Predicting and Improving the Reliability of Software Intensive Systems

Larry Bernstein
Email: lbernstein@ieee.org
By Software Engineering Professor
NJIT Dept. of Computer Science, Newark, NJ

Introduction

Can you predict the reliability of your software system; understand what makes software reliable and how to make it more reliable with the least effort? If there was an analogue to the Hardware Bathtub curve[1] with its period of constant failure rate we could apply classic reliability engineering theory and practices to certain classes of software-intensive systems. Then we could assess a system's availability and, if unsatisfactory, improve it. . This paper puts together the work of many researchers to show how this can be done.

Today's design approaches focus on rapid feature development at the expense of availability that leads to fragile systems. Recent advances allow designers to focus on both features and reliability with the understanding that "reliability is only one aspect of engineering uncertainty.[2]"

Whenever there is a constant failure rate, reliability follows the classic reliability equation

$$R(t) = e^{-\lambda t}$$

Where t is the execution time of the system that starts at t = 0 and continues forever or until it fails or is stopped and the reciprocal of λ is the mean-time-to-failure (MTTF). Software systems are vulnerable to failure whenever a computer executes a latent fault.

At t =0, R(0) = 1 and signifies that the software was launched and initialized properly. This result is obtained only after fully testing the launch of the software system at a specified initial state.

To avoid the effects of software *execution* aging[3] we need to limit t to: t < T. Then relaunch the software at its initial state. This software engineering design technique is called rejuvenation. If the time to launch is t_l, then software availability A(t) is

$$A(t) = \lambda / \lambda + t_l \text{ for the time when the failure rate is constant.}$$

All reliability textbooks teach that the Mean-Time-To-Repair (MTTR)and the MTTF maximizes availability. The issue is how this applies to software systems.
Parnas defined another type of aging, software *environmental* aging, due to feature enhancements added to meet new business needs or to deal with computer or support software changes. Parnas wrote, "There are two, quite distinct, types of software aging. The first is caused by the failure of the product's owners to modify it to meet changing needs; the second is the

result of the changes that are made. This one-two punch can lead to rapid decline in the value of a software product.[4]"

Software Product Lines

This paper focuses on software products that are part of a product line. A product line has products sharing a common architecture, basic feature set, similar subsystems and library of components for a specific problem domain. By the time principal features are developed and incorporated into some specific version the software begins to age.

Each time there is a major feature change, a new version must be released to the customer sites and reliability analysis must be repeated. This excludes most of the one-off software systems that are tailored made to a client's specification. It also excludes estimates during the development process because the failure rates are not constant with time.

Boehm emphasizes that, "A good software engineering practice is to adopt a product line approach. This approach involves determining the right product lines for your organization, developing domain-specific software architectures for your product line, and developing product-line solutions. The CMU SEI Product Line
Practices Web site http://www.sei.cmu.edu/activities/plp/plp_init.html) contains a wealth of useful guidelines…[5]"

Most software products are developed in stages as features evolve. Modern Agile method advocate that iterative or incremental development results in fastest time-to-market; so, waiting for several releases to estimate production reliability is reasonable. Have no fear there are methods for estimating reliability during software development by examining test plans and test footprints; here is a checklist for developers. But, these are not the focus of this paper.[6]

Software Design for Reliability Checklist

1. Hire exceptional people and keep them
2. Create a set of design and coding rules and use them
3. Simplify
4. Use fault tolerant libraries and transfers for on-the-go recovery
5. Limit feature scope and reduce algorithm complexity
6. Check module memory use for memory leaks.
7. Bound the execution domain.
8. Improve maintainability [devote 20% of staff to this effort after beta release]
    a. Use uniform conventions, structures, naming conventions and data descriptions
    b. Optimize code for understandability while minimizing code complexity
    c. Comment to explain extraordinary programming structures
    d. Format code for clarity and understanding
    e. Modularize architecture for configuration flexibility and growth.
    f. Store configuration parameters in a global database for ease of change

g. Harmonize all code after every change to assure compliance with the design and coding rules.

