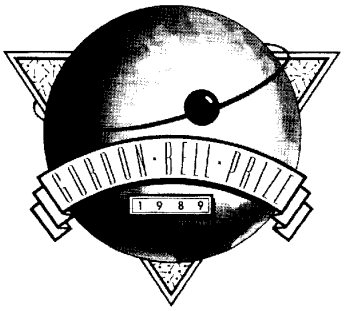

SPECIAL REPORT

1989 Gordon Bell Prize

Jack Dongarra**, **Alan H. Karp**, **Ken Kennedy**, and **David Kuck

This year's winning entries both used the Connection Machine to solve oil-industry problems. The winning performance entry ran six times faster than last year's winner.

The Gordon Bell Prize recognizes significant achievements in the application of supercomputers to scientific and engineering problems. In 1989, two prizes were offered in three categories: performance, price/performance, and compiler parallelization.

The performance prize recognizes those who solved a real problem in less elapsed time than anyone else. The price/performance prize encourages the development of cost-effective supercomputing. The compiler prize encourages the development of smart, parallelizing compilers.

We received eight entries, with several submitted for consideration in more than one category. We examined five entries in each of the price and price/performance categories and one entry in the compiler category, and we awarded a $1,000 prize in each of the first two categories.

Gordon Bell, vice president of engineering at Ardent Computer in Sunnyvale, Calif., is sponsoring two $1,000 prizes each year for 10 years to promote practical parallel-processing research. This is the third year of the prize, which *IEEE Software* administers. The winners were announced Feb. 28 at the Computer Society's Compcon conference in San Francisco.

Three entries presented us with a new problem: The submissions did not involve any floating-point operations. Because the rules assume that scientific and engineering computing involves only floating-point numbers, we had to find a way to compare the apples of Mflops with the oranges of MIPS.

Dongarra, Karp, Kennedy, and Kuck were judges for the 1989 Gordon Bell Prize. Dongarra is a professor of computer science at the University of Tennessee. Karp, who chaired the judging committee, is a staff member at the IBM Palo Alto (Calif.) Scientific Center. Kennedy is a professor of computer science at Rice University in Houston. Kuck is professor of computer science at the University of Illinois at Urbana-Champaign.

0740-7459/90/0500/0100/$01.00 © 1990 IEEE

Fortunately, the two winning entries were so outstanding in terms of Mflops that we were able to avoid confronting the issue directly. Instead, we awarded honorable mentions to two of the entries that did not involve floating-point operations.

In the performance category, we awarded the prize to a team from Mobil Research and Development and Thinking Machines Corp. The team members are Mark Bromley, Harold Hubschman, Alan Edelman, Bob Lordi, Jacek Myczkowski, and Alex Vasilevsky of TMC and Doug McCowan and Irshad Mufti of Mobil Research. Their solution of a seismic data-processing problem ran at almost 6 Gflops on a CM-2 Connection Machine. Not only does this computation rate exceed that of last year's winner by six times, it is almost twice the *peak* speed of a Cray Y-MP/8!

We awarded the price/performance prize to Philip Emeagwali of the Civil Engineering Department and Scientific Computing Program at the University of Michigan at Ann Arbor. He also used a CM-2 to solve an oil-reservoir-modeling problem. His model ran at a price/performance of slightly less than 400 Mflops per $1 million. While the Mobil/TMC team achieved almost 500 Mflops per $1 million, we decided to award only one prize per entry.

The first honorable mention was awarded to Sunil Arvindam, Vipin Kumar (now at the University of Minnesota), and V. Nageshwara Rao of the Electrical and Computer Engineering and the Computer Sciences departments at the University of Texas at Austin for some highly parallel search problems related to VLSI design. The second honorable mention was awarded to Daniel Lopresti of Brown University and William Holmes of the Institute for Defense Analyses Supercomputer Research Center for their use of a processor array to solve a problem in DNA sequence matching.

Connection Machine

A little background on the CM-2's architecture is helpful to understand how this year's winners achieved such outstanding performance.

The Connection Machine is a massively parallel, single-instruction, multiple-data machine, in which all processors execute the same instruction at each machine cycle. It handles conditional execution by providing each processor with a mask bit — a processor executes an instruction only if its mask bit is on.

The CM-2 has up to 65,536 one-bit processors and 4,096 nodes, each containing 16 processors. Each processor has a local memory of 32 Kbytes. For floating-point computations, groups of 32 processors talk to a single floating-point processor capable of a 14-Mflop computation rate. Thus, if all 2,048 floating-point processors can be kept busy, the CM-2 can compute at 28 Gflops. A Data Vault made up of 42 hard-disk drives, connected directly to the processors, provides high-speed I/O.

Some entries did not involve floating-point operations, so the judges had to find a way to compare the apples of Mflops with the oranges of MIPS.

There is no global memory, so the processors must exchange data explicitly. They communicate over a 12-dimensional hypercube, so each chip has 12 neighbors. The CM-2 provides two communication mechanisms: the North-East-West-South network, which lets nearest neighbors communicate in a 2D grid, and a router for more general communications. In addition, there are library routines for commonly used discretizations of partial differential equations, which makes the CM-2 particularly efficient at running finite-difference methods.

To program a machine like the CM-2, you follow the basic data-parallelism concept: You divide the problem into an extremely large number of pieces that need the exact same sequence of operations performed. For example, to add two matrices, you assign one element of each of the two input matrices and the output matrix to each processor, and issue load, add, and store instructions to all processors. If the matrix is no larger than order 256 (256 times 256, or 65,536), adding these two matrices will take exactly as long as adding any single element pair.

You can handle very large problems with virtual processors, which let you treat the machine as if it had many more processors. For example, with a virtual-processor ratio of 4:1, you write the program as if you had 262,144 processors, each with 8 Kbytes of memory. Each virtual processor is time-sliced on the physical processors, so adding two matrices of order 512 will take four times longer than two order-256 matrices. But virtual processors are more than a programming convenience. Because the virtual processors assigned to a single physical processor share a common memory, communications among them is particularly fast.

You run the CM-2 from a front-end processor: a Digital Equipment Corp. VAX, a Sun 4 workstation, or a Symbolics Lisp machine. You do all application development on the front end. The front end executes any instructions dealing with single numbers; the CM-2 executes any instructions that refer to parallel variables.

You program the Connection Machine in Paris (a machine language), Lisp*, C*, or Fortran 8*x*. Both of this year's winners programmed in Fortran 8*x*. TMC's ver-

sion of Fortran 8x does all the work on Fortran arrays on the Connection Machine and all scalar work on the front end.

Performance prize

Finding oil is expensive. Each exploratory well costs millions of dollars, and it is not unusual to sink many wells before oil is found. There is, therefore, a great incentive to reduce the number of dry wells.

Today's engineers use several distinct steps to find new oil fields. First, they collect data in the field using an array of microphones, either placed on the ground or towed behind a boat. They generate sound waves, which reflect off subterranean structures and are detected by the microphones. The data is cleaned up and filtered to remove the effects of noise and multiple reflections.

