Evaluating Software's Impact on System and System of Systems Reliability

John B. Goodenough

March 2010

Present-day practice is inadequate for developing justified confidence in software's impact on system reliability. The inadequacies are especially obvious when dealing with systems of systems.

As will be shown in this paper, there is a fair amount of uncertainty among system engineers about how to determine the impact of software on overall system reliability, and this uncertainty is especially clear when attempting to evaluate the impact of software on system of systems (SoS) reliability. This paper discusses the uncertainty that is evident today, based on presentations given at a reliability, availability, maintainability, and testability (RAM-T) summit for a large system of systems. Clearly, new approaches (or at least, better guides) are needed to deal adequately with software aspects of system and SoS reliability. A few suggestions are provided in this paper (the need for giving software failures consideration when doing system-level FMECAs,[1] the need for specifying failure definitions and scoring criteria at the SoS level (not just at the constituent system, or platform, level), and the need for Software Reliability Improvement Programs undertaken during system design), but the main point is that it is not enough to simply formulate software reliability goals or to collect statistics on detected defects.

Some Observations

We start with a real life example. At a RAM-T Summit for a major DoD SoS, various contractors responsible for producing different constituents of the system briefed their approach and findings regarding system reliability, availability, and maintainability.[2] As part of the briefing template, each contractor was asked to discuss their approach to software reliability and its contribution to overall system reliability goals. There were a wide variety of statements in response to this requirement:

- One developer said: "There are no specific software reliability requirements outlined in the … [specification] (e.g., MTBF). However, [our] RAM-T

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Phone: 412-268-5800
Toll-free: 1-888-201-4479

www.sei.cmu.edu

1 FMECA: Failure Mode, Effects, and Criticality Analysis.

2 Although the meeting title implied testability was a topic, it was not discussed.

team understands that software reliability is important. Activities are planned (mostly through software engineering) to ensure the … software is as reliable as possible." They pointed out that they were only responsible for supplying basic computing platforms, including the operating systems (OSs), device drivers, and various OS extensions. It appeared that most of the software delivered would be COTS. They did not describe any process for estimating the failure rate of such software or its possible effect on platform MTBSA[3] and MTBEFF.[4]

Their final statement was: "*SW failure rate* [for the software they were responsible for] *is negligible compared to the HW based on our analyses.* $\therefore SW\ Rel = \approx 1$…" This was equivalent to saying that failures due to the operating system (e.g., Linux) were less likely than failures due to a head crash, for example. This may in fact have been the case, but some analysis should have been provided to show this. Such an analysis could have been difficult to provide since authoritative sources of failure rates for OSs or other COTS products are not generally available. In any event, the contractor's final assertion implied, without further justification, that any software failures would have a negligible effect on MTBSA and MTBEFF for all applications depending on their deliveries.

In this case, the contractor apparently did not have any responsibility to report on likely failure modes and failure rates for the COTS products they were supplying so platform developers could potentially evaluate the impact of such failures on their platform's reliability. Furthermore, it appeared that no team had responsibility for analyzing the potential effects of such errors on platform MTBSA and MTBEFF; at least, no such team gave a presentation.

- A communications system developer said: "All the communication software is currently furnished as part of [system1]. Prognostics and Diagnostics software is part of [system2]. [We are] providing only emulator and simulator software that is not hosted on-board." In other words, since they provide no on-board software, there is no software impact on communications system reliability, availability, and maintainability; any software reliability problems are problems that are the responsibility of the other system developers. While this may be a defensible contractor position, it wasn't clear that anyone would be responsible for taking such potential problems into account when attempting to determine overall SoS reliability.

---

[3] Mean time between system aborts, i.e., system crashes.

[4] Mean time between essential function failures, i.e., failures that are critical to mission success.

- Another developer said: "Software contributes to failure rate but the deterministic nature of software makes it challenging to incorporate into reliability block diagrams that have an underlying assumption that failures are probabilistic (e.g. exponential distribution). Early and continuous Integration & Test (I&T) will be used to detect bugs/defects in software code that will be corrected as they are discovered."

  This statement reflects two unhelpful assumptions about how software behaves in complex systems: first, the emphasis on the deterministic nature of software behavior, and second, the idea that software is perfectible — just get rid of all the "bugs/defects." We'll explore these assumptions further later.

  From the viewpoint of what system engineers know about evaluating the impact of software on system reliability, this developer admitted that they didn't know how to incorporate software reliability estimates into their normal (hardware-based) methods for estimating system reliability. The contractor then made an explicit request for guidance on how to integrate software reliability considerations into their efforts.

- One supplier noted software reliability among their "Issues and Concerns," saying that they needed to understand how to integrate software reliability into their allocations and assessments.

