



## MAINTAINING CORRECTNESS IN SCIENTIFIC PROGRAMS

By Paul F. Dubois

**T**HE MORE FREQUENTLY A PROGRAM IS CHANGED, THE MORE DIFFICULT IT IS TO MAINTAIN ITS CORRECTNESS. SCIENTIFIC PROGRAMS ARE FREQUENTLY CHANGED

throughout their lifetimes, not just when they're young. For most scientific programs, the rate of change doesn't decrease significantly even after many years. Like sharks, scientific programs that aren't moving are dead.

This high rate of change is bad enough, but the problems are further exacerbated by the increased importance of correctness to the scientific programmer. Most programmers can reasonably tell when their programs are incorrect, but for scientific programmers, this is not the case. A bug that doesn't cause the program to fail in an obvious way could be indistinguishable from an error in modeling the real world with equations, approximating the solution of those equations with a numerical algorithm, implementing that algorithm correctly, and using it only within the domain of its accuracy (see Figure 1). Therefore, the consequences in lost time (and subtly incorrect answers) are much larger for a scientific programming team than for most other programming teams.

Combine a high rate of change (which makes correctness hard to maintain) with an increased sensitivity to failure to maintain correctness and you have a big problem. Solving this problem must be the focus of our methodology, be it for a single person writing a 10,000-line program to a team of 20 or more writing half a million lines. In this article, I'll describe the layered approach that I've found to be the most successful in maintaining correctness in the face of rapid change. I've found that this approach is something physicists are willing to adopt because it actually helps them get their jobs done. I believe you'll find this to be a practical program for improving correctness in your own programs.

### Defense in Depth

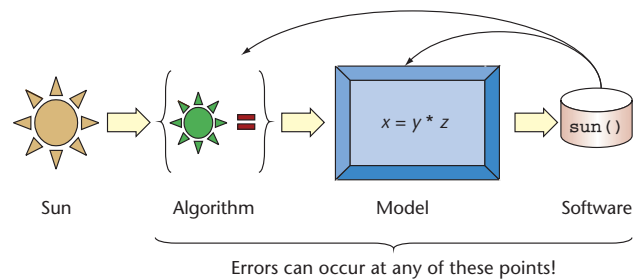
The approach revolves around a *defense in depth*—things can go wrong at the front line, so we must devote part of our

effort to contain breakthroughs and localize their effects. Especially in a project that involves several people, it's very important that one person's mistakes don't end up contaminating the other project members' sources, causing the team to lose time, grow confused, and become annoyed with each other.

Let's look at the layers of defense:

- *Protocol for source control.* Policies and procedures for managing the source can isolate errors when they occur.
- *Language-specific safety tools.* Each computer language has some facilities for ensuring correctness, but they're often underutilized.
- *Design by contract.* Bertrand Meyer's design by contract (DBC) methodology is a good fit with scientific programming, and its optional runtime checking of the contracts catches many errors.
- *Verification.* Defending against bad user input or data is separate from checking contracts.
- *Reusing reliable components.* We use third-party libraries for many things; the biggest benefit of reusing code isn't that you don't have to write it, but that the software is more likely to be correct already.
- *Automating testing.* A simple automation of testing procedures makes it easier to do as much testing as you ought to.
- *Unit testing.* Hand-in-hand with DBC, unit testing ensures component integrity.
- *To-main testing policy.* We insist on a certain level of testing before committing developments to our "main line."
- *Regression testing.* Additional nightly or weekly testing on all target platforms catches problems caused by our own errors as well as those caused by changes in environment.
- *Release management.* A disciplined approach to release management gives most users a stable experience.
- *Bug tracking.* Simple open-source tools can help make sure issues don't get lost.

Now let's look at each of these layers in greater detail.



**Figure 1. The modeling cycle. Errors in modeling, algorithms, and implementation can be hard to distinguish, making correctness of the implementation crucial.**

## Source-Code Management Policies

The strategy I describe here occurs in a context of source-code management (SCM). A project needs either an open-source or a commercial SCM tool, such as CVS, Subversion, BitKeeper, or Perforce. We believe the crucial feature of these tools is their branching model, which must be robust and easy to work with; see <http://perforce.com/perforce/branch.html> for a good discussion of branching models.