Using this data, the engineers build a picture of the underlying geological formations by determining the sound velocity at each point in the plane lying directly under the line of microphones. Once the velocity distribution has been determined, geologists can use the fact that different types of rocks have different sound speeds to determine where they are most likely to find oil.

Model building, the step that consumes most of the CPU time, is an inverse problem, solved by repeatedly modeling the sound-wave propagation through different assumed velocity distributions. Today, the most common algorithms used in the oil industry are finite-difference methods applied to the sound-wave equation, which involves second derivatives of the pressure with respect to the two space variables and time. The Mobil/TMC team used this approach in its winning entry.

Mobil/TMC entry. The Mobil/TMC team had to address two problems when coding for the Connection Machine. First, the implicit integration schemes often used for stability on serial machines do not map well to highly parallel SIMD machines, so the team members had to find a stable explicit scheme. Second, it had to find a fast method to handle some complicated boundary conditions.

The team solved the stability problem with a discretization that is fourth-order accurate in space and second-order accurate in time. While such high-order schemes require more operations per point, they do let you take larger steps. These high-order schemes also require more interprocessor communication, which can degrade performance. Fortunately, the CM-2's library of discretization stencils use highly optimized communication patterns to solve partial differential equations.

The boundary-condition problem was tricky because ground is, in effect, infinitely deep. So the model's boundary conditions must let waves propagate through the boundary without significant reflection. The Mobil/TMC team solved this problem with "spongy" boundaries: It modified the governing equations in a small region near the boundary so that waves moving toward the boundary propagate unchanged and waves reflecting from the boundary are attenuated.

The Mobil/Thinking Machines team tackled the problem of oil-field model building, with its tricky stability and boundary-condition problems.

The most straightforward way to program boundary conditions on the CM-2 is to use the mask bit to cause processors assigned boundary points to skip instructions intended for interior points and vice versa. Unfortunately, this approach uses processors poorly. Consider a run with a 256-by-256 grid that has a virtual processor ratio of 1:1 (one grid point is assigned to each physical processor). If you used the mask approach, the machine would spend as much time updating the 1,024 boundary points as it spends updating the 64,512 interior points.

So the Mobil/TMC team handled the boundary conditions differently. The new value of the pressure at any interior grid point is a linear combination of the old values of the pressure at its 10 nearest neighbors. Normally, the coefficients are kept as scalar variables in the front-end processor. If, however, they are replicated on each virtual processor, processors holding boundary points would simply be given a different set of coefficients. Now, all points — both boundary and interior — can be updated simultaneously at the expense of some unnecessary arithmetic on the boundary points. Not only is this twice as fast because all the work is done simultaneously, but it means you can use the stencil libraries for boundary and interior points.

The model submitted used a grid of 1,024 by 4,096 points and followed the time evolution of the pressure for 72,000 time steps. The speed of sound at each grid point was read in from the disk drives, a step that took about 22 seconds. The computation ran for 16 minutes, resulting in a computation rate of 5.6 Gflops. If you ran the same problem on a 1-Mflop workstation, like a Sun 3, you would have to wait two months for the answer.

Price/performance prize

Once you find the oil, you want to maximize the amount you pump out of the ground. The amount of oil you recover can be reduced drastically if you do things wrong. For example, many oil fields are surrounded by water, and water is much more mobile in the rock. If you pump the oil too fast, the surrounding water may intrude into the oil field. You'll end up pumping lots more water than oil, and you'll lose the remaining oil forever.

In addition, secondary recovery is almost always used to increase the amount of oil recovered. Secondary recovery involves injecting water or chemicals into some wells in a field to force oil out of other wells. Reservoir modeling addresses such production-scheduling issues and issues like the number of wells to drill, where to place them, and which wells should be used for secondary recovery.

The amount of money at stake is staggering. For example, you can typically expect to recover 10 percent of a field's oil. If you can improve your production schedule to get just 1 percent more oil, you will increase your yield by $400 million (at $20 per barrel in a 20-billion-barrel field). Maybe that's why 10 percent of all supercomputers are used for reservoir modeling.

Equations. Reservoir-modeling equations are extremely challenging. The simulations must follow many constituents, like water, methane, and liquid hydrocarbons; the equations are highly nonlinear, resulting in steep discontinuities in the solution; the underlying geometry is complicated by irregularities in the rock's boundaries and flow properties; and realistic results normally require following the flow in three dimensions.

Reservoir-modeling equations are similar to other fluid-flow models, but they do have some unique properties. Some equations balance the forces due to pressure, gravity, advection, and drag due to viscosity on the components being modeled. Other algebraic relations, called state equations, specify things like the volume of a given water mass at a specified temperature and pressure and a constituent's viscosity under specified conditions. There are also many prespecified quantities like fluid composition and the rock's properties.

These equations are nonlinear: The gases' vapor pressure is determined by the total pressure, which in turn depends on flow velocity; flow velocity depends on pressure and rock properties, which depends on saturation. (Saturation describes fluid composition: If the fluid is 10-percent water and 90-percent oil, its water saturation is 0.1 and its oil saturation is 0.9.)

Many of these nonlinear relations lead to near discontinuities in the flow. Numerical solutions do not do well in the presence of discontinuities, so you must do something to get accurate solutions in a reasonable amount of time. The most common approach is to add a small amount of artificial viscosity to the physical viscosity, which smears the discontinuities over several grid points without affecting the solution too much.

Time dimension. To advance the equations in time, you can use explicit or implicit methods. Explicit methods are computationally simple: You express the unknowns at the new time in terms of the unknowns at the old time and compute the new values directly. The disadvantage of explicit methods is that the discretization is stable only for extremely small time steps — seconds for reservoir modeling. Because you need to model the reservoir over decades, explicit methods have been considered impractical. Implicit methods involve more computation: You express the unknowns at the new time in terms of both the old values and the new values and then you solve these nonlinear, implicit equations iteratively.

Most commercial reservoir simulators use a combination of these methods called implicit pressure, explicit saturation. These methods solve the saturation equation for each constituent explicitly and the more unstable pressure equation implicitly.

In practice, you can simplify the saturation equations by ignoring convective terms. This assumption is safe because these terms are small relative to the other terms. What is left is an equation that says the force on the fluid is proportional to the pressure gradient.

The price/performance winner solved two very large reservoir-modeling simulations, a high-stakes research area that occupies 10 percent of all supercomputers.

You can simplify the pressure equations by assuming that velocity depends linearly on viscous force, which is accurate for these high-viscosity problems. This leaves a parabolic equation for pressure instead of the more difficult hyperbolic equation you began with. The pressure equations are solved implicitly, using the pressure gradient to determine the velocity, and then the saturations for each constituent are advanced explicitly to the new time.

The iterative methods used to solve the pressure equations involve solving large systems of linear equations at each iteration of the nonlinear solution. These linear equations are much too large to solve with a direct method like Gauss elimination, so they are also solved by iterative methods. Unfortunately, iterative methods work best for symmetrical, positive, definite matrices — which these systems are not. Thus, today's reservoir simulators differ primarily in how well they solve very large linear systems.