- Another supplier noted that software is critical to achieving RAM-T requirements, but they made no statements about how they are dealing with the software risk other than to say that they "*will monitor and support the software processes, releases, testing, and verification.*"

- One supplier presentation did not have a slide on software reliability. In response to a question, the supplier said that they are using CMMI as their approach to software reliability. They planned to depend on testing and a quality process to produce software that is "reliable," the implication being that their central approach to producing reliable software was to find and eliminate bugs. But, as we shall see, this approach is not sufficient.

- A presentation from a testing center said that they have no failure definitions and scoring criteria at the SoS level because *"there is no SoS definition of failure."* Of course, without a definition of failure, it is impossible to even begin to consider how to measure or improve SoS reliability.

In short, the presenters were all over the map regarding their approaches to software reliability. One said it wasn't their problem and the others provided no definitions or provisional analyses; they just waved their hands. One supplier admitted they needed guidance on what would be an appropriate approach. Moreover, the discussion of software reliability did not extend into the systems reliability realm. No one had any significant or useful discussion of an approach that would take into account the impact of software on platform reliability, avail-

ability, and maintainability, much less SoS reliability.[5] The lack of software reliability analysis was especially significant compared with the extensive analyses showing estimated MTBSA, MTBEFF, and A_o for the hardware system and components.

It seems clear that with respect to platform reliability and availability,[6] the developers needed guidance on how to assess (and improve) software's contribution to platform reliability and availability. They needed to know to what extent essential platform functionality is dependent on software, e.g., they needed to at least perform a failure mode analysis that takes software into account.

Reliability improvement programs are not just for hardware.

A standard definition of reliability growth activities is: *Reliability growth is the improvement in a reliability parameter over a period of time due to changes in product design or the manufacturing process. It occurs by surfacing failure modes and implementing effective corrective actions*.[7] The part of the definition referring to "product design" can be readily applied to software if we consider software development activities that are (or could be) devoted to "surfacing failure modes" in the software design. It was clear that contractor development organizations know how to analyze their hardware designs to surface failure modes. And once the significant failure modes have been identified, they know how to determine the probability of failure and how to redesign the product to reduce or eliminate the possibility of this failure mode occurring. What is lacking in typical software development activities (outside of safety-critical or spaceborne applications) is an explicit activity focused on identifying possible or critical failure modes due to software and mitigations of these failure modes to reduce either their criticality or their likelihood.

A striking difference in software and hardware reliability engineering evident in the contractor presentations was the explicit discussion of funding for Reliability Improvement Programs (RIPs). In typical software development planning, no funding (i.e., effort) is allocated to *identify and reduce* the impact of possible software *design* defects leading to a SA or EFF. Funding is allocated to find and remove code faults, but typically there is no software FMEA followed by design

---

5 To the extent that there was any discussion of how to improve software reliability, it was limited to defect prevention (good processes) and defect detection/elimination via testing.

6 Software has an impact on system availability primarily with respect to the time it takes for the software to recover after a system failure. Typically this is mostly reboot time, but if the system failed in the middle of a complicated process, the operator could require even more time before the system is able to get back to an appropriate operating point unless the software has been designed to minimize the necessary recovery time.

7 AMSAA Reliability Growth Guide, TR-652. See http://www.barringer1.com/nov02prb.htm for links to sections of this guide.

modifications to ensure that even when critical software components fail, the likelihood of a SA or EFF is reduced.

One supplier gave a diagram showing that "99%" of the system's software lay in the middleware and application software layers. The supplier's implication was that the main software impacts on reliability and maintainability would be found there — the RAM contribution of operating system software would be negligible. In fact, this may be a reasonable position, but if so, shouldn't there have been a presentation addressing the contribution to RAM-T made by middleware and by application software? And shouldn't someone's presentation show to what extent such software is sensitive to underlying infrastructure system failure?

The impact of software on RAM-T

Unhelpful assumption: software is perfectible.

Software reliability theories and techniques, as traditionally considered, consider the computer itself 100% reliable and do not consider interactions between software and non-computer hardware. This is due to a mental model that says it is possible for software to be perfect because once a problem is found, it can be removed and will never occur again. An alternative, more realistic, mental model would say that software is never perfect so a system needs to be designed to recover from (currently unknown) faults whose effects are encountered only rarely. Fault tolerance techniques attempt to provide methods for detecting and recovering from such failure effects.

Software failures can be non-deterministic.