Various ideas abound within the professional SCM community as to how to best use such tools, but I believe that there's a "right" answer for scientific programming, especially if more than one person is working on the program. Rather than describe the other approaches, I'll just give you the "right" answer. (Laura Wingerd and Christopher Seiwald discuss these ideas more fully in "High-Level Best Practices in Software Configuration Management;" <http://perforce.com/perforce/bestpractices.html>. I can't recommend this paper enough.)

A part of the repository represents the "main" line—that is, the part that lives forever and evolves over time (see Figure 2). Developers on a project take turns modifying the main line. For a small bug fix, they might work directly on the main line itself, but for most developments, a developer makes a "branch," or copy, of the main line at a particular point in time. The SCM system is cognizant of this branch's history and helps merge any changes in it back into the main line when that main line has changed after the original copy was made. Some SCM's are better at branching than others—be aware that CVS is weak in this regard.

Placing changes from a branch back into the main line is called "integrating up to main;" changes that have been made in the main line can also be "integrated down" to the branch. Either type of integration requires some effort (assisted by SCM tools) to merge the source changes and detect conflicts, but in practice, scientific projects tend to be fairly well partitioned among developers, and most integrations are relatively painless.

A good SCM lets you decide when to integrate down, thus giving you a platform on which to do your work that doesn't change underneath you. You accept changes when your own work is in a stable state and you're ready to accept changes. Using frequent integration down to your branch, it's quite practical to work separately from your team for extended periods. Moreover, you can maintain one of these branches as a variant that never integrates to main, if the conditions are right.

Sub-branches are also possible. Say, for example, that three people on a team work on a certain type of physics. They might have a branch devoted to their team, from

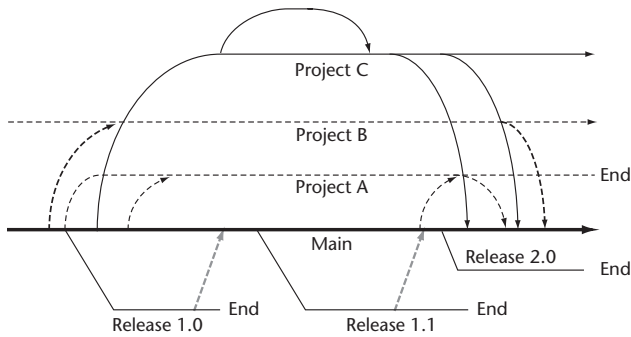
which they can take branches and integrate to and from main when it makes sense. A project manager needs to insist on a reasonable rate of integration to main and not let a team "live in its own world" for too long, though; if they do, users won't get the improvements. Don't do this kind of layering unless you decide it's necessary and appropriate for a particular improvement.

A key aspect of the Wingerd-Seiwald methodology is to think about branches as having an *owner* and a *policy*. The owner is the person ultimately responsible for the branch; he or she decides the policy and directs the branch's integration to and from its parent. The policy is the set of rules for the branch: who can change it, when they can change it, and, most importantly, the conditions a set of changes must satisfy. For a branch I work on by myself, for example, I'm the owner, and my policy might be that I can do whatever I want. For a branch worked on by a physics team, the policy might be that only team members can contribute and that they must get permission from the team leader to submit any changes that might temporarily render the source unusable.

Two little tricks are worth mentioning. One, have a "news" directory in the project, to which developers can add small files describing improvements. A glance at a particular branch's news directory will tell you which contents have been absorbed by that branch up to a certain point. If you make releases, you can clean out this directory and use the contents for release notes. Two, you can use a small program to maintain a queue for integration to main. Each entry contains a one-line synopsis of the changes being made; this utility allows all developers to plan their work, negotiate priorities with one another, and so on. Our little Python utility has commands such as "ready" to add yourself to the queue, "tomain" to indicate the integration is in progress, "done" to indicate completion, and of course "retreat," "flee," and "surrender" to lower yourself down a slot, to the bottom, or out of the queue.

## Language-Specific Safety Tools

We use the source-code language's safety facilities to the fullest extent possible. Most compiled languages have one degree or another of type-safety and are able to detect certain



**Figure 2.** Developers branch from, and integrate back to, a main line that lives forever. Not shown are frequent integrations from the parent lines into the development lines to absorb new code. Release branches don't integrate from the main line after their initial branch. (Figure courtesy of Carolyn Owens, Lawrence Livermore National Laboratory.)

classes of errors at compile time. We try to help the compiler do as much as possible, which means using modules and explicit interfaces in Fortran. A programmer can write better code using the “modern” part of Fortran, but it's more work. Unfortunately, using modules makes the build problem more difficult and fragile. If compilers supporting the latest standard become available, even more safety will be possible.<sup>1</sup>

In C++, we can maximize the type-checking's usefulness. By using class instances instead of raw integers, for example, the programmer can't look up an index on a zone-centered field and then accidentally use it as an index into a node-centered field: the lookup routine will return a class instance, not an integer. Compilers are smart enough to make this safety cost-free.