Solution. Sometimes you need to step back from a problem to see a solution. Emeagwali did just that. He returned to the original pressure equation. Solving the full system, although more daunting, carries certain advantages. You can compute velocities directly instead of from a numerical derivative of the pressure, so they are more accurate. You can use more general boundary conditions. You can achieve a higher order of accuracy when you discretize the equations on nonuniform grids. Finally, you don't need to change the equations when the flow becomes turbulent.

The only problem is to make it work with reasonable time steps. Although implicit discretization allows large time steps, the CM-2 is much more efficient with explicit methods, which require much smaller time steps. Emeagwali's solution was direct — he simply multiplied these small terms by a large number. His numerical experiments showed virtually no change in the computed results with multipliers as large as 1,000, but he could increase his time steps by a factor of 30.

In a way, this artificially large inertial term is analogous to the artificially large viscosity that smoothes discontinuities. As long as the corresponding terms are kept small relative to the other terms, they can be increased without drastically affecting the results.

Emeagwali's entry described two problems. One was a 2D model that followed the flow of pure oil through a grid of 8 million points. The other, more realistic, 3D model followed the flow of three phases — oil, water, and gas — through a grid of 1 million points. The size of the 3D simulation was limited solely by the CM-2's memory.

These are *very* large simulations. Commercial simulators typically use a few hundred thousand grid points. Due to limited memory, Emeagwali's 3D model solved for only 18 million unknowns per time step compared to 24 million in the 2D model. Both models took about one-sixth

of a second of real time per time step, and both were followed for several hundred time steps. These times translate into 3.1 Gflops for the 2D model and 2.9 Gflops for the 3D model.

Honorable mentions

The two entries awarded honorable mentions this year do not involve floating-point operations.

The University of Texas group applied its search algorithm to three VLSI design problems: floor-plan optimization, which places components to minimize chip area; tautology verification, which verifies the logic of VLSI designs; and automatic test-pattern generation, which finds bit patterns to send through a circuit to see if it is functioning correctly. All three applications involve searching, and all three are extremely time-consuming.

You can visualize a search as a tree traversal. Let's say you want to reach the highest point on the tree. Two approaches are commonly used: breadth-first or depth-first search. A modification of breadth-first search is best-first search.

The University of Texas team submitted a depth-first, parallel search strategy. Their strategy assigns each processor a disjoint part of the tree to search. A processor that has finished its part of the search asks its neighbors for more work. If none of its neighbors has work for it, the processor checks to see if all processors are idle. If some are still working, it rechecks its neighbors for more work. The advantage of this strategy is that, except for checking a single flag to see if everyone else is done, all communication is among nearest neighbors.

The University of Texas team has used this approach successfully on shared-memory machines like the Sequent Balance and BBN Butterfly, on distributed-memory machines like the Intel iPSC and N-Cube, and even on a network of Sun workstations.

Their speedups are uniformly good; in fact, they often obtained superlinear speedups. In one floor-plan optimization problem, 1,024 processors of an N-Cube ran 1,100 times faster than one processor. Such superlinear speedup is not hard to understand: Because there are many processors searching the tree, the probability of finding a profitable branch to search is increased. More importantly, processors can communicate their best result, which lets other processors terminate unprofitable searches earlier. Their algorithm is a hybrid of breadth-first and depth-first searches. In the example cited, the single processor searched more than 15 million nodes while the parallel version searched fewer than 13 million.

The other honorable mention went to Lopresti and Holmes, who submitted some timings on a computer called Splash made up of a linear array of 32 Xilinx programmable gate arrays with 128 Kbytes of memory between each pair. Splash fits in two slots of a Sun workstation and is programmed in a special language called the Logic Description Generator. Depending on the application, this machine, which costs only $35,000 in parts, can run up to 300 times faster than a Cray 2.

One honorable-mention winner applied a search algorithm to VLSI design problems. The other solved a DNA sequence matching problem with a linear, systolic array.

Several applications have been run on Splash, most involving bit-stream computations. The problem submitted matches a new DNA sequence against a library of sequences to find the closest match. In this case, "close" is measured in evolutionary distance: the minimum number of single character changes, insertions, or deletions needed to transform one DNA sequence into another.

Evolutionary distance is important because it plays a role in determining the tree of evolution and its timing. For example, one way to find out if apes or chimpanzees are man's immediate predecessor is to determine evolutionary distance. You can also use evolutionary distance to infer how long it took for one species to evolve from another by simply assuming that mutations appear at random and that the rate of mutation is constant.

Finding the evolutionary distance is similar to a common word game where you turn the word "worm" into "bird" in the fewest number of steps (worm-word-ward-bard-bird). Now imagine that you want to turn "worm" into "human," a word of different length. Of course, what you really want is to find the closest match between "worm" and all five-letter words in the unabridged dictionary. Pretty daunting.

Imagine how hard the problem is if the words you are using are hundreds or thousands of characters long. Now add the complication of there being a different cost for insertions, deletions, and changes. That's the problem Lopresti and Brown solved on Splash, simplified by the fact that the DNA alphabet has only four letters.

To solve this problem, Splash is programmed as a linear, systolic array. A systolic array can be thought of as a synchronized assembly line. Each processor does its own job, but all jobs take exactly the same amount of time. In the DNA matching problem, one string enters at each end, is partially processed at each station, and is passed down the line after a fixed time interval. The term systolic is meant to invoke an image of a beating heart in which blood moves from one stage to another on each beat.

The Xilinx processors can be configured to suit the problem at hand. Here, Splash is configured to have 256 processors. Every microsecond, a pair of characters is fed into the systolic array from the memory. The first chip, chip 0, unpacks the characters into the appropriate bit pattern and sends them down the array for comparison. The sequences move in opposite directions through the array like trains passing. As they pass, the table of evolutionary distances is built up diagonal by diagonal: first the (1,1) element in the array (this is the evolutionary distance between the first character in each sequence); then the (2,1) and (1,2) elements in parallel; next the (1,3), (2,2), and (3,1) elements, and so on. If you are comparing two sequences of length 100, you fill in the table of 10,000 entries in

Continued on p. 110

QUALITY**TIME**

New views of mature ideas on software quality and productivity.

Editor: Vincent Shen
MCC Software Technology Program
3500 W. Balcones Center Dr.
Austin, TX 78759
CSnet: shen@mcc.com

Where design testing fits in

A popular approach to reducing development costs significantly is rapid prototyping, where the prototype is used to assess the functionality and performance of the system being built. Prototypes not only reduce requirement misinterpretations, but code in a good prototype may even be released as part of the final system.

As development schedules get tighter, I expect more and more developers will retain code that was originally intended to be replaced before release. In fact, I once heard a product manager say, "It's better to have the customers use my system and complain than to have them use someone else's system."

