. A smooth running assembly line was demonstrated by the factory tour guide. He showed how simple it was to determine if a line was running properly. He allowed us to put our hands on the line and feel the rhythm. That was probably the major teaching incident that kind of focused me on vibration.

All machines have a certain rhythm. Today we have various devices to isolate problems. Placing your hand on machine allows you to sense the rhythm. In many cases improper rhythm indicates a bearing may be going out, drive gears may be wearing, hydraulic fluid may be pulsing, and if we wait long enough we know we’re going to have a shutdown. It is an indication of an increasing problem and calls for preventative maintenance.

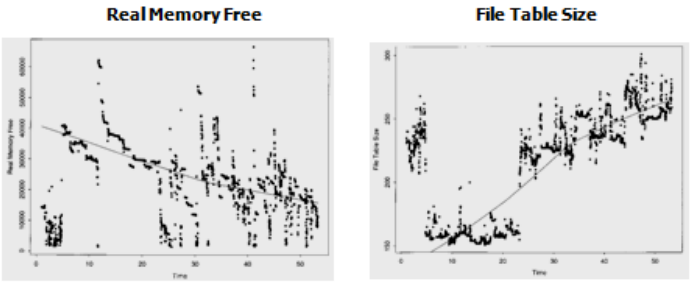
There are similar signals today in our complex systems. System operators can key in on those signals and take preventative action. A company that I was associated with had a 24-7, safety critical computer application. Two computers were operational and one stood in “Hot Standby” mode. The operators would notice that one of the two operating computers would slow down. Seeing this they would vary that computer off line while replacing it with hot stand-by computer. This action restored the computer speed and kept the system operating satisfactorily. We used to have to take the same preventative actions with the MS Windows operating system. “Blue screens” were a common malady in the 1990’s with PC’s. This prompted PC users to reboot their PC before beginning typing a long paper, for instance. We would also regularly save our documents to file, in case of the PC locking up. This is a good practice now but a necessary one back then. The tri-plex computers were essentially being rebooted when they were being cycled from off line to on line. This involves occasionally stopping the running software, “cleaning” its internal state and/or its environment and then restarting it. This is labeled software rejuvenation. Notably, we are rejuvenating the system environment, not the software itself. This action

reduces the likelihood of sudden aging-related failures. These system restarts can be can be applied at the discretion of the user/ administrator when system loads are least, e.g., in the middle of the night. postponing/preventing crash failures and/or performance degradation.

So, software aging results in a deterioration of OS resources, depletion of free resources, and the accumulation of internal errors. The software ages as the queues build up, internal errors accumulate, memory leaks increase, memory fragments, round off errors occur, and data gets corrupted. Professor Kishor S. Trivedi, Duke University, North Carolina, USA has been a long term researcher in the area of software reliability and rejuvenation. Two of his system data graphs are shown below.

Figure 2: Time Plots

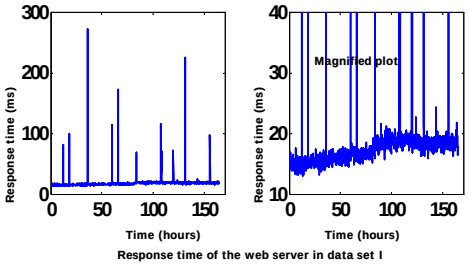
Non-parametric Regression Smoothing




Trend detection: Seasonal Kendall test for trend

This shows that the real memory is being depleted over time, thus further constraining the space the operating code works in. This resource reduction slows the response time of the software. The slower the code gets, the more problematic the software can become. Figure 3 shows the slowing response time resulting from depletion of system resources.

