LETTERS

"Silver Bullet" on target

To the editor:

I thoroughly enjoyed reading F.P. Brooks’ "No Silver Bullet" (Computer, April 1987) in which he predicts that there will be no significant improvements in software engineering methodology. It reminded me of the US Patent Office director's classic turn-of-the-century prescience that there would be no further inventions of any value, since everything of any possible significance had already been invented. Brooks, himself a major contributor to software engineering, now in essence predicts there will be no followers. While I claim none of Brooks' impressive credentials, I have been working with a high-level language somewhat different from the sort he considers. Rather than creating a "tool-mastery burden that increases [with the degree to which it] furnish[es] all the constructs that the programmer imagines in the abstract program," the S/R language has a simple Pascal-like syntax which serves as a data-flow language constructor. With the same S/R language, special-purpose languages can be created "on the fly" for such diverse purposes as to create software to implement a communication protocol, a software prototype for the implementation of integrated hardware, a model for analysis of bond market transactions, strategy analysis, implement a Petri net description language, or simply implement algorithms for ordinary numerical computations, to name a few applications I have tried. In each such application a few minutes (or hours) of preparation is spent first to create the appropriate special-purpose data structures — in effect, to create a new special-purpose language. Then the required programs, algorithms or architectures are created using these specially tailored structures.

Moreover (and probably, first and foremost), the S/R language has a semantics which facilitates formal symbolic as, analysis of its programs. Specifically, a software system called COSPAN implements algorithms used to test S/R programs for any omega-regular property. The problem of computational intractability normally associated with such tests is ameliorated (in "most" cases) by algorithms in COSPAN which perform formal reductions in conjunction with the symbolic analysis. S/R programs, thus analytically debugged, compile into serial or parallel C-code programs (at the user’s direction). Currently, a project is under way for silicon compilation of S/R programs.

While all this is quite new and the jury is still out on whether this will be a "silver bullet," first indications suggest possibly as much as two orders of magnitude increase in the number of lines of debugged C-code which can be produced per programmer-hour. At any rate, it suggests that there still may be room for significant breakthroughs in software engineering.

R. P. Kurshan
AT&T Bell Labs
Murray Hill, N.J.


To the editor:

As a software practitioner (and especially after attending the Monterey ICSE [International Conference on Software Engineering]), I find Fred Brooks’ comments on the state of the software arts refreshing (Computer,
April 1987, p. 10). If only the computer science establishment (well represented at the conference) also listened to the voices from the flock once in a while... In another regard, however, I believe that Brooks' comments are really off the mark, namely in his conclusion that software is incomparably more difficult to master than hardware. To paraphrase a famous comedian: "I've done hardware, and I've done software. Believe me, software is better!" When designing hardware, engineers must fight physics. In software, they only fight their own limitations. Put differently, in software, everything is possible, and therein lies the rub.

I have noticed recently that among engineers a new view of software has begun to emerge — that it can be controlled by imposing on it some of the constraints that nature imposes on hardware. If we regard data as signals and control as logic gates, we can create "software ICE." Provided the "gate" are labelled correctly and proper design rule checks are performed on the "net list," software components can then be interconnected as reliably as hardware components.

Thanks to CAE technology, such checks are standard practice in hardware. Unlike Brooks, I believe that even in software they are no longer just desirable, but eminently practical...

I suspect the main reason for software's apparent intractability lies not so much in its complexity, but in the way it is taught. Few computer science curricula include courses on the principles of design. Yet, CS graduates should know at least as much about design as EEs know about programming....

It became obvious in the plenary sessions of ICSE that the CS mainstay — mathematical formalism — only applies to a few percent of the total software engineering effort....

Worse yet, most software errors today (not counting faulty specifications) can probably be traced to the inability of programmers to deal with present levels of abstraction....

Even in the academically biased atmosphere of ICSE, the one comment from the audience that drew applause in a panel session was this: Circulate more software people between academic and industrial environments. This approach has worked wonders in the hardware world, but software folk on both sides seem to shy away from it. I suspect, mathematicians don't care about industry, and vice versa.

Max J. Schindler
Prime Technology

To the editor:

While I enjoyed Fred Brooks' "article in the April issue, "No Silver Bullet," I also feel he has helped reinforce and perpetuate more software development folklore than he eliminated.

According to Brooks, the key to successful software development is that it is carried out by one software engineer (or very few). Unfortunately, real-world software development doesn't follow a story-book plot. Today's complex software has grown beyond the capacity of an individual — or even a small group — to handle.

I noticed on Brooks' bio that he was involved in the development of the 360. That, in itself, is an excellent example because the operating system and applications software weren't developed by a single group or even by IBM itself...

Today, we have a different situation. Software engineers coming out of school have absolutely no idea what is going on in the operating system because they aren't trained in (nor allowed to tinker with) the internals. But it's OK because, in today's computerized organization, no one can be expected to have the expert knowledge or time needed to manage, analyze, design, code, test, document, and maintain an extensive and detailed software system. For this reason, it is safe to say that software development is a team effort.

As with all group activities, the team developing a new software package must be coordinated to be efficient and effective. Yordon and others have been preaching (and proving) that to be truly successful, software developers have to get away from perpetuating the myth that software development is an art. Instead, they have to take the approach that it is a (software) engineering and manufacturing process.

Once this is done, productivity will increase dramatically.