How can we develop prototypes that both help us understand user requirements and contain usable code? Nok Viravan and Buster Dunsmore suggest that designs be subjected to a rigorous testing process, just like code. They report on research in the theory, methods, and tools to help designers build better prototypes.
—Editor

Chonchanok Viravan and H.E. Dunsmore, Purdue University

A design is one of the products of software development. Generally, the development process begins with an analysis of user requirements that results in a conceptual-level design. This design represents the customer's view of how application tasks should be carried out.

Developers then base a functional requirement specification on both the requirements set (enhanced through requirements analysis) and the conceptual-level design. This specification leads in turn to a detailed design, from which developers decide how the software should carry out the tasks. Finally, the detailed design leads to code.

Testing code is an important part of the development process. But why limit testing to code? We believe that other phases of the development process — especially design — should be subjected to testing. Like code testing, design testing should execute a design under some specific test scenario to detect design errors and increase confidence in the design's correctness.

Why test design? The need to detect design errors is supported by several studies that suggest that errors can be introduced early in the development process. Some even claim as many as 80 percent of errors are design problems that arise from misunderstanding and miscommunication of user requirements.

Failing to detect design errors can decrease software quality and increase the quantity of errors, since one error may be amplified into many others as it passes through the development.

How can we develop prototypes that both help us understand the user requirements and contain usable code? By subjecting designs to a rigorous testing process.

You can increase confidence in your design's correctness if you can determine the level of its correctness after testing, as is possible with some code-testing methodologies.

Why incorporate design testing into the development process? Because, surprisingly, design testing potentially can reduce not only the number of errors but also the cost and effort involved in development.

First, you can lower both the cost to fix errors and the rework effort if you detect errors early, as is suggested by industrial evidence cited by Barry Boehm ("Improving Software Productivity," *Computer*, Sept. 1987, pp. 43-57).

Second, you can reduce the effort to detect design errors manually in design reviews and walkthroughs if you can exercise the design automatically and thoroughly with test cases.

Third, you can reduce code-testing effort. Because one error can hide another, fewer design errors may make it easier to detect other errors that lurk in the program. Also, if your design-testing methodology gives you high confidence in detecting design errors, you can reduce the effort to test for such errors in the code.

Techniques. Design testing clearly seems like a good idea, but we need techniques to support good design testing. A good technique requires three things: a mechanism to execute the design, a methodology to find design errors, and theoretical support for design-testing methodologies.

Mechanism to execute design. Design execution is a simulation of a design's dynamic behavior — behavior like state changes in its components (dataflow, control flow, and data transformation). The design components' states actually refer to their status, which may range from very general (like data availability) to very specific (like data value). The state of a composite component, like a data transformation, may be composed of the states of its subcomponents, or it may simply be its global state (like idle or suspended).

Although the input and output for design testing varies according to the approach and executable format you use, the test's input and output should at least contain the states of the input and output data, respectively.

You can test a design directly, using an executable design format, or indirectly, using a translation of the original design into an executable format. You can characterize the nature of design-testing mechanisms by the type of states of the se-

lected design components to be observed.

You can translate the design, or part of it, into code and then test the code as a prototype, an approach suggested by Luqi and Valdis Berzins ("Rapidly Prototyping Real-Time Systems," *IEEE Software*, Sept. 1988, pp. 25-36). In this case, the data states you are processing and observing are actual values.

If you use a formal design-specification language, you can execute the design directly, an approach used by Richard Kemmerer's Ina Jo ("Testing Formal Specifications to Detect Design Errors," *IEEE Trans. Software Eng.*, Jan. 1985, pp. 32-43). In this case, the data states you are processing and observing are either actual or symbolic values. The execution of Pamela Zave's and William Schell's Paisley ("Salient Features of an Executable Specification Language and Its Environment," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 312-325), simulates a system's behavior in terms of the sequences of discrete state changes of a synchronous process set.

To persuade developers to adopt design testing, we need evidence to show that the effort and cost involved helps reduce the overall effort and cost of development or leads to measureable improvement in the final product.

Token-based execution is a popular way to execute graphical designs. Execution originates in a petri net and proceeds by distributing tokens and letting these tokens flow through the graph. In general, a transition is enabled when there is at least one token in all its input places. When an enabled transition fires, it consumes the input tokens and produces the output tokens. The design's global state changes as the token distribution changes. (See J.L. Peterson, *Petri Net Theory and Modeling of Systems*, Prentice-Hall, 1981.)

The semantics of token-based execution, however, lie in the semantics of the place that holds the tokens and the information the token carries. The places may represent the design states, or the tokens may carry the state of the components being marked. In a finite-state machine, for example, a place represents a global state. In a condition/event net, a place may represent the data's condition (or state). The presence of a token in such a place implies that the state is current. In a color net, a place may represent data, and tokens carry their values as the state.

Among the tools that support design testing via token-based execution are the System Architect's Apprentice, developed at the University of California at Los Angeles; the Visual Environment for Raddle Design and Investigation, developed at the Microelectronics and Computer Technology Corp.; and Start/Act, developed by Interactive Development Environments and McCabe & Associates.

While we are pleased with the availability of these tools, it is unfortunate that most of them emphasize execution issues instead of error detection.

Testing methodology. In this context, design-testing methodology requires both a test-data selection strategy and, at least, guidelines on how to recognize errors in test results. The ad-hoc nature of both of these requirements is one of the major shortcomings in current design-testing methodologies.

To execute a graphical model like an extended dataflow diagram, you are either supposed to observe token flow visually or examine an execution trace, with few or no guidelines on how to recognize errors. A formal specification language can help you detect errors systematically, if you know the test output's postconditions and you have enough details to manipulate the data's actual or symbolic values.

Theoretical support. We need theoretical support to gain insight into the effectiveness of a design-testing methodology. Theory can help you determine how effectively the selected test data exercises different parts of the design and how reliable the technique is in detecting the errors.

The absence of design-testing methodologies and theories are not the only two problems facing design testing. We also lack a standard design and executable design format. This slows down the spread of design testing because most design-testing tools depend on the design format.

Finally, to persuade developers to adopt design testing, we need evidence to show that the effort and cost involved helps reduce the overall effort and cost of development or leads to a measurable improvement in the final product.

At the Software Engineering Research Center at Purdue University, we are exploring and trying to overcome some of the weaknesses of design testing. We hope this essay will spur others in academia and industry to contribute to this important area.

HUMAN FACTORS

Tools, techniques, and concepts to optimize user interfaces.

Editor: Kathleen Potosnak
Rolm Systems
Bldg. 6, MS 646
4900 Old Ironsides Dr.
Santa Clara, CA 95052
Compmail: k.potosnak

Big paybacks from 'discount' usability engineering

In the last issue, you learned how to build a usability testing lab. If you have started building your lab (or have already built one), this issue's essay will give you some ideas for putting it to good use without much additional investment. If you don't have a formal lab yet, this article will help you get started without one (and maybe give you the incentive to consider building one).