For parts written in non-type-checked languages such as Python, we still have some options. We can use Python properties and metaclass facilities, for example, to make attributes that check the type of what is assigned to them, only accept numbers in certain ranges, and so on.

### Design by Contract

Bertrand Meyer devised DBC while designing the Eiffel language; it's a methodology for designing the interaction between software components and optionally for checking that design at runtime.<sup>2</sup> Most importantly, DBC *works*. It actually helps you get your job done, and once you try it, you won't view it as some idea imposed by dreamy idealists from the computer science team, but as a practical tool that is worth the investment.

In languages other than Eiffel, macros typically implement the DBC assertions so that a traceback to the source and line number can be provided when an error occurs, and so that it can be omitted in the production code; some people use specialized preprocessors to do a more thorough job of implementing Meyer's ideas. A key concept in DBC is that the pro-

gram's operation should never depend on whether assertion checking is done. Ideally, further assertion checking should be turned off during the execution of a particular assertion statement; this allows a freer use of assertions without worrying about creating infinite loops.

Likewise, in most languages, it's nearly impossible to check class invariants, and some postconditions are difficult to express cleanly. The saving grace here is to provide extensive checking of class invariants at the unit-testing layer. The relationships of a subclass's contracts to those of its parent class are similarly not practical to ensure in most languages; instead, we must rely on a combination of developer education and unit testing.

Meyer developed Eiffel's mechanism as a practical approximation of the formal proving of programs. He recognized that some of the things programmers would naturally want to express in a DBC assertion might be difficult to say or too expensive to check, even in a development version of the code. For example, you might expect a large array to contain only positive values or that a matrix is to be positive definite, but it isn't always practical to check this. Or, a check might, in fact, be simple, but due to its location in a method called too frequently, it might be impractical to check.

We've found the middle ground: we use a separately controlled set of macros for what we call “expensive assertions.” We can turn on this checking when we can't easily locate a bug, without having to pay the price all the time. Maybe we should call them “desperate assertions.”

### Verification

It's extremely important to distinguish DBC from verification. DBC is designed to ensure that a program is correct in how its parts communicate with one another, but it can't ensure that the program is correct when handling data from external sources such as user input. You can't have as a precondition, “The user knows what he's doing and hasn't made a mistake.” Checking user input for validity must *always* be done. DBC statements can be compiled out without harm, so it's important not to use DBC statements for verification.

A second difference is that if a contract statement is violated, the program should halt—it simply can't continue, because we've detected that the software's basic premises aren't true. In contrast, when verification fails, we haven't detected anything wrong with the program. Therefore, if an exception is available in the language, it should be raised. Perhaps the routine that requested the input has a way to correct it; if so, the exception can be caught.

## Reusing Reliable Components

In previous issues, I've discussed why reuse has been difficult in the past and how to design reusable library components.<sup>3</sup> My present project builds 23 libraries written by others. Naturally, we hope and trust that our suppliers are practicing "safe computing" like we are so that they won't infect us, but in the long run, a library used for a wide variety of other uses will get the bugs shaken out of it pretty quickly.

As Meyer has written, the chief benefit of reuse isn't that you save the effort to write the component, but that the component is reliable. The more your program consists of stable components, the less confusion there will be about the source of any of the errors that do occur.

That said, don't simply link to a set of public libraries maintained by others. If you keep and build your own copy of the things you use, your upgrade to a new version of someone else's software will go through the same testing and integration procedure as changes you make to your own parts of the software. If you don't do this, you'll get the "Bad Monday" phenomenon: software that worked fine when you built it on Friday will stop working on Monday.

## Automating Testing

Before I talk about testing schemes, let's discuss the testing process. When I was one of the judges for the Los Alamos Software Carpentry contest, I was surprised to learn that most of the existing industry methodology relies on textual comparison. A test runs and is expected to produce an output file that can be textually compared to a reference file—the files are expected to agree exactly.