Recent Insights

Studies of Microsoft and IBM software releases resulted in a model that estimates post-release failure rates for software products.[7] Jalote and his partners, "… introduce the concept of *initial transient failure rate* (λ_0) to characterize initial unreliability for a specific system version. This transient failure rate decreases every month by a decay factor (α) Eventually, the product reaches a *constant steady state failure rate* (λ_f) after a *product settling time*. (t_s)[8]

Figure 1 shows the failure rate characteristics with λ_0 being the initial failure rate when the release is first deployed and λ_f is the steady state failure rate. The decay factor, α is a linearly decreasing function of time and describes the failure rate reduction curve before it settles to a constant value.

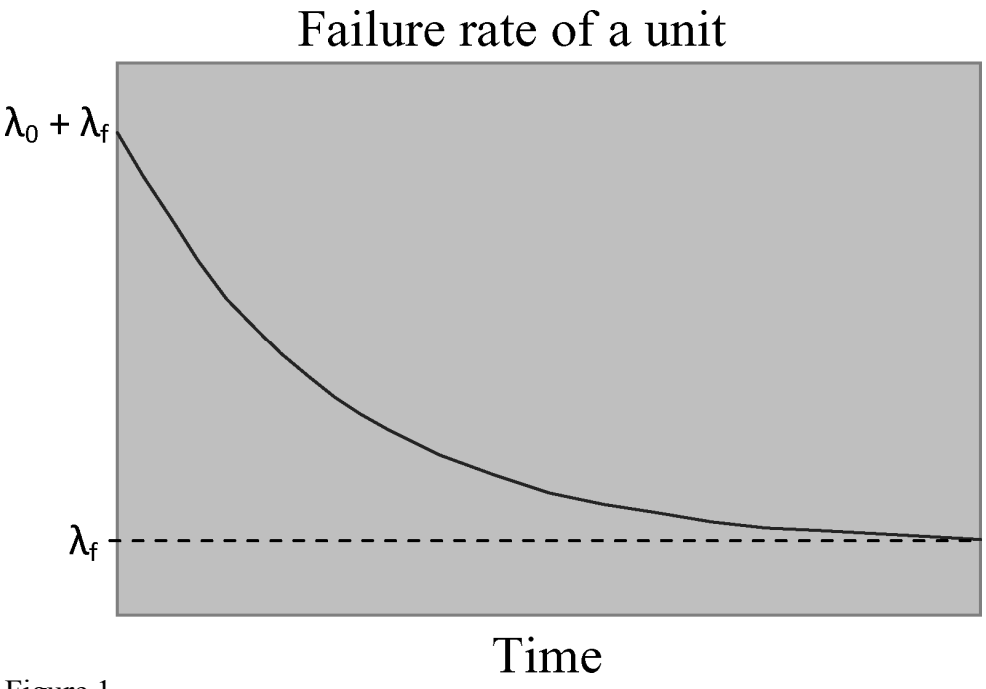

Figure 1

The curve in figure 1 follows the same conceptual reliability model used by engineers to estimate the reliability of systems using electronic devices where infant mortality is characterized by an initially high but rapidly decreasing rate. Wear-out, typical of the "bathtub" curves for mechanical devices, is not included because when electronic components fail they are replaced. [9] Software typically fails like these electronic devices. When it fails, it is relaunched with new data inputs. Only if the relaunch also fails is a code change made, tested, installed and relaunched. Using this curve for software is

counter-intuitive; but, experienced software project managers and, now, data [10] supports the observation that the software failure rate decreases with time, *even when no software changes are made.* We don't understand why this happens; but it does! It might be due to"

1. Users learning to avoid faults that cause failure.
2. Users relying on a small set of product features, thereby reducing the number of fault carrying paths that are actually exercised.

The steady state failure rate (λ_f) may not be predominantly due to software bugs. It is likely due to changes in computer configurations and system administration difficulties. This suggests an engineering approach that increase feature testing to decrease λ_0 and architecture non-feature testing to decrease λ_f.

If the settling time is defined as the time it takes for the failure rate to get within 10% of λ_f then

$$t_s = [\log (\lambda_f / \lambda_0) + \log (0.1)] / \log(\alpha),$$

Recall that α is a strong function of the learning curve of the software system. A high α reduces the settling time and occurs when a system is explicitly designed to be easy to learn, to use, to install and to administer. α must be re-measured for each product version. So, investing in human factors design reduces the settling time (t_s) and follows the mantra "usability equals beauty.[*]"

A New Understanding Software Reliability

Traditional software reliability models generally assume that software reliability is a function of software faults and remains unchanged if the software is unchanged. But, field experience shows that the software product failure rate often gets smaller with time, even when there are no code changes. This may be due to users learning to avoid the situations that cause failures or their using a small number of features. Eventually a steady state constant error rate is reached that can be reduced by making the software product less sensitive to its environment or making it easier to configure or administer. The time it takes to go from the initial failure rate to the steady state failure rate is the settling time. A good approach is to shake out a software system in a pilot installation with problem domain experts who are naïve system users before wide deployment Measuring and reducing the settling time during this shakedown will reduce training time and user errors later.