Hardware engineers typically think that software failures are deterministic because certain inputs or uses can reliably cause a failure. But although all software failures are deterministic in the sense that they occur every time certain conditions are met, the likelihood of the conditions being met becomes, eventually, a function of usage patterns and history, neither of which are deterministic. In effect, after egregious software faults have been removed, failure occurrences become non-deterministic. In fact, certain types of software failure are inherently non-deterministic because they depend on more knowledge of program state than is typically available. For example, failures due to race conditions and memory leaks typically depend on usage history and, for race conditions, subtle details of system state. Although these are removable design deficiencies, their occurrence appears to be random (although typically the frequency of such failures increases as the load on the software system increases). In short, it is not unreasonable to think of software failures as eventually mimicking hardware behavior in their seemingly non-deterministic occurrence.[8]

---

[8] Of course, we can't push the analogy too far. There is no "bathtub curve" for software, that is, the end of a software system's useful life is not signaled by an increase in its failure rate.

The idea of predicting and then improving the reliability of a software design *before* it has been implemented is foreign to usual software development approaches. Software reliability efforts typically focus on modeling trends in defects discovered in code;[9] reliability modeling and analysis (to determine the potential impact of certain types of software failure) is not routinely done prior to code development. Moreover, work focused on improving the robustness[10] of a design, when done, is hardly ever considered a part of system reliability and availability improvement activities even though such work improves the reliability and availability of a system.

Reliability improvement activities in the hardware realm are focused on identifying design deficiencies that are the source of an unacceptable failure rate. Reliability improvement activities include the identification of failure modes and the identification of stress points that are likely points of failure. Hardware reliability improves as these design defects are remediated. Exactly the same process can apply to software, although neither hardware nor software reliability engineers usually think of it this way. For example, consider a program that uses multitasking. Such programs typically share data among some of the tasks. From a reliability viewpoint, this is a situation that should immediately signal the need for careful design analysis based on how the engineers have decided to ensure against race conditions for accessing and modifying the shared data. If the chosen mechanism is through explicit use of semaphores, experience shows that race conditions are highly likely; very careful analysis will be needed to gain justified confidence that race conditions have been eliminated. Performing such an analysis could be considered as part of a software RIP if one could show that the likelihood of failure would be reduced as a result of doing the analysis and changing the design.

In general, certain software design approaches imply certain potential failure modes. Reliability improvement is possible to the extent these potential failure modes are recognized and eliminated or reduced.

Software RIPs are needed.

The kind of modeling, simulation, and analysis that is done to identify hardware design defects *before* a system is built is exactly the kind of activity that is needed as a software reliability improvement program (SRIP). Although the activities of software reliability *assessment* involve testing and tracking the occurrence of failures (and this is what the software reliability engineering community is typically focused on), the failure impact and redesign analysis activities under-

---

[9] Typically code reviews and testing are used to find code defects.

[10] Robustness is here used to mean the ability of software to behave acceptably even when subjected to unplanned usage conditions. “Behave acceptably” may mean gracefully shutting down or reduced performance (as opposed to no performance at all).

lying software RAM *improvement* are less common and mature (except in safety-critical and space-borne systems, where software dependence is clearly understood to be crucial). Improving development activities devoted to analyzing the potential impact of software failures (regardless of cause) is needed to minimize software's impact on system SAs and EFFs in complex stand-alone systems as well as in systems of systems.

Failure definitions for SoS are not obvious.

The developers of stand-alone systems already have difficulty deciding how software affects system reliability; developing an estimate for a system of systems is even more difficult, in part because SoS failures arise from constituent system *interactions*, not from the behavior of a constituent system *per se*. For example, constituent systems might each, individually, deliver certain information in a manner that satisfies their users, but the end-to-end exchange of information across the SoS could nonetheless fail to meet a timing requirement unacceptably often. Similarly, SoS mission threads require interactions among constituent systems, so mission thread failure is one type of SoS failure. But if a SoS function is defined in terms of a set of mission threads, how ineffective does a mission thread have to be to be considered a failure? Finally, because the constituents of a SoS have different stakeholders, not every member of a SoS will necessarily agree on what constitutes a SoS failure, especially when mitigating such a failure mode requires changes to a constituent system that do not provide any particular benefits to the constituent system's stakeholders.

In a system of systems, there may be no failure definitions and scoring criteria other than at the system (platform) level.[11] If the DoD expects a system of systems to accomplish its mission function, there must be some definitions of what constitutes failure at the SoS level. Although such definitions may be hard to develop, their absence makes it highly likely that insufficient attention will be given to SoS failure modes.

