

# Software Design Rules

Monica S. Lam  
Computer Science Department  
Stanford University  
lam@stanford.edu

Software unreliability is one of the most pressing problems in computer science. A single security hole can compromise the integrity of the entire system or, as software and its errors are replicated, the integrity of entire networks. The recent failures caused by the Nimda, Code Red, and Slammer worms were poignant reminders of these facts. Although buffer overruns have been well understood since the Internet worm in 1988, despite many man-years of effort put into eradicating them, we still cannot ensure that programs obey the simple property of not overflowing their buffers.

Buffer overruns is but one of the design rules obeyed in programs. There are many more and they exist at all levels. The most general properties must be obeyed by all programs. Examples include the rules that multi-threaded programs should not have data races and freed memory should not be used. Rules may be *system-specific*: for example, operating systems must re-enable interrupts after disabling them and passwords should never be stored in unencrypted form. There are also *low-level* ones, many of which are implicitly created as programmers provide an implementation. An example of this category is that input strings to `strncpy` should not overlap or the result of every `malloc` call must be tested to determine if the operation is successful.

High-level design rules are more powerful and succinct than the conventional approach of providing pre-conditions and post-conditions at procedure boundaries and loop invariants. Consider just the simple rules that “null pointers should not be dereferenced” or that “all allocated objects must eventually be de-allocated.” They would have been translated to many lines of specification had we tried to write down the pre- and post-condition of each procedure.

The use of design rules in software development has been demonstrated by tools such as Intrinsa’s PREFIX[1] and the Metal compiler[2] that statically check for conformance of simple design rules in operating systems, and Rational’s Purify that dynamically check for memory errors. We think these tools provide just a glimpse of the role software programming tools can play in the future.

We envision that in the future, each program will be accompanied by a large set of design rules. These design rules include not just all the general rules obeyed by all programs but many application-specific ones. These design rules can be consulted by programmers to help them understand the program; they are also enforced through a combination of static and dynamic techniques. While it is preferable to isolate errors statically and fix the program once and for all, this may not be possible. In that case, it is useful to enforce these design rules dynamically and catch errors that may otherwise compromise the system. Recovery may be possible if programmers can provide appropriate exception handling routines.

High-level design rules will be specified by the programmers. We envision that high-level linguistic support be developed that allows, for example, programmers to say that a web server should not pass any user input string directly to its local operating system in just about as many words.

Low-level design rules are too numerous and tedious to be supplied reliably by the programmer.

Instead, these rules should be extracted directly from the program automatically, using a combination of static and dynamic techniques. Some of these rules can be deduced from examining how each procedure enforces the high-level design rules; some can be mined from the code by observing common patterns. Because design rules are extracted automatically, they are guaranteed to track the code as the software evolves.

Recent research results suggest that the approach outlined above is both promising and feasible. Examples of existing tools that automatically extract design rules from programs include an effective static memory leak detector[5], a static analyzer that has found thousands of bugs in existing operating systems[3], and a dynamic tool that finds root causes of large software systems automatically[4]. Moreover, program analysis researchers have made great stride in recent years in the area of pointer alias analysis and path-sensitive analysis, made possible partly because computers have become very fast and have very large memories. Many more exciting discoveries are likely in this area in the near future. Such precise analysis is key to creating the next generation of powerful software productivity tools.

## References

- [1] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.
- [2] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.
- [3] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.
- [4] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.
- [5] David Heine and Monica S. Lam. A scalable flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

Monica Lam is a Professor of Computer Science at Stanford University. She received a Ph.D. in computer science from Carnegie Mellon University in 1987 and a B.S. from University of British Columbia in 1980. Her current research interests are in improving software productivity and usability of computers. Professor Lam led the SUIF parallelizing compiler project, which produced a widely used research compiler infrastructure known for its memory hierarchy optimizations and interprocedural parallelization. Many of the compiler techniques she developed have been adopted by the industry. Honors for her research work at Stanford include an NSF Young Investigator award, an ACM Most Influential Programming Language Design and Implementation Paper Award, and an ACM SIGSOFT Distinguished Paper Award. She chaired the ACM SIGPLAN Programming Languages Design and Implementation Conference in 2000, and served on the Editorial Board of ACM Transactions on Computer Systems and numerous conference program committees including ASPLOS, ISCA, PLDI, POPL, and SOSP.