Jakob Nielsen, assistant professor at the Technical University of Denmark in Copenhagen and author of the recently published book Hypertext and Hypermedia *(Academic Press, San Diego, 1990), shares the results of some inexpensive usability methods that have proven valuable in his own work. He takes you through the methods step by step and demonstrates the benefits along the way.*

—Editor

Jakob Nielsen, Technical University of Denmark

If you have read some of the previous contributions to this department, you know that user interfaces are important for the success of software projects and that several human factors methods exist to improve usability. But have you actually started using these methods in you own projects?

If you are like most software people, you have not.

My experience in large and small companies in Europe, Japan, Australia, and North America indicates that the situation is practically the same everywhere: People want to improve usability but they are too busy meeting deadlines to use the full set of carefully designed human factors methods recommended by usability experts.

Luckily, you can benefit from usability engineering even if you cannot use a 100-percent perfect usability method. You can improve the usability of your software a lot by using a few cheap and simple methods. Of course, even better results might be possible if you invested a larger budget and spent more time on usability work, but the usability-engineering approach is to improve the user interface as much as possible given whatever resource constraints apply.

In most companies, you have to start usability work on a small scale. When the first, quick methods have demonstrated the success of taking usability seriously, management might allow larger usability budgets for future products. The important point is to include a few usability-engineering methods in even the smallest budgets since even the cheapest usability methods have a much greater effect on usability than doing nothing.

The first step. Go visit your customers. It is surprising how many people work for several years on projects without ever having seen the people who will use the product or the location where they will use it. By "visiting customers," I don't mean having your marketing people talk to the purchasing people of customer organizations. I mean having the development people visit the actual users who will have to work with the product every day.

Luckily, you can benefit from usability engineering even if you cannot use a 100-percent perfect usability method.

During such a visit it is important to *listen* to the users. Developers easily fall into the trap of talking too much and spending the entire time teaching users smart tricks. It is important to remain silent and pay attention to signals from the users. If they have your system in place, you can see how they use it and what parts of it present special difficulties. Even if the people are not yet using a computer system, you can still benefit from seeing how they conduct their work and how their working environment is structured.

What you can find. For example, three one-day visits to the branch offices of a medium-sized insurance company produced a list of 130 usability problems. Basically the system design was sound and most of the problems would actually be simple to fix. Many of the 130 items would only be serious problems for novice users. However, even very experienced users were estimated to waste at least 10 minutes every day because of usability problems.

A specific example was shortcut commands that connected the subsystems for insurance-policy forms. These commands were inconsistent and hard to learn, so many on the sales staff did not know them. Therefore, they had to return to the main menu each time they wanted to shift between insurance types. This caused one of the subsystems to clear the basic information about the client, so the agents had to ask clients for their name and address for each additional type of insurance they wanted to buy.

The staff was often interrupted by the telephone or walk-in clients. Unfortunately, several subsystems were not designed for interruptions — users lost all of their work if a transaction was not carried to completion. At one small branch, an agent stated that she never used the damage-claims subsystem during periods where she was the only person in the office and had to answer all calls. In some cases, agents were observed using other agents' terminals (and "borrowing" their passwords) to deal with interruptions rather than quit one of the unforgiving subsystems in the middle of a transaction.

Several cases of problematic error messages also were observed. For example, the message "PF03 was used" appeared

when the agent pressed the PF3 key in a system state where that function key did not apply. There was no indication of why the key was not active.

In another case, the system allowed only one line for error messages, so it had to give an obscure, truncated version of a long message. The full message was available by pressing the help key, PF1, an action the developers in the central data-processing office felt was very natural. But users in the branch office had not made the conceptual leap that told them the help key was doubling as an extended-error-message key. Instead, they wasted a lot of time trying to understand the truncated message. A better design would have spent the one line on the main screen on a brief indication of the error followed by something like "PF1 for more information."

A three-step plan. In addition to the basic activity of visiting customers, there are several steps you can take to include usability considerations in your development process. Here are three simple, inexpensive methods you can add to your development methodology.

• Step 1: Fast prototyping and iterative design. First you have to learn what interface your users need. You cannot learn this by just asking them because users normally do not know what they want. And they never know what they need. After all, they are the users and you are the designer, so you should not expect them to do the design work for you.

Instead, you can do some quick, preliminary design and present it to the users in the form of a prototype. Doing so gives users something concrete to relate to and lets them understand the design in a way no abstract specification could. This is true even if the prototype is nothing more advanced than a few screen designs drawn on overhead transparencies.

You should be able to produce your first prototype in less than one day. There is no need to spend more time on programming an advanced, fleshed-out prototype, since most of it will have to be changed anyway. Instead, rely on iterative design — gradually refine your prototype from the first few hand-drawn screens to a final design. You may use a fourth-generation environment for later prototypes, but you don't have to do so to benefit from this method.

Just as no program is without bugs the first time it is tested, no early version of a user interface is without usability problems. The only question is whether you will discover those usability problems or whether your customers will have to find them for you (and demand that they be fixed in a future release).

I recommend the iterative design approach because it is always much cheaper to correct problems if they are found early in a development process. You need to perform some kind of usability evaluation on your design and redesign the prototype to fix the problems you are bound to discover. Many advanced usability evaluation methods have been proposed, but I recommend that you start by doing heuristic evaluation and running thinking aloud experiments.

• Step 2: Heuristic evaluation. Just by looking at a user interface, you can draw your own conclusions about its usability. This very simple method works best if you know what you are looking for and use certain heuristics as a basis for your judgment.

For example, we know that good error messages should use plain language (no codes), precisely indicate the problem, and constructively suggest a solution. You establish a small set of these rules and go over the interface design to check if it complies. The documents described in "Getting the Most Out of Guidelines" (Human Factors, *IEEE Software*, Jan. 1988, pp. 85-86) are a good source of heuristics.

No early version of a user interface is without usability problems. The only question is whether you or your customers will discover them.

Unfortunately, experience shows that you can only count on finding a small portion of the usability problems this way. In four systems I tested, the proportion of usability problems found by heuristic evaluation was between 20 percent and 51 percent when the evaluation was performed by a single evaluator.

But it turns out that different people find different usability problems. You can, therefore, increase the number of usability problems you find by having several people evaluate the interface independently. In the same four systems, having five evaluators instead of one raised the proportion of usability problems found to between 55 percent and 90 percent.

• Step 3: Thinking aloud. Heuristic evaluation can get you far, but only so far. Users have an infinite potential for interpreting user-interface designs in new and unexpected ways, and so it is impossible to predict everything. You simply must perform some kind of user testing as part of your interface evaluation.

The simplest test method is the thinking-aloud technique: You ask a representative user to use your system to perform certain given tasks and to think out loud while doing so. By observing real users and listening to their utterances, you get a good picture of how users approach the task and how they comprehend the interface design. By listening in on a user's thoughts, you can assess how each individual interface item is interpreted by that user and you will therefore get a very direct understanding of what parts of the dialogue cause the most problems.

It will normally suffice to run a few users through the thinking-aloud method to get data for your redesign. In most cases, you will be able to find a large proportion of the usability problems with only three test users. Obviously, you will not get detailed statistical evidence, and you will not find every single usability problem. But you will have gained the necessary insight into the basic usability issues surrounding your design and you will be able to improve your design.