Just as software itself is a productivity tool, software engineers need to rely on productivity tools to do their jobs. But individual tools fall short of fulfilling the market's needs.

A better solution is a total software engineering environment (SEE) that takes the group dynamics of managerial and technical relationships into account. This approach links, coordinates, and manages the activity of the group. It relays information and acts as the true development environment for the deliverable product....

G.A. Marken
Marken Communications

Brooks replies...

Max Schindler and G.A. Marken are each pushing their own "silver bullets." I obviously advocate and support disciplined engineering practice and SEEs. Neither of their proposals are, however, accompanied by analyses as to how they can even conceivably give order-of-magnitude improvements. My central argument is that so much of the software task is now essence — the fashioning of the conceptual constructs themselves — that any truly promising attack must aim at that part. So, as to that central argument, neither nostrum upsets my rostrum.

R.P. Kurshan's letter, on the other hand, has both preliminary data and a theoretical argument as to why the S/R language addresses the software problem at the concept-constructing, or essence, level. It would seem from his letter to be very similar to object-oriented programming, and it simplifies concept construction in much the same way. Perhaps that bullet is indeed silver — I hope so.

Concerning Schindler's "Believe me, software is better!", I can only say that we have had different experiences. Since entering the computer field in 1952, my efforts, time, and concerns have been almost equally divided between hardware and software. (In 1961-65, for example, I was project manager for System/360 hardware from inception through first customer shipment, and project manager of Operating System/360 and all its compilers and utilities, from the first design of that project through Alpha Test.)

I quite agree with Schindler that software is in principle easier to do, for exactly the reasons he describes (see my The Mythical Man-Month, Chapter 1). In practice, however, I have found software much more difficult to manage and to construct on large scale. This is due, I think, in part to the immaturity of the discipline and in part to the difficulties of doing quantitative assessment and measurement on conceptual constructs that are so nearly disembodied.

Frederick P. Brooks, Jr.

...and adds his own comments


To give credit as due, the illustrations and the explanatory sidebar, "To Slay the Werewolf" (pp. 13), were the work of the Computer Society editors, not part of the paper as delivered at IFIP 86.1
Wrong algorithm used?

To the editor:

I read with great interest the article "Multiprocessing the Sieve of Eratosthenes" in Computer (April 1987). My research at Columbia University includes the problems of primality detection and factorization by use of massively parallel machines (i.e., many thousands of processors)....

The Sieve of Eratosthenes may be the best algorithm for finding all primes between 1 and \( N \), for \( N \) of reasonable size. However, this depends on the model of computation. The implementation of an algorithm under the model depends on the issues you describe, such as load balance, trade-offs between signal and process time, and programming techniques.

However, it is crucial to consider other elements in the problem. I wonder why your article ... does not consider these. For example, other algorithms would be appropriate if there were a large number of processors.

In the evaluation of parallel architectures, it is crucial to use the best possible algorithm to solve the problem. The April 1987 article makes the error of using the wrong algorithm to evaluate the architecture. The Sieve of Eratosthenes is not the "only procedure for finding prime numbers," nor is it the best for finding large prime numbers. Indeed, the available parallelism of "\( p = 6 \)" (p. 57) is certainly not inherent to the problem.

A speedup linear in the number of processors is achievable in the primality detection problem. This can be done by use of randomized algorithms and massively parallel machines. Several such machines were described in the January 1987 Computer. This linear speedup has been demonstrated, as part of my research at Columbia University on the DADO parallel computer. This allows detection of 50 digit prime numbers in a matter of seconds. Similar work has been reported for the MPP computer.

In addition, the "lower bound" of Figure 11 pertains only to memory references. This is distinct from the problem complexity, which is almost polynomial. By use of randomized algorithms a nearly linear speedup can be obtained on massively parallel machines....

Mark Lerner
Columbia University

Author's reply:

As far as I am aware, the Sieve of Eratosthenes and its variants are the only algorithms for finding all prime numbers between 1 and some given \( N \). Lerner confuses this problem with the problem of determining if a given number is prime. In any case, the objective of my article was to use a parallel version of the Sieve as "a test of some of the capabilities of a parallel machine" (p. 50, ¶ 3) and "not to generate prime numbers efficiently" (p. 51, line 19).

I disagree with Lerner that it is "crucial to use the best possible algorithm to solve the problem" in the evaluation of parallel architectures. A specific experiment may measure only some aspects of an architecture, the particular algorithm used is irrelevant. The proximity of the observed run time curve to the lower bound in Figure 11 (p. 58) is a measure of a machine's efficiency. When the experiment is run on another machine, the curves may be closer together or further apart — this has nothing to do with the optimality of the algorithm.

I have been unable to find any research reports or publications related to prime number detection on DADO or MPP. Perhaps Lerner will describe his research in a forthcoming issue of Computer.

Shahid H. Bokhari
University of Engineering & Technology
Lahore, Pakistan

No sixth reference

To the editor:

Although I dislike being perceived as a complainer, it seems to me that reference number 6 (referred to in para. 2 of col. 2 on p. 99 in Computer, May 1987) is missing from the list of references (bottom of col. 3, same page, same issue). If this is the case, would you please consider having the missing reference published as an addendum in a future issue?

Thom Grace
Illinois Institute of Technology

The references are complete. The sixth reference was deleted by a typesetter after the article had been typeset.

Unfortunately, the editors overlooked the corresponding deletion of its in-text citation.

Ed.