This approach is incompatible with our world of floating-point numbers and high rates of change. We rarely change our program without the *intent* to change its output, and different compilers, optimizations, and machines will produce mildly different answers. Therefore, the tests should decide for themselves if they've passed or not: if they have, they can exit normally, and if they haven't, they can exit with an error status.

This makes it easier to automate the running of test suites, because the task is simply to start each job correctly and observe its exit status. As I described in a last issue's Café Dubois, it's then possible to distribute this work over a cluster. Such a tool is most easily produced in a scripting language instead of as a shell script and the like. I hope to release the tool I use for this purpose by the time this article appears in print.

In any case, you have to automate tests because you need a lot of them. As Mark Pilgrim says in *Dive into Python* (<http://diveintopython.org>):

---

### EDITOR IN CHIEF

Norman Chonacky  
cise-editor@aip.org

---

### ASSOCIATE EDITORS IN CHIEF

Denis Donnelly, Siena College  
donnelly@siena.edu

Douglass E. Post, Los Alamos Nat'l Lab.  
post@lanl.gov

John Rundle, Univ. of California at Davis  
rundle@physics.ucdavis.edu

Francis Sullivan, IDA Ctr. for Computing Sciences  
fran@super.org

---

### EDITORIAL BOARD MEMBERS

Klaus-Jürgen Bathe, Mass. Inst. of Technology, kjb@mit.edu

Antony Beris, Univ. of Delaware, beris@che.udel.edu

Michael W. Berry, Univ. of Tennessee, berry@cs.utk.edu

John Blondin, North Carolina State Univ.,  
john\_blondin@ncsu.edu

Michael J. Creutz, Brookhaven Nat'l Lab., creutz@bnl.gov

George Cybenko, Dartmouth College, gvc@dartmouth.edu

Jack Dongarra, Univ. of Tennessee, dongarra@cs.utk.edu

Rudolf Eigenmann, Purdue Univ., igenman@ecn.purdue.edu

David Eisenbud, Mathematical Sciences Research Inst.,  
de@msri.org

William J. Feiereisen, Los Alamos Nat'l Lab, ill@feiereisen.net

Geoffrey Fox, Indiana State Univ., gcf@grids.ucs.indiana.edu

Sharon Glotzer, Univ. of Michigan, sglotzer@umich.edu

Anthony C. Hearn, RAND, hearn@rand.org

Charles J. Holland, Office of the Defense Dept.,  
charles.holland@osd.mil

M.Y. Hussaini, Florida State Univ., myh@cse.fsu.edu

David P. Landau, Univ. of Georgia,  
dlandau@hal.physast.uga.edu

B. Vincent McKoy, California Inst. of Technology,  
mckoy@its.caltech.edu

Jill P. Mesirov, Whitehead/MIT Ctr. for Genome Research,  
mesirov@genome.wi.mit.edu

Charles Peskin, Courant Inst. of Mathematical Sciences,  
peskin@cims.nyu.edu

Constantine Polychronopoulos, Univ. of Illinois,  
cdp@csrd.uiuc.edu

William H. Press, Los Alamos Nat'l Lab., wpress@lanl.gov

John Rice, Purdue Univ., jrr@cs.purdue.edu

Ahmed Sameh, Purdue Univ., sameh@cs.purdue.edu

Henrik Schmidt, MIT, henrik@keel.mit.edu



“Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by ‘bug’? A bug is a test case you haven’t written yet.”

I recommend this policy: if you find a bug, don’t just fix it. Rather, add a test that would’ve detected the bug. I admit, sometimes this isn’t practical, but you’ll be repaid for your efforts time and again.

### Unit Testing

The process of testing whether small pieces of the software work in isolation is called *unit testing*. We can unit test a single class or small cluster of classes, for example, to be sure that the individual methods perform properly (that is, that they satisfy their postconditions and class invariants if they’re called with their preconditions and class invariants satisfied). If each class is internally correct, and DBC monitors the connections between classes, we’ve made a lot of progress toward having a correct program.

In Fortran 90, this might correspond to good module testing, and in C, to checking the public routines in a given header file. Sometimes this is hard to do, especially in Fortran and C, because the context required to call a given routine might be large and hard to construct. Different frameworks exist to help with unit testing. C/C++ can use a Boost unit testing framework ([www.boost.org](http://www.boost.org)), Java can use JUnit (Kent Beck’s first work of this family; [www.junit.org](http://www.junit.org)), and Python can use the standard module “unittest.” The chapter on unit testing in *Dive into Python* is particularly good.