It worked for me. I used these simple usability methods to redesign a set of printouts for a small bank. A total of 12 bank statements went through eight versions with a resource expenditure of only 90 man-hours. To check whether the "improved" final version was actually better than the original design, I followed up with a more scientifically valid experiment where 152 test users were asked questions about the information contained in the forms. The average proportion of correct answers was 76 percent for the final design as compared with only 56 percent for the original design.

Of course you might complain that a 76-percent understandability rating is far from perfect, but remember that I never claimed that the cheap usability methods would be perfect. They are not. But they will get you a significantly better user interface than if you do nothing.

I call these methods the discount usability-engineering approach. The methods are sufficiently cheap and simple to use that you have no excuse for not using them. Get started now — on your current project. Spend just a few hours on systematic usability work and observe how much better your product gets. Maybe you can use more advanced methods later, but you have to start somewhere. A goal to wait until the "next" project can easily end up being postponed perpetually.

SOFTWARE MANAGER

How to get people and technology to work together.

Editor: Elliot Chikofsky
Index Technology
1 Main St.
Cambridge, MA 02142
Compmail: e.chikofsky
CSnet: chikofsky@northeastern.edu

How to speed development with group sessions

To manage the software process better, it is essential to improve communication. One of the hardest groups to communicate with well is the users of the system being created.

To cope with this problem, several processes like the Joint Application Design are gaining popularity. Richard Zahniser is a software-engineering consultant and experienced group facilitator for JAD and similar techniques.
—Editor

Richard A. Zahniser, CASE Lab

In *Up and Running* (Prentice-Hall, 1984), Dines Hansen described a successful system development that designed and delivered 21 releases of a system in two years. Before that, system definition — getting the users to agree on *what* the system was supposed to do — alone took two years! System definition is usually the most prolonged — and sometimes most painful — part of system development. If you shorten it, you usually pay with user dissatisfaction and interminable changes during programming.

But, you *can* shorten system definition and analysis without sacrificing quality. Joint Application Design is the best known of several facilitated group processes that significantly speed up system definition and analysis.

The idea is to get the right people in a room together with a skilled neutral facilitator and, in a week or less, find out exactly what the user wants. This shortens a lengthy interviewing process and focuses on the output: a requirements spec that satisfies both developers and users.

How JAD works. Succeeding with the JAD process means paying attention to several essential characteristics of the facilitated group sessions' makeup and conduct:

• The right people. To be effective, the active user participants in the JAD process must have a good background in the application area and they must be people empowered to make decisions for the group they represent. Developer participants should be similarly empowered. The decisions they must make usually involve user details and high-level design decisions, not implementation considerations. A typical question would be "What is the correct business rule here?" At least one of the participants should be able to answer with a high degree of certainty.

When people say, "Well, I don't know. I'll check and get back to you," that contributes to the agony of prolonged requirements definition.

If you shorten development time, you usually pay with user dissatisfaction and endless changes. But there is a way to shorten time without sacrificing quality.

• The right-sized group. The group must be large enough to provide multiple viewpoints and avoid groupthink (where everyone always agrees) but small enough so each member can (and will) participate.

The optimum size for a JAD session seems to be between six and 10. While JAD is structured for sessions of that size, in my experience as a facilitator, a group of four or five is better. However, a larger group can be effective when you use techniques like storyboarding.

• Executive sponsor. The group has a high-level sponsor who can mandate cooperation by everyone involved. This neutralizes many political problems and ensures that everyone is motivated to see the session succeed.

• Group memory. The group records ideas on a group memory — typically a flip chart. Sheets are torn off the pad and taped to the wall to create a war-room atmosphere that contributes to team spirit and a sense of accomplishment.

The group memory is essential because, through its use, ideas become "ours" rather than "yours" and "mine." Other successful group-memory devices include whiteboards and magnetic symbols.

• Neutral facilitator and venue. The session is led by a neutral facilitator with no vested interest in the design's detailed content. The facilitator concentrates on the group processes and the successful completion of the tasks involved. You hold sessions in a location that does not belong to either the users or the developers.

There is an unspoken language associated with a location: For example, if it is a users' conference room, the users tend to dominate the group. A neutral location also gets people away from distractions.

• Group-smart facilitator. The facilitator is aware of what a group (as opposed to individuals or pairs) can do best and keeps the group working on those tasks.

The group must concentrate on tasks that use the group as a resource to alternately capture a lot of ideas (creation) and then organize and improve those ideas (evaluation).

Presentation (selling) and negotiation (arguing) are usually not good uses of the group resource.

• Well-defined deliverables. The session is supposed to produce a set of deliverables (usually derived from the application's life cycle), and the facilitator has a good idea of how to move the group through a logical progression to produce this set.

It is critical for the group to have a target. Of course, the specific deliverables vary by company. Examples include high-

level descriptions of the problem, module names and descriptions, and various diagrams and written specifications.

- On-line data capture. You should capture the session's discussions and results during or immediately after the group session. For most sessions, a PC-based word processor is adequate to this task, although you can better capture graphical details with a CASE tool set.
- Planning. Good results don't just magically happen without a lot of planning. IBM, which designed JAD, recommends that you spend several days interviewing the potential participants to ensure that you have the right people in each session.

Then, your first session should be a planning session lasting two to five days with top-level participants who define the functional components of the topmost layers in the application. This group further defines a series of two- to five-day design sessions that include the people who know the application details.

By convening separate sessions on each major application area in the decomposition, you can bring in people with different specialties to participate in detail sessions where they are needed most.

Overcoming JAD's disadvantages. JAD, as originally defined, does have some disadvantages. The flip chart, whiteboard, or magnetic board is "chauffeur-driven": Participants are funnelling their ideas through the facilitator or a recorder. That funnel is a bottleneck and a potential barrier to group ownership.

As practiced by IBM, JAD is geared to functional decomposition and does not aggressively build models of the system's static and dynamic data. As a result, the designers may use an inherited database design rather than taking the opportunity to produce a new one.

Also, JAD covers only a limited part of the development cycle; it does not include strategic planning or physical design. However, "JAD Across the Life Cycle," recently published by the Chicago-based IBM user group Guide (publication GPP-219), describes successes by several large organizations in applying facilitated group processes across the entire spectrum of development activities.

