Stopping the arms race

Readers of Computer are invited to sign and circulate the following petition. To endorse the statement, please send your name, title, address, and affiliation (optional) to High-Technology Professionals for Peace, c/o American Friends Service Committee, 2161 Massachusetts Ave., Cambridge, MA 02140.

As members of the high-technology community, we are alarmed by the increasing likelihood of nuclear war. Despite the claims of some, there is no reason to expect a nuclear war to be "limited." 1 Soldiers and civilians alike would die in numbers that have no precedent. There would be no effective civil defense nor a realistic plan for "recovery" in the aftermath. Fallout, and the attendant increase in birth defects and cancer, would be the legacy to our children. 2 Therefore, the human race can no longer rely on nuclear might to maintain international order.

As engineers and scientists, we have an ethical responsibility to use our talents in the public interest. Instead, our technical expertise is being diverted into a futile arms race. Technological developments like the MX, in a misguided attempt to mitigate the consequences of nuclear war, actually make such a war more likely by forcing a showdown. 3

We recommend that the United States and the Soviet Union stop the arms race. Specifically, they should adopt a mutual freeze on the testing, production, and deployment of nuclear weapons, missiles, and new aircraft designed to deliver nuclear weapons. This is an essential first step toward lessening the risk of nuclear war. Either country could take modest, unilateral steps that would demonstrate good faith and invite reciprocation.

Our specific recommendations are:

- To undertake a three-month moratorium on nuclear test explosions, to be extended if reciprocated;
- To stop further deployment, for a specific period, of one new strategic weapon or improvement of an existing weapon; and
- To draw up and publish comprehensive conversion plans for the facilities and employees that would be affected by such a freeze.

Alan Hochberg
292 Harvard St., Apt. 4
Cambridge, MA 02139


OAT: Testing to minimize undesirable side effects

How many times have you made a simple programming change, checked it out, placed it into production, and then had something go bad that wasn't even supposed to be related to the change? Consider the fact that only a simple programming change was involved—not an operating system upgrade, not a major application rewrite, and not a hardware upgrade or replacement. Add to this the computer industry's current propensity to distribute work among multiple processors that are sometimes geographically dispersed, and you should be starting to see the beginnings of one of my standard headaches.

Simply stated, in our current incredibly complex computer hardware and software environments, how do we assure that system modifications are efficacious and cause no undesirable side effects? Today's generally accepted practice is to test all modifications for efficacy. In my programming shop, we do this three times: design testing, requirements testing, and validation testing. By the time a modified piece of software goes into production, we're all fairly confident that it does what it's supposed to do. We follow a similar approach when a major system change is implemented or when we shuffle hardware around. What concerns me is that approximately half the time we don't get through validation testing without finding at least one undesirable side effect in a portion of our environment that nobody even considered looking at.

Now you might be thinking that this could be due to a bunch of unaware programmers and analysts. Nothing could be farther from the real situation. Not only are the bulk of our personnel conscious of our complex environment, but our design process also requires explicit analysis and documentation of all affected interfaces. It's not that we aren't trying. It's just that we aren't succeeding often enough. Our environment is too complex.

Well, you might say, it can't be all that bad since we are obviously finding undesirable side effects before they can affect production. Yes, we are finding some, but are we finding all or even enough of them? If, on the third testing iteration, we find side effects roughly one-half of the time, how many times would we find them on a fourth iteration, or a fifth, or a sixth?

I know about the law of diminishing returns. We can only afford to test so much. I also know that in most com-
VALIDATE
AND
OPTIMIZE
Gate Array
and PCB Layouts

Using our heavily documented, performance proven optimizer/verifier program. Detects wiring shorts and spacing violations. Verifies connectivity for all routed nets. Optional controls for via elimination, path reduction, corner diagonalization, adjacent wiring spread.

Write or call for a report on this software, as well as descriptions of our other demonstrably transportable products.

VR Information Systems Inc.
5766 Balcones Dr., Suite 203
Austin, Texas 78731
(512) 458-8131

Computer installations, system (program and data) integrity is critical—in most cases, more critical than efficiency. Obviously, the line must be drawn somewhere, but where?

The answer is not what you would expect. Although I don’t know where to draw the line, I do know a way to enhance system integrity. From this, I can be more confident that undesirable side effects are being minimized. The cornerstone for this process is a special kind of testing—operational assurance testing, or OAT.

OAT differs from normal change-oriented testing in two significant ways. First, OAT is performed regularly, independent of all other activities. Its frequency is determined solely by the period of time a production capability can be allowed to remain out of service versus the cost of testing. For example, communications capabilities may be worth testing every hour, a data-base management system once a day, and particular application programs once a week.

The second significant difference is in each test’s comprehensiveness. Essentially, a test used as part of an OAT program is not comprehensive. Instead, it is a short, simple exerciser of the mechanisms of a capability. The approach is not to test every option, capacity, and limitation of a capability, but to test that the capability is accessible in its usual manner and responds appropriately. Spot checks of key data bases or files associated with a capability are included as appropriate.

My programming shop has been performing OAT for a little over one year now. We actually got started on it because we wanted to ensure reliability of some critical intercomputer communications links. As time went on, we discovered one area after another that was worth testing. Today, we have approximately two dozen tests running between one and five times daily.

The results have been excellent—an average of 11 faults a month have been discovered that otherwise might have gone undetected for several hours or days. Of those faults, at least two have been directly attributable to undesirable side effects from system and application changes. Neither of those faults had been located by our change-oriented testing regimen.

We are still observing side effects beyond those located by change-oriented testing and OAT. In today’s complex environment, it would be paradise to have error-free programs or configurations. The trick is to have a series of effective tools which can be routinely applied to ensure the most reliable performance we can afford.

Change-oriented testing is one such tool: operational assurance testing is another. Based on our success to date, I heartily recommend both.

David A. Feinberg
3662 SW Othello St.
Seattle, WA 98126

Back into reliable software

Currently software reliability and quality assurance utilize a paradigm similar to the one employed in nondestructive testing—that is, since we cannot directly verify the integrity of the system under consideration, we must instead test for likely (i.e., visible) errors in hopes that the level of undetected faults is low enough that system reliability is not unacceptable.

Such a paradigm is necessary when all errors cannot be directly observed and indirect measures are therefore forced upon us. This is not the case, however, with software—all errors lie exposed within the code itself. The difficulty is in being able to discern the actual conceptual operation of the program from the incredibly minute detail of the source code, a task that is humanly impossible. When a task is humanly impossible due to the high level of detail, the obvious candidate for the job is a computer.

What is required in this situation is a very high-level “reverse” language which, using the source code as input, would generate a flowchart of the program, validity limits, and typing of all variables accompanied by a putative problem statement for the program (or module). This language would allow the actual program logic to be directly compared with the desired logic. Similarly, it would compare the actual task performed with the requested task, thereby drastically reducing testing requirements.

Admittedly, this is a nontrivial process. However, not only would it dovetail nicely with current efforts to develop very high-level languages, but in light of a recent study showing the pervasiveness and cost of logic errors, it may even prove to be cost-effective.

Richard D. Ferrante
18 Homer St., Apt. 5
Brookline, MA 02146