

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

George Novack

Defects for Sale Revisited

George expounds on Software Reliability. Go with him as he kicks the tires and puts some software through the paces. Before he's done, you'll know how you can quantify software reliability.



In his recent article, "Defects for Sale," George Martin made an important point all engineers should live by. The same, but less general, concern was raised several years ago when software developers (excluding many commercial publishers who maintain the cavalier attitude that "all software has bugs" and continue to sell us upgrades) recognized the need for bug-free code.

Participating in the development of a multimillion-dollar satellite, only to see it die as a result of a software bug during the first orbit, does not help in getting another aerospace contract. To avoid such disasters, the industry developed a rigorous methodology, along with many effective tools, for software development, testing, verification, and validation. Concurrently, the ubiquitous DO-178B standard nailed down the process and documentation requirements for producing reliable, bug-free code.

But that is another subject. What I want to talk about is the expression "reliable code," also known as "software reliability." When it was first introduced, many critics belittled it, arguing that software does not wear out, therefore, there is no such thing as software reliability to be concerned with. Obviously, they were playing with the semantics, pretending as if they did not understand that reliable in this case did not mean a failure due to the wear and tear, but that it meant a failure due to a defective design.

And, there were those who argued that software should be treated the same way as the reliability of hardware. They wanted to quantify software reliability, give it a number, that you could plug into a Fault Tree Analysis (FTA) for an entire system. One such study was sponsored by the respected Rome Laboratories, publisher of the military reliability bible, MIL-HDBK-217.

The purpose of the "number" is to define probability of existence of a software bug or a specification non-compliance in a given piece of software. Similar to hardware reliability, we start the analysis by determining the initial failure rate λ_0 , which estimates the fault content at the beginning of the formal system testing (in other words, when the software guy thinks he's finished). The fault is not just a bug! It is anything, including incorrect specification, that has caused the software not to perform as expected.

$$\lambda_0 = f \times K \times w_0$$

where f = linear execution frequency
 K = fault exposure ratio. $K = 4.2 \times 10^{-7}$ is considered an average value.
 w_0 = number of faults in the program.
It is defined as:

$$\omega_0 = \omega_f \times I_s$$

where ω_f = fault density. The recommended value based on experience is 3.6 faults per 1,000 source instructions (or lines of code).

I_s = number of source instructions

$$f = \frac{r}{I_s \times C_{ex}}$$

where r = processor speed expressed in MIPS.

C_{ex} = language expansion ratio. C_{ex} = 2.5 for C language.

The value of r is estimated by dividing the internal microprocessor clock speed by the average number of clock cycles per instruction estimated from the specification sheet. Let's assume you are working with a MC68HC11 microcontroller running on a 4-MHz clock. Assuming an instruction requires, on average, four cycles:

$$r = \frac{4 \times 10^6}{4} = 1 \times 10^6 \text{ (or 1 MIPS)}$$

You will multiply the result by 3,600 to express the failure rate per one hour of operation. There are also several adjustments to be made. First, software can fail only when it is running, and there are times when the microprocessor is not executing the code, such as during data acquisition from an A/D converter. Therefore, an adjustment coefficient is applied to the program failure rate λ_0 to obtain the system operating failure rate λ_s using the formula:

$$\lambda_s = \lambda_0 \times u_{es}$$

where:

$$u_{es} = \frac{t_C - t_N}{t_C}$$

Based on your program structure, you can estimate, for example, that the software cycle time t_C = 10 ms. In addition, once every program cycle you burn t_N = 50 ms for data acquisition. The resulting correction factor (u_{es}) is 0.995, or 99.5%, and is negligible. Assuming your program has 10,000 lines of code, you can

then plug in the values and calculate:

$$\lambda_s = \frac{r}{I_s \times C_{ex}} \times K \times m \times l \times u_{es}$$

$$\lambda_s = \frac{1 \times 10^6 \times 3600}{10000 \times 2.5} \times 4.2 \times 10^{-7} \times \frac{6}{1000} \times 10000 \times 0.995 = 3.611$$

The result shows that when you start testing, you can expect 3.6 failures for every hour of software execution time. It is interesting to note that the initial quantity of faults depends primarily on the fault-density assumptions and the speed of execution. The fault-density assumptions are based on experience, historic data from similar projects, and the software development process taking place. The failure rate is not affected by the program size!