Because of these drawbacks, practitioners have developed several modified approaches that improve the process of facilitated group analysis and design, sometimes drastically. Here are some of the augmented approaches:

- Application blitzing, by John Palmer of Atlantic Systems Guild, produces an essential design that follows the Yourdon/Constantine/DeMarco conventions. The essential design concept is covered in *Essential Systems Analysis* by Stephen McMenamin and Palmer (Prentice-Hall, 1984).
- Accelerated System Analysis Process, by Myers & Associates of Vail, Colo., uses a facilitator and a CASE expert working with a fast PC and guarantees a complete logical application design in a week.
- Facilitated Application Specification Technique, by M.G. Rush Co. of Barrington, Ill., begins with strategic planning and integrates JAD, CASE, and structured methods to cut requirements time by as much as 75 percent, its developers claim.
- Information Engineering JAD, by consultant James Martin, uses a group Information Engineering session — supported by a CASE tool — to determine which application should be developed first.
- System Storyboarding Techniques, by my company, CASE Lab of Colorado Springs, Colo., uses storyboarding, brain writing (structured, collaborative brainstorming), and other group techniques to convert the chauffeur-driven JAD into a more collaborative process and cut time — from definition to code start — by as much as 75 percent. We support these techniques with off-the-shelf CASE software and groupware.

Few developers have been trained in group techniques, nor do they approach the idea of group analysis and design on purpose. But those who do find that it significantly improves the process.

The bottom line. From informal surveys I've taken of participants in my structured-design seminars over the past 10 years, I know that most — probably 80 percent — of all developers work in teams. Yet few have been trained in group techniques, nor do they approach the idea of group analysis and design *on purpose*. Those who do find that it significantly improves the development process.

Try it — find a group-smart facilitator and give it a shot. I think you'll be pleasantly surprised.

Gordon Bell Prize

Continued from p. 104

only 199 steps — the answer is obtained after 199 steps.

The timings reported compare 100 sequences of length 100. Splash takes 0.02 seconds to complete the search, compared to 4.7 seconds on a Connection Machine CM-2 and 6.5 seconds on a Cray 2. Because the CM-2 and Cray 2 each cost millions of dollars compared to $35,000 for Splash, this Xilinx array is an outstanding price performer running at about 77,000 VAX MIPS per $1 million.

Other entries. R. Andrew Beard and Gary B. Lamont of the Electrical and Computer Engineering Dept. of the Air Force Institute of Technology at Wright-Patterson Air Force Base in Ohio looked at solving NP-complete problems on an Intel iPSC/2 hypercube. Their entry posed two problems to the judges: Beard and Lamont used no floating-point operations and they achieved a speedup of 249 on a 32-processor machine! Unfortunately for Beard and Lamont, there was no category for the greatest superlinear speedup.

The entry from Robert Benner, John L. Gustafson, Thomas Sullivan, and Mark P. Sears of Sandia National Laboratory in Albuquerque, N.M., successfully parallelized an extremely challenging application, simulating synthetic aperture rader images, on a 1024-node N-Cube/10 hypercube. They achieved seven times the performance of a Cray Y-MP running the same application.

Andre Goforth of NASA Ames Research Center in Moffet Field, Calif., submitted a program written in Ada that extracts class definitions from an object database. The Ada runtime system was able to schedule the programmer-defined tasks efficiently enough to achieve a speedup of almost 13 on a 20-processor shared-memory machine.

Olaf Storaasli of NASA Langley Research Center solved a finite-element model of the space-shuttle booster rocket with almost 55,000 degrees of freedom in six seconds on a Cray Y-MP/8, a rate of almost 1.5 Gflops. ❖

IEEE Software will publish the rules and submission guidelines for the 1990 Gordon Bell Prize later this year.

SOFT NEWS

How software affects the world. How the world affects software.

Editor: Galen Gruman
IEEE Software
10662 Los Vaqueros Cir.
PO Box 3014
Los Alamitos, CA 90720-1264
Compmail: g.gruman
Internet: g.gruman@compmail.com

ICSE shows concerns over growing software gaps

Ware Myers, Contributing Editor

"The structure of distributed programs is similar to that of centralized programs," said Barbara Liskov, a computer-science professor at the Massachusetts Institute of Technology, "But there are some new problems." Liskov spoke at a plenary session of the 12th International Conference on Software Engineering, held March 26-30 in Nice, France. Her comments reflected the general theme of the conference: There are problems, some new and some longstanding, but all formidable.

Hardware technology progresses at an exponential pace while software-development methods move glacially. As software systems grow into the tens-of-millions lines-of-code range, reduction in error rates becomes ever more pressing. Programs that in aggregrate take hundreds of billions of lines of code are realizing their end-of-life point and must be redone — somehow.

And with all these needs to meet, better methods — even when available — transfer into practice distressingly slowly.

Gigaflops to teraflops. One example of the fast hardware pace is the development of parallel systems functioning in the range of billions of floating-point operations per second, said Stephen L. Squires of the US Defense Dept.'s Advanced Research Projects Agency. DARPA is funding several such projects.

Underlying these parallel machines are the semiconductor exponential growth curves. Squires said he expects to see 50-million transistors on a one-inch-square chip operating at 250 MHz by the year 2000, providing teraflops capabilities in parallel machines. But, he said, the catch is that all these developments make problems for software people.

One way of reducing these problems is to avoid an architecture that is "unprogrammable," Squires said. Another is DARPA's policy of devoting about 25 percent of its budget on each hardware project to software.

ESPRIT software. Similarly, in the European Community, David Talbot cited 69 projects where the primary emphasis is on software. Talbot is the head of the European Strategic Programme for Research in Information technology's Directorate General XIII, which is responsible for research in information technology and telecommunications. So far, results from 33 projects have reached the marketplace, he said.

ESPRIT research includes environments, techniques and tools, formal methods, reuse, quality, performance modeling, fault tolerance, metrics, certification, standards, knowledge engineering, and technology transfer.

Even when methods are available to relieve the many problems in software, their transfer into practice is distressingly slow.

Concern over reliability. When the software community gets to computing with billions of operations per second, even low error rates will result in large absolute numbers of errors. Even a 30-million-line software systems would have 3,000 errors at the lowest error rates being achieved today. That already may be too much in life-critical systems. As processing speed (and thus instructions executed) increases, the reliability gap will grow.

A panel on formal methods seemed agreed that better quality, including reliability, was the main benefit of applying such methods. In addition, formal methods help simplify design and uncover problems not found by other methods, panelists said. But the bad news is that there is considerable resistance to using formal methods.

Another reliability-related technology-transfer delay was implicit in Marilyn Bush's report on the introduction of formal-inspection (or Fagan) methodology used at the California Institute of Technology and the Jet Propulsion Laboratory over the last few years. These techniques were devised by IBM's Michael E. Fagan in the mid-1970s yet are only now being introduced in many organizations like JPL with difficult software needs. "By the mid-1980s, software required about 50 percent of JPL's work hours," Bush said. "By the year 2000, some software efforts will rise ... to up to 80 percent of the total." And critical flight software, which JPL works on, must be error-free.

Bush said that some companies using Fagan inspections find close to 95 percent of all defects before the test phase, have increased productivity 14 to 25 percent, and have reduced corrective maintenance by as much as 90 percent.

At JPL, 10 projects have adopted formal inspections, completing 300 inspections. "The typical number of defects found per inspection was four major defects and 12 minor defects," Bush said. By a rough estimate, each inspection saved $25,000 in later costs, she said.