Determining SoS failure modes is not easy today, and this makes reliability evaluation of a SoS more complicated. Reliability evaluation depends on defining what loss of functionality constitutes a system abort (SA), essential function

[11] When this was pointed out at the RAM-T review, one response was that "If the platforms meet their reliability goals, [the SoS] will work as intended." Such a response misses the point because a system of systems is defined by the (allowed) interactions between constituents; understanding how these interactions can lead to undesired SoS behavior is the essence of understanding SoS failure. For example, misinterpretation of shared data is a typical SoS failure mode. Depending on the context, the effect of the misinterpretation might be mild or it might be catastrophic. Such a SoS failure mode is not surfaced if one focuses just on analyzing "platform" reliability because each system can be considered to analyze the shared data completely correctly, from each system's point of view, and yet, the shared view can be inconsistent.

failure (EFF), or non-essential function failure (NEFF).[12] Application software developers need to determine the potential contribution software could make to each defined SA and EFF, i.e., they need to perform a software-oriented FMECA. This analysis would identify what role various software subsystems could play in contributing to a SA or EFF. For example, a mission usage profile analysis could indicate what proportion of mission time each critical software subsystem is in use (with the understanding that a software subsystem is critical if its failure would contribute to a SA or EFF). Further analysis could show what probability of failure could be tolerated before exceeding the overall mission reliability requirement. The next step would be to determine, both by analysis and by actual test, how likely the software subsystems are to meet the necessary reliability requirement. The kind of analysis we are talking about here is, to my knowledge, not commonly done for software as part of overall system and RAM-T analysis. This is an area in which software-reliant programs need to change.

For each failure mode, additional analysis is needed to show what the recovery method will be, e.g., after a software-caused failure, is a system reboot necessary, can the operator fall back to a previously saved "good" state and try again, is there an alternate method that might avoid the subsystem that isn't working? This kind of analysis is needed to determine the likely time to recover from a software-caused SA or EFF. This is more difficult than for the hardware case because diagnosis of the software fault is likely to be too time consuming to allow correction in the field. In addition, the potential causes of software failures are more difficult to determine in advance for software than for hardware. In hardware, you can use analysis to show the components that are most likely to fail eventually under certain conditions of stress or usage, i.e., you know that hardware eventually *will* fail *for a particular reason;* you can know the potential faults in advance of their occurrence. Because of this, you can put diagnostic methods in place in advance of the fault's occurrence. Obviously this is not true for software because software faults are more like unanticipated hardware failure modes due to design errors.

Possible Steps Forward

The observations and analysis above suggest that software's potential contribution to SAs and EFFs is not being adequately addressed by current activities of teams developing large software-reliant systems and SoSs, and that even if some software teams for constituent systems are doing an adequate job in this area, it

[12] These are the failure definition classes for which scoring criteria are required. The scoring criteria define when failures are counted against these classes. For example, a failure prior to start of a mission might not count as an EFF.

isn't clear that their work is integrated with SoS RAM-T efforts. At the minimum, given that a SoS RAM-T team has the responsibility for overall SoS reliability, the SoS RAM-T team should be aware of all activities helping to ensure that software's contribution to SoS SAs and EFFs is within acceptable limits.

- **Recommendation**: A SoS RAM-T team should determine failure definitions applicable to the system as a whole as well as SA and EFF failure definitions applicable to constituent systems. Without a definition of SoS failure, it will be impossible to conduct a reasonable analysis of software's contribution to SoS reliability and availability. These failure definitions will be easier to develop if there is first an understanding of the general nature of SoS failure modes and what makes them peculiar to systems of systems (e.g., see footnote 11).
- **Recommendation**: Given definitions of failure, a RAM-T team should help lead an assessment of software's contribution to potential failure modes for constituent systems and SoS configurations of these constituents. Of course, the RAM-T team will need a lot of assistance from the software development team. This is a potentially big job, but without such an analysis, it will be impossible to decide whether a SoS design is sufficiently robust against failure modes leading to SAs and EFFs.
- **Recommendation**: Given an understanding of a SoS's software-dependent failure modes, the next step is to map mission profiles to particular software functions to develop an understanding of which functions make the biggest potential contribution to SAs and EFFs. This leads to an analysis of the software design to determine whether the design appropriately mitigates against the critical failure modes, and if not, to recommend that appropriate changes be made.

Carrying out these recommendations on a pilot basis for some subset of functionality and missions on a DoD SoS would give a good indication of the difficulty and importance of the task.

References

Maier, M. "Architecting Principles for Systems-of-Systems." *Systems Engineering* 1, no. 4 (1998): 267-284.

Acronyms

AMSAA	Army Materiel Systems Analysis Activity
CMMI	Capability Maturity Model Integration
COTS	Commercial Off The Shelf
EFF	Essential Function Failure
FMEA	Failure Modes and Effects Analysis
HW	hardware
I&T	Integration & Test
MTBEFF	Mean Time Between Essential Function Failure
MTBF	Mean Time Between Failure
MTBSA	Mean Time Between System Abort
OS	operating system
RAM	Reliability, Availability, Maintainability
RAM-T	Reliability, Availability, Maintainability - Testability
RIP	Reliability Improvement Program
SA	System Abort
SW	software

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about SEI reports, please visit the publications section of our website (http://www.sei.cmu.edu/publications).