### Integration-to-Main Testing

The next level up from unit testing is testing larger subsystems and whole-code problems. Of course, some items such as multiphysics code might require quite a few tests simply to exercise the different combinations of physics packages and their major options.

Some substantial suite of these tests must be chosen as those that *must* pass before changes can be submitted to the main branch. This policy ensures not only that the main branch lives forever, but that it’s always in a good state. Absent such a policy, the primary problem isn’t that the main line sometimes becomes bad; it’s that between the introduction of an error and its detection and removal, other developers will integrate down from main, taking the infected piece with them. Suddenly, their tests will begin to fail, or they might accidentally integrate the mistake back to main after they’ve fixed it by hand (if the merging isn’t done carefully). Many people on the team could waste time and become confused, trying to

find bugs that they believe they’ve introduced into the code, but that are actually a consequence of the infection.

### Regression Testing

Depending on your program’s running time, you should do some scheme of weekly or nightly testing of the main branch on all target architectures. This suite of tests should augment those in the integration-to-main suite.

You’ll find failures in tests that aren’t the code team’s fault: disks that go bad or are down, changes made by the computer center, and so on. Regression testing is a layer of insurance that is independent of the timing of integrations to the main branch, and it can include problems that are too long-running to be practical for the developers’ daily work.

Regression testing should be done in two modes: contracts off and optimization on, and contracts on. Depending on the amount of overhead introduced by turning on contracts, the to-main integration might or might not require testing with contracts enabled. Working with contracts on during development is, of course, recommended.

### Release Management

With the main branch model, releases are simple. First and foremost, a user who needs a recent fix can always take the current main line as an “experimental” version; the team can even make public such an experimental version from time to time (the weekly regression testing produces a code that could be released as “experimental” if it passes various tests).

When it’s time to make an official release, simply branch the main program into a branch devoted to the release. You can make crucial bug fixes to the release in this branch and integrate it back to main, but no integrations down from main are normally permitted. The idea is to keep bug fixing to a minimum in this branch, instead pushing the main branch forward to the next release. Releases therefore should be made as often as the team and the customers find them useful (I recommend at least three per year). For a small program in its early stages, it’s often best to only have experimental versions available until most of the major functionality is in place—releases consume effort. Some tests for releases are above and beyond the regression suite, especially for a program that requires lengthy runtimes.

When bugs are reported, a bug tracker is an enormous help: simple open-source trackers can yield big dividends in communication between developers and users, and help prioritize tasks and track achievements.<sup>4</sup>


### Future Work Is Needed

All these layers work together to ensure correctness: reliable

libraries give us a solid base, DBC and safe coding practices ensure internal consistency, verification mediates between the user and the code to ensure that bad input doesn't penetrate into the DBC-checked layers, and unit and cluster testing ensure that major subsystems work correctly.

An unsolved problem is the issue of test coverage. Unlike the software you might write for a bank or a small business, the number of variants and controls in a scientific simulation is simply too huge to test them all. Test-coverage tools show which lines of code we aren't testing, but no project I've worked on has ever gotten as far as trying such a tool, and I'm not sure the results would be all that useful. (That's a politically correct way of saying that I'm pretty sure the tools wouldn't be useful or I would've tried them by now. I could be wrong.)

One amazingly frequent source of errors is the code that actually handles errors. It's hard to test things that aren't supposed to go wrong, but some unit test frameworks support this kind of testing in languages that support exceptions. They call the software in a way that should fail, the software throws a particular class of exception, and then the framework catches the expected exception; exceptions of different types are treated as failures.

Research is still needed on a methodology to detect or prevent the wide variety of errors possible with parallel or distributed programs. So far, the human brain is the tool that does the most good. 

## Acknowledgments

I'd like to acknowledge the physics and computer science teams on the Kull project at Lawrence Livermore National Laboratory, who have worked with me to adopt these practices and who have helped refine this approach.

## References

1. J. Reid, "The Future of Fortran," *Computing in Science & Eng.*, vol. 4, no. 4, 2003, pp. 59–67.
2. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice-Hall, 1997; [www.eiffel.com](http://www.eiffel.com).
3. P. Dubois, "Designing Reusable Components," *Computing in Science & Eng.*, vol. 4, no. 5, 2002, pp. 84–90.
4. P. Dubois and J. Johnson, "Issue Tracking," *Computing in Science & Eng.*, vol. 5, no. 6, 2003, pp. 71–77.