You also want to express the cumulative effect of the software testing and fixing on the failure rate reduction. Obviously, when a failure occurs during testing, you don't keep running the software but stop the test, fix the problem, and then continue. However, as hard as it is to admit, nobody is perfect, and when you open the code to fix a bug, you are likely not to fix it completely or to introduce another one. This is expressed as:

$$V_0 = \frac{\omega_0}{B}$$

Where B is a fault reduction factor indicating the net number of faults removed from the program per occurrence, and value <1 indicates imperfect debugging. It is customary to use $B = 0.995$.

Assigning variable t as a cumulative execution time since the start of the system testing, the fault density as you continue to test is expressed as:

$$\lambda(t) = \lambda_s \times e^{-\frac{\lambda_s}{V_0} t}$$

The program, as I already stated, has I_s = 10,000 source-code instructions and the initial number of faults in the program is:

$$\omega_0 = \omega_f \times I_s = \frac{6}{1000} \times 10000 = 60$$

$$V_0 = \frac{\omega_0}{B} = \frac{60}{0.995} = 62.827$$

Finally, let's say you have tested your system for 100 hours. This is accumulated time and may consist of five systems, for example, running for 20 hours each. But, you must make sure that all routines are being exercised during this test. The probability of finding a fault after the 100 hours of test would be:

$$\lambda(t) = 3.611 \times e^{-\frac{3.611}{62.827} \times 100} = 0.01 \text{ or approx. 1\%}$$

When performing a Fault Tree Analysis for safety-critical systems, you must show that the probability of a catastrophic failure is less than 10^{-9} . If your software is in line to contribute to such failure, the question is, how long do you need to test it before you can be reasonably sure? Performing the above calculations, you will see that after 400 hours of testing, the probability that our example system contains one more bug will have dropped to 3.341×10^{-10} .

You may dispute the assumptions, especially when you are not adhering to a rigorous software development methodology, but the lesson all of us can learn is that with testing, the potential for a defect will decrease exponentially. Regulatory agencies, having proof that you have logged so many hours of testing, may get the "warm and fuzzy" feeling that there are no surprises waiting. And, it will make us perhaps a little sympathetic to commercial designers, who love to stick us with the upgrade cost for their bloated code. Do the arithmetic for their hundreds of thousands of lines of (often spaghetti) code running on a 500-MHz screamer! You will recognize and perhaps appreciate their plight. If you design software for living, you can use this tool to calculate (ahead of time) how much testing you'll need to perform to ensure your product is not an embarrassment. ☐

George Novacek has 30 years of experience in circuit design and embedded controllers. He is currently the general manager of Messier-Dowty Electronics, a division of Messier-Dowty International, the world's largest manufacturer of land-

ing-gear systems. You may reach him at gnovacek@nexicom.net.

REFERENCES

- Hughes Aircraft Company (sponsored by Rome laboratory at Griffis AFB), "Report RL-TR-92-15", published by NTIS in the volume ADA256 347.
- B. Beizer, *Software Testing Techniques*, Van Nostrand, Reinhold, NY, 1990.
- J. D. Musa et al. (AT&T Bell Laboratories), "Software Reliability, Measurement, Prediction, Application", McGraw-Hill, NY, 1987.

Circuit Cellar, the Magazine for Computer Applications.
Reprinted by permission. For subscription information,
call (860) 875-2199, subscribe@circuitcellar.com or
www.circuitcellar.com/subscribe.htm.