After the Settling Interval

After reaching t_s a constant steady state failure rate λ_f occurs and so we can compute the reliability, R(t) by taking its Fourier Transform;

---

[*] This design guidance was found in the Rural Toy Museum, in Kurashiki (倉敷市), Japan.

$R(t- t_{s)} = \int \lambda_f \exp[-\lambda_f (t-t_s)$ for { t from 0 to ∞} ,

$R(t) = \int \lambda_f \exp[-\lambda_f (t)$ for {t from t_s to ∞},

$R(t) = \exp(-\lambda_f t)$ from $\{t_s$ to $\infty\}$,

Professor Sha of the University of Illinois has written eloquently on how simple software leads to reliable software[11].

Sha sets $\lambda_f = E/kC$ with reliability

$$R(t) = e^{-kCt/E},$$

where C is a complexity measure of the effort needed to make a system appropriately reliable over that needed for nominal development. E is the development effort expended and t is the continuous execution time for the software. The development effort (E) can be estimated by such tools as Checkpoint, COCOMO or the Software Life-Cycle Model tool (SLIM) approach found in most software engineering textbooks and k is a product line calibration scaling constant. Project managers can choose to apply more staff than called for by COCOMO to improve the reliability of the software. Development effort is a function of the complexity of the software in the models, so average complexity should be used in the COCOMO equations

E = Productivity^{-1} (Developed and Delivered source lines of code) $^{\text{av complexity}}$

where Productivity is the small project capability of the software shop in source instructions/staff month[12].

As pointed out earlier R(0) = 1 means that the test team has verified that all the startup failures are removed through classical unit, block and system testing.

I extended the equation by adding an effectiveness factor (ε) to the denominator: $R(t) = e^{-kCt/E\varepsilon}$, where ε reflects the investment in software engineering tools, processes and code expansion that makes the work of one programmer more effective.[13]

Sha's model of reliability is based on these observations:

a. The effort to make a system reliable adds complexity which introduces faults and faults lead to failures. For a given execution time software reliability decreases as complexity increases.
b. Faults are not equal; some are easy to find and fix and others are not. Software failures are not random but occur when faults are executed. These residual faults that elude extensive testing are often obscure and failures are intermittent and they are exercised by specific input dependent data patterns.

c. All budgets have limits, so there is not unlimited time or money to pay for extensive and exhaustive testing.

Recapitulation

The longer the software system runs, the lower the reliability and the more likely a fault will be executed to become a failure. Reliability can be improved by limiting the execution time or by investing in tools thereby increasing ε, or by simplifying the design thereby decreasing C, or by increasing the effort (E) or making it more productive or an engineered combination of these factors. Of course these factors may also decrease the initial failure rate λ_0.

The software engineer can make tradeoffs between schedule, level of effort, complexity, tools and execution time by using these equations. It is reasonable, if unorthodox, to model the software engineering process based on this model. The longer the software executes, the more likely it is to execute a latent fault that soon become a failure.*

λ_0 and λ_f is a function of the factors a software project manager controls throughout the development process. For example, by providing better tools (such as higher-level languages), the reliability of the final product improves.. The project manager reduces the complexity of the system by reusing reliable components and properly integrating them, again making it more reliable. Adding staff beyond the minimum staff predicted by models so that effort can be placed on such activities as diabolic testing and system audits is another way to increase reliability. Specific technologies like software rejuvenation can bound software execution[14] to make the software less likely to execute latent faults.

Controlling changes to the environment reduces the effects of configuration changes and reduces settling time.