**Paul F. Dubois** is retiring after 29 years at Lawrence Livermore National Laboratory as a mathematician and computer scientist. He is a member of LLNL's Center for Advanced Scientific Computing, and System Architect for the Kull Project. He has edited the Scientific Programming department since 1993.

## EDITORIAL OFFICE

### COMPUTING in SCIENCE & ENGINEERING

10662 Los Vaqueros Circle, PO Box 3014

Los Alamitos, CA 90720

phone +1 714 821 8380; fax +1 714 821 4010;

[www.computer.org/cise/](http://www.computer.org/cise/)

## DEPARTMENT EDITORS

**Book & Web Reviews:** Bruce Boghossian, Tufts Univ., [bruce.boghossian@tufts.edu](mailto:bruce.boghossian@tufts.edu)

**Computing Prescriptions:** Isabel Beichl, Nat'l Inst. of Standards and Tech., [isabel.beichl@nist.gov](mailto:isabel.beichl@nist.gov), and Julian Noble, Univ. of Virginia, [jvn@virginia.edu](mailto:jvn@virginia.edu)

**Computer Simulations:** Dietrich Stauffer, Univ. of Köln, [stauffer@thp.uni-koeln.de](mailto:stauffer@thp.uni-koeln.de)

**Education:** Denis Donnelly, Siena College, [donnelly@siena.edu](mailto:donnelly@siena.edu)

**Scientific Programming:** Paul Dubois, Lawrence Livermore Nat'l Labs, [dubois1@llnl.gov](mailto:dubois1@llnl.gov), and George K. Thiruvathukal, [gthiruv@luc.edu](mailto:gthiruv@luc.edu)

**Technology Reviews:** Norman Chonacky, Columbia Univ., [norman.chonacky@yale.edu](mailto:norman.chonacky@yale.edu)

**Visualization Corner:** Jim X. Chen, George Mason Univ., [jchen@cs.gmu.edu](mailto:jchen@cs.gmu.edu), and R. Bowen Loftin, Old Dominion Univ., [bloftin@odu.edu](mailto:bloftin@odu.edu)

**Your Homework Assignment:** Dianne P. O'Leary, Univ. of Maryland, [oleary@cs.umd.edu](mailto:oleary@cs.umd.edu)

## STAFF

**Senior Editor:** Jenny Ferrero, [jferrero@computer.org](mailto:jferrero@computer.org)

**Group Managing Editor:** Gene Smarte

**Staff Editors:** Kathy Clark-Fisher, Rebecca L. Deuel, and Steve Woods

**Production Editor:** Monette Velasco

**Magazine Assistant:** Hazel Kosky, [cise@computer.org](mailto:cise@computer.org)

**Technical Illustrations:** Alex Torres

**Publisher:** Angela Burgess, [aburgess@computer.org](mailto:aburgess@computer.org)

**Assistant Publisher:** Dick Price

**Advertising Coordinator:** Marian Anderson

**Marketing Manager:** Georgann Carter

**Business Development Manager:** Sandra Brown

## AIP STAFF

Jeff Bebee, Circulation Director, [jbebee@aip.org](mailto:jbebee@aip.org)

Charles Day, Editorial Liaison, [cday@aip.org](mailto:cday@aip.org)

## IEEE ANTENNAS AND PROPAGATION SOCIETY LIAISON

Don Wilton, Univ. of Houston, [wilton@uh.edu](mailto:wilton@uh.edu)

## IEEE SIGNAL PROCESSING SOCIETY LIAISON

Elias S. Manolakos, Northeastern Univ., [elias@neu.edu](mailto:elias@neu.edu)

## CS MAGAZINE OPERATIONS COMMITTEE

Bill Schilit (chair), Jean Bacon, Pradip Bose, Doris L. Carver, Norman Chonacky, George Cybenko, John C. Dill, Frank E. Ferrante, Robert E. Filman, Forouzan Golshani, David Alan Grier, Rajesh Gupta, Warren Harrison, James Hendler, M. Satyanarayanan

## CS PUBLICATIONS BOARD

Michael R. Williams (chair), Michael R. Blaha, Roger U. Fujii, Sorel Reisman, Jon Rokne, Bill N. Schilit, Nigel Shadbolt, Linda Shafer, Steven L. Tanimoto, Anand Tripathi



IEEE Antennas & Propagation Society

IEEE

Signal Processing Society