Figure 3: Response time degradation over time - Aging




Response time of the web server in data set I

A classic example of depletion of system resources impacting reliability

- Sometimes system deficiencies grow subtly. The classic example is the Patriot missile whose operational profile dictated the missile shall be relocated daily, thus complicating enemy efforts to locate it by tracking its electromagnetic emissions. This regular relocation resulted in rebooting the Patriot computer every day. This rejuvenated the code set every day. During the first Gulf war, the enemy was not sufficiently sophisticated to localize the Patriot location. That lack of Patriot missile locating ability led to user 's complicity in not choosing to daily relocate the missile. This choice permitted an inherent error to accumulate in the Patriot computer. The inherent error was normally zeroed out with every relocation and subsequent reboot of the Patriot computer. With longer intervals between relocation events the Patriot computer error accumulated to the point of defeating the guidance capability. A consequential outcome when a scud attacked Dhahran was the Patriot missile interception capability was not capable enough to intercept the scud missiles. So 126 allied casualties resulted (http://www.ima.umn.edu/~arnold/disasters/patriot.html). This deficiency had not surfaced when operating under the nominal operational profile.
- Hughes Aircraft System released a missile defense system. It had distributed computers around the user's country. They were designed to detect and intercept incoming missiles. Random failures were observed at the different interception computer installations. There are two kinds of failures: Random and Special Cause (or Pattern Failures). One time the several computer stations were all restarted at the same time. Then the random failures were found to not be random. They were occurring simultaneously at across the system. The timing synchronization revealed the pattern of the failures. So a couple of rules come forth.

Conclusion:

- Software and systems age as their resources get more constrained over operational time of the system. There is an apparent wear out mode in the software via the increasing failure rate over time.
- For maximum robustness of the overall system, the individual computer installations should be run asynchronously so concurrent system failures don't result and bring down the several sites at the same time. The goal is to preserve the redundancy that is designed into the system. Concurrent failures would defeat the redundancy that is designed into the system.
- Run the system components in synchronization mode during testing. This will help reveal the pattern failures and tie them to the timing characteristics of the system.
- There is always a tradeoff between the frequency of restarting the system (cost of down time, system unavailability, labor to do this maintenance action) vs. the impact and saving from being able to defer or avoid an unplanned system interruption (failure). The system restarting could also be keyed to measured deteriorization of system performance parameters such as system response time.

Some useful references on Software Rejuvenation for further study

- Software Rejuvenation: Analysis, Module and Applications, Y. Huang, C. Kintala, N. Kolettis and N. Fulton, In *Proc. of IEEE Intl. Symp. on Fault Tolerant Computing*, Jun. 1995.
- Analysis of Software Rejuvenation using Markov Regenerative Stochastic Petri Net, S. Garg, A. Puliafito M. Telek and K. S. Trivedi, In *Proc. of the Sixth IEEE Intl. Symp. on Software Reliability Engineering*, Toulouse, France, October 1995.
- Analysis of Preventive Maintenance in Transaction Based Software Systems, S. Garg, A. Puliafito, M. Telek and K. S. Trivedi, *IEEE Trans. on Computers, Jan. 1998*.
- A Methodology for Detection and Estimation of Software Aging, S. Garg, A. van Moorsel, K. Vaidyanathan and K. S. Trivedi. *Proc. of IEEE Intl. Symp. on Software Reliability Engineering*, Nov. 1998.
- A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems, K. Vaidyanathan and K. S. Trivedi. In *Proc. of the Tenth IEEE Intl. Symp. on Software Reliability Engineering*, Boca Raton, Florida, November 1999.
- Statistical Non-Parametric Algorithms to Estimate the Optimal Software Rejuvenation Schedule, T. Dohi, K. Goseva-Popstojanova and K. S. Trivedi, *Proc. of the 2000 Pacific Rim Intl. Symp. on Dependable Computing,* Dec. 2000.
- Proactive Management of Software Aging, V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan and W. Zeggert, *IBM Journal of Research & Development*, Vol. 45, No. 2, Mar. 2001.
- Analysis and Implementation of Software Rejuvenation in Cluster Systems, K. Vaidyanathan, R. E. Harper, S. W. Hunter and K. S. Trivedi. In *Proc. of the Joint Intl. Conf. on Measurement and Modeling of Computer Systems*, *ACM SIGMETRICS 2001/Performance 2001*, Jun. 2001.
- An Approach to Estimation of Software Aging in a Web Server, L. Li, K. Vaidyanathan and K. S. Trivedi. In *Proc. of the Intl. Symp. on Empirical Software Engineering, ISESE 2002*, Nara, Japan, October 2002.
- Analysis of Inspection-Based Preventive Maintenance in Operational Software Systems, K. Vaidyanathan, D. Selvamuthu and K. S. Trivedi. In *Proc. of the Intl. Symp. on Reliable Distributed Systems, SRDS 2002*, Osaka, Japan, October 2002.