This article is part of the IEEE Reliability Society 2008 Annual Technology Report.

Design Constraints That Make Software Trustworthy

Lawrence Bernstein
Email: lbernstein1946@verizon.net

C.M. Yuhas

Do you lose data when your software system crashes and comes back up again? Too often the answer is yes. Reliable software behavior must be achieved as people come to depend on systems for their livelihoods and, as with emergency systems, their very lives. Software is fundamental to computerized systems, yet it is rarely discussed as an entity whose quality can be controlled with specific techniques. This technology on which systems are built has itself got a weak theoretical foundation. Until some very difficult questions can be resolved to provide that foundation, constraints on software design can result in a more trustworthy product.

Most current software theory focuses on its static behavior by analyzing source listings. There is little theory on its dynamic behavior and its performance under load. Often we do not know what load to expect. Dr. Vinton Cerf, commonly known as father of the INTERNET, has remarked that "applications have no idea of what they will need in network resources when they are installed." As a result, we try to avoid serious software problems by over-engineering and over-testing.

Software engineers cannot ensure that a small change in software will produce only a small change in system performance. Industry practice is to test and retest every time any change is made in the hope of catching the unforeseen consequences of the tinkering. The April 25, 1994 issue of Forbes Magazine pointed out that a three-line change to a 2-million line program caused multiple failures due to a single fault. There is a lesson here. It is software failures, not faults, which need to be measured. Design constraints that can help software stability need to be codified before we can hope to deliver reliable performance. Instabilities arise in the following circumstances:

1. Computations cannot be completed before new data arrive,
2. Rounding-off errors build or buffer usage increases to eventually dominate system performance,
3. An algorithm embodied in the software is inherently flawed.

There are six constraints that can be imposed today on software development that will help prevent these circumstances. Though more study of design constraints is needed, that lack is no reason to neglect what can be done.

First Constraint: Software Rejuvenation
The first constraint is to limit the state space in the execution domain. Today's software runs non-periodically which allows internal states to develop chaotically without bound. Software Rejuvenation is a new concept that seeks to contain the execution domain by making it periodic. An application is gracefully terminated and immediately restarted at a known, clean, internal state. Failure is anticipated and avoided. Non-stationary, random processes are transformed into stationary ones. One way to describe this is rather than running a system for one year with all the mysteries that untried time expanses can harbor, run it only one day, 364 times. The software states would be re-initialized each day, process by process, while the system continued to operate. Increasing the rejuvenation period reduces the cost of downtime but increases overhead. One

system collecting on-line billing data operated for two years with no outages on a rejuvenation interval of one week.

A Bell Laboratories experiment showed the benefits of rejuvenation. A 16,000 line C program with notoriously leaky memory failed after 52 iterations. Seven lines of rejuvenation code with the period set at 15 iterations were added and the program ran flawlessly. Rejuvenation does not remove bugs; it merely avoids them with incredibly good effect.

Second Constraint: Software Fault Tolerance
If we can not avoid a failure, then we must constrain the software design so that the system can recover in an orderly way. Each software process or object class should provides special code that recovers when triggered. A software fault tolerant library with a watchdog daemon can be built into the system. When the watchdog detects a problem, it launches the recovery code peculiar to the application software. In call processing systems this usually means dropping the call but not crashing the system. In administrative applications where keeping the database is key, the recovery system may recover a transaction from a backup data file or log the event and rebuild the database from the last checkpoint. Designers are constrained to explicitly define the recovery method for each process and object class using a standard library.

Third Constraint: Hire Good People and Keep Them
This might have been the first constraint because it is so important, but any software shop can adopt the first two constraints as they set about improving the quality of their staff. Hiring good people is not easy. Today, there are over 100,000 openings for skilled software engineers and this will grow to 600,000 by the turn of the century. One small company projects an average of 16 weeks to bring someone up to speed--8 weeks to fill a job and another 6 to 8 weeks to train the new hire in the ways of the company. This is not news but the high correlation between defects in the software product and staff churn is chilling.

George Yamamura of Boeing's Space and Defense Systems reports that defects are highly correlated with personnel practices. Groups with 10 or more tasks and people with 3 or more independent activities tended to introduce more defects into the final product than those who are more focused. He points out that large changes were more error-prone than small ones, with changes of 100 words of memory or more being considered large. This may have some relationship to the average size of human working memory. The high .918 correlation between defects and personnel turnover rates is telling. When Boeing improved their work environment and development process, they saw 83 percent fewer defects, gained a factor of 2.4 in productivity, improved customer satisfaction and improved employee moral. Yamamura reported an unheard of 8 percent return rate when group members moved to other projects within Boeing.

Fourth Constraint: Limit the Language Features Used
Most communications software is developed in the C or C++ programming languages. Les Hatton's book, Safer C: Developing Software for High-Integrity and Safety-critical Systems (ISBN: 0-07-707640-0), describes the best way to use C and C++ in mission-critical applications. Hatton advocates constraining the use of the language features to achieve reliable software performance and then goes on to specify instruction by instruction how to do it. He says, "The use of C in safety-related or high integrity systems is not recommended without severe and automatically enforceable constraints. However, if these are present using the formidable tool support (including the extensive C library), the best available evidence suggests that it is then possible to write software of *at least* as high intrinsic quality and consistency as with other commonly used languages." For example, a detailed analysis of source code from 54 projects showed that once in every 29 lines of code, functions are not declared before they are used.

C is an intermediate language, between high level and machine level. There are dangers when the programmer can drop down to the machine architecture, but with reasonable constraints and limitations on the use of register instructions to those very few key cases dictated by the need to achieve performance goals, C can be used to good effect. The alternative of using a high level language that isolates the programmer from the machine often leads to a mix of assembly language and high level language code which brings with it all the headaches of managing configuration control and integrating modules from different code generators. The power of C can be harnessed to assure that source code is well structured. One important constraint is to use function prototypes or special object classes for interfaces.

Fifth Constraint: Limit Module Size and Initialize Memory
The optimum module size for the fewest defects is between 300 to 500 instructions. Smaller modules lead to too many interfaces and larger ones are too big for the designer to handle. Structural problems creep into large modules.

All memory should be explicitly initialized before it is used. Memory leak detection tools should be used to make sure that a software process does not grab all available memory for itself, leaving none for other processes. This creates gridlock as the system hangs in a wait state because it cannot process any new data.

Sixth Constraint: Reuse Unchanged
A study of 3000 reused modules showed that changes of as little as 10 percent led to substantial rework--as much as 60 percent--in the reused module. It is difficult for anyone unfamiliar with a module to alter it and this often leads to redoing the software rather than reusing it. For that reason, it is best to reuse tested, error-free modules as is.

Conclusion
Software developers know that their systems can exhibit unexpected, strange behavior, including crashes or hangs, when small operational differences are introduced. These may be the result of new data, execution of code in new sequences or exhaustion of some computer resource such as buffer space, memory, hash function overflow space or processor time. Fixes and upgrades create their own errors. The fact that the only recourse has been exhaustive re-testing limits the growth of software productivity in enhancements to existing systems and modules. Experienced software managers know to ask "What changed?" when a system that has been performing reliably suddenly and catastrophically fails. Under current methods of software production, systems are conditionally stable only for a particular set of input and a particular configuration.

The point is that feedback control theory must be the watchword of software professionals if trustworthy software systems are to be a reality. One way to do this is to constrain the dynamic behavior of software by following design rules. The problem is that we do not have the all rules we need. Even NASA, which creates the best software in the world, must admit to eleven mission failures due to software defects. That’s not good enough.