Summary

System Trustworthiness is a measure of how safe, how secure and how available it is and has the implied quality of "no surprises." Users (end user or developer integrating a component into a system) have good reason to expect to understand the behavior of the software under all anticipated ranges of inputs and environments. This raises questions about how well the operational context and likely evolution of the system are understood. It also implies that in order to be trustworthy, software must be robust in the face of unexpected uses and evolutionary change, or at the least, if it is fragile, it has to fail in understood ways. Current software executing presents too many surprises; but the sources of these surprises can now be better understood and actions can be taken to mitigate risks,

* A failure is a state of no response to external stimuli due to the execution of a fault.

The software engineer must design for transparency and for reliable operation in the defined execution environment. The quicker a systems gets to a constant failure rate the more dependable it becomes.

During system reliability tests operate the system in a simulated field environment, measure λ_0, α and λ_f. Compute the settling time and any necessary changes to meet performance expectations.

Bio

- Larry Bernstein is Professor of Software Engineering at NJIT, Newark, NJ.
- His book "Trustworthy Systems Through Quantitative Software Engineering," Lawrence Bernstein and C.M. Yuhas, Wiley, 2005, ISBN 0-471-69691-9 is taught in several software engineering undergraduate and graduate programs.
- He had a 35-year distinguished executive career at Bell Laboratories managing huge software projects. His systems are used worldwide.
- He is a Fellow of the IEEE and the Association for Computing Machinery for innovative software project management that he introduced to Bell Labs.
- He was an active speaker on Trustworthy Software in the IEEE Computer Society DVP program, visiting a chapter monthly in the US, Europe, Canada and Mexico. He served on the Board of Governors with the IEEE Communications Society. He is available to give seminars and talks for the IEEE Reliability Society in the Northeast.

References

[1] O'Conner, Patrick Practical Reliability Engineering fourth edition by, Wiley 2002, ISBN 0-470-84463-9 and see http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/#bathtub

[2] Op. Cit.

[3] http://srejuv.ee.duke.edu and K. S. Trivedi, G. Ciardo, B. Dasarathy, M. Grottke, A. Rindos, and B. Vashaw Achieving and Assuring High Availability. Proc. 13th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems/22nd IEEE International Parallel & Distributed Processing Symposium, April 2008

[4] Parnas, David Lorge, "Software Aging, Invited Plenary Talk," *Proceedings, ICSE '94 Proceedings of the 1*

[5] Boehm, Barry "Managing Software Productivity and Reuse," *Computer*, vol. 32, no. 9, pp. 111-113, Sept. 1999, doi:10.1109/2.789755

[6] Bernstein, Lawrence and Yuhas, C.M. Trustworthy Systems Through Quantitative Software Engineering, Wiley 2005, ISBN 978-0-471-69691-9, chapter 11

[7] Jalote, P., Murphy, B., and Sharma, V. S. 2008. Post-release reliability growth in software products.
ACM Trans. Softw. Engin. Method. 17, 4, Article 17 (August 2008), 20 pages. DOI = 10.1145/
13487689.13487690 http://doi.acm.org/10.1145/13487689.13487690

[8] Op. Cit.

[9] Klinger, David, Nakada, Yoshinao and Menendez, Maria, AT&T Reliability Manual, Van Nostrand Reinhold, New York, 1990, ISBN 0-442-31848-0, Chapter 1, page 17,18.

[10] Op. Cit. reference 7 see figures 5 and 6.

[11] Sha, Lui. "Using Simplicity to Control complexity," IEEE Software, July/August 2001, Volume 18, Number 4, IEEE 0740-7459/01, software@computer.org, page 27.

[12] Boehm, Barry et al. Software Cost Estimation with COCOMO II, Prentice Hall 2000, Upper Saddle River, NJ, ISBN 0-13-026692-2, see page 44-45 and use the nominal value (1.0) for CPLX.

[13] Bernstein, L and Yuhas, C.M. "Software Investment Strategy," Engineering Management Journal, Vol. 7, No. 4, Dec. 1995, ISSN 1042-9257 and ACM SIGSOFT March 1996 and

[14] Bernstein, Lawrence, Yao, Yu-Dong, and Yao, Kevin "Software Rejuvenation: Avoiding Failures Even When There Are Faults: a Study of ARQ Wireless Protocol Implementation in C++,", The DoD SoftwareTech, www.softwaretechnews.com, October 2003, Vol. 6, No. 2, DACS PO Box 1400, Rome, NY 13442-1400 and see http://srejuv.ee.duke.edu/ for current information, theory and practice.