Software archeology. There are hundreds of billions of lines of code in existence, said Gilles Lafue, of Andersen Consulting, at a panel. According to a member of the audience, IBM had counted 77 billion lines of Cobol code in the hands of its US customers alone. Wojtek Kozaczynski, also of Andersen Consulting, described an "old code" business environment he has seen: 240 maintenance programmers, with an average age of 25 years, working with code with an average age of 27.

Because the original developers are long gone and documentation in sparse at best, reengineering these dinosaurs in-

Continued on p. 114

AD/PRODUCT INDEX

Advertiser Index: May 1990

For display advertising, contact:

Northern California and Pacific: Roy McDonald Assoc. Inc., 5915 Hollis St., Emeryville, CA 94608; (415) 653-2122.

Jim Olsen, P.O. Box 696, Hillsboro, OR 97123; (503) 640-2011.

Southern California and Mountain States: Richard C. Faust Co., 24050 Madison St., Suite 100, Torrance, CA 90505; (213) 373-9604.

Midwest: The Kingwill Company, 4433 W. Touhy Ave., Suite 540, Lincolnwood, IL 60646; (708) 675-5755.

East Coast: Atlantic Representative Group, 349 Maple Place, Keyport, NJ 07735; (201) 739-1444.

New England: Arpin Associates, P.O. Box 6444, Holliston, MA 01746; (508) 429-8907.

Europe: Heinz J. Görgens, Parkstrasse 8a, D-4054 Nettetal 1 - Hinsbeck (F.R.G.); phone: (0 21 53) 8 99 88; telex 841(17)2153310=HJG tlx d.

Southwest/Southeast: Heidi Rex, IEEE Computer Society, 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; (714) 821-8380.

For production information, conference, and classified advertising, contact Heidi Rex or Marian Tibayan.

IEEE Software, 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1264; (714) 821-8380; fax (714) 821-4010.

Product Index

IEEE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Use the Reader Service Card to obtain information on:

- Membership application—student #203, others #202
- Perodicals subscription form for individuals #200
- Periodicals subscription form for organizations #199
- Publications catalog #201
- Standards working groups list #195
- Compmail+ international electronic mail/database brochure #194
- Technical committee list/application #197
- Chapters lists, start-up procedures—student/regular #193
- Student scholarship information #192
- Awards description/nomination forms #198
- Volunteer leaders/staff directory #196
- IEEE senior member application #204

To check membership status or report a change of address, call the IEEE toll-free number, 1-800-678-4333. Direct all other Computer Society related questions to the Publications Office.

Purpose

The IEEE Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 100,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

Members receive the acclaimed monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

Computer. An authoritative, easy-to-read magazine containing tutorial and in-depth articles on topics across the computer field, plus news, conferences, calendar, interviews, and new products.

Periodicals. The society publishes six magazines and five research transactions. Refer to membership application or request information as noted above.

Conference Proceedings, Tutorial Texts, Standard Documents. The Computer Society Press publishes more than 100 titles every year.

Standards Working Groups. Over 100 of these groups produce IEEE standards used throughout the industrial world.

Technical Committees. More than 30 TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.

Conferences/Education. The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.

Chapters. Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

European Office

Payments for Computer Society membership and publication orders are accepted by checks in Belgian, British, German, Swiss, or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by American Express, MasterCard, or Visa credit cards.

Asian Office

Payments for Computer Society membership and publication orders are accepted by checks in Japanese or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by electronic fund transfer to the Bank of Tokyo, Akasaka Branch, Toza acct. 0767956; the credit receiver is the IEEE Computer Society Headquarters Office. Payment may also be made by American Express, MasterCard, or Visa credit cards.

Ombudsman

Members experiencing problems — magazine delivery, membership status, or unresolved complaints — may write to the ombudsman at the Publications Office.

Continued from p. 111

volves first an archaeological phase. The archaeologists must reverse-engineer the old code to abstract a high-level representation that a person can understand. Then, in a forward-engineering phase, designers and programmers produce new target software.

While perfectly straightforward in theory, transforming the billions of lines of existing code is impossible on a large scale. Lafue used $3 trillion as a rough estimate of the cost of reengineering existing software. Commercial tools barely scratch the surface, he said.

A two-way street. The common conception of technology transfer is moving new technology from researchers to developers. In fact, panelists from Spain, Italy, the US, England, and Japan reported their experiences in doing just that. But Les Belady, director of MCC's Software Technology Program, went a step beyond: "Technology transfer is a two-way street," he said, "New or improved technology moves from research to development, but operational experiences moves in the other direction.

"We need increased awareness." Belady emphasized, "so that opportunities for collaboration can be identified."

ICCL focuses on language research

Carl Chang, Associate Editor-in-Chief

The International Conference on Computer Languages, held March 12-15 in New Orleans, focused on programming languages. It did not cover other types of languages, such as those for requirements, specification, design, hardware description, and protocols. The focus was on the concerns of language researchers. Some of the 105 people attending wondered why the conference had such a broad title while having such a narrow focus. For example, one participant, from South Korea, attended to learn more about hardware-description languages but found only one relevant paper.

In the conference's 32 papers and two tutorials, there were no major breakthroughs reported. The tutorials covered the Haskkell functional language and Concurrent C (which has a compile-time option to support Concurrent C++). Session topics included visual, functional, distributed, parallel, object-oriented, and logic languages, as well as design and implementation issues. Speakers did agree that these types of languages have become more mature, especially functional languages. Another trend reported was work to integrate the logic and object-oriented programming paradigms. Many papers also touched on the issues of process distribution, concurrency, exception handling, communication, and synchronization — all issues in large, distributed systems now being increasingly developed.

In planning for future conference, one concern was whether adding practitioners' participation would degrade the conference's quality and cause the predominant group of participants — language researchers — not to come. But two steering-committee members, Pei Hsia of the University of Texas at Arlington and Joseph Urban of Arizona State University, agreed that more industry participation is needed. Practitioners could help answer several questions of concern to language researchers, they said, including:

- How many languages have been invented and then discarded?
- How many languages have become a success?
- Who should determine the degree of success?

ACM conference embodies visions and worries

Paul Oman, Software Test Lab Editor

This year's ACM Computer Science Conference, held Feb. 20-22 in Washington, D.C., was a study in contrasts. From optimistic visions to critical examinations and complaints, it focused on much of what's good and what's bad in computer science today.

Geometry, not just topology. In his talk on system integration and collaboration, Les Belady, director of software technology at MCC, gave his vision of computing in the near future. The world of software has traditionally been divided into systems programming and applications programming, he said. In the future it will be divided into components (boxes) and integrators (glue). Building software will be the task of gluing together the software components. Program visualization and design recovery will play key roles in this process. Belady said.

"We should be able to recognize programs by their shape," he said, suggesting that "we should go into the area of the geometry, and not just the topology, of systems." According to Belady, this visualization is important to organizations like the US Defense Dept.'s Advanced Research Projects Agency because it lets them see the results of programming efforts, much like physicists can view the results of their experiments.

Education worries. In another presentation, James McGroddy, director of research at IBM T.J. Watson Research Center, lamented the state of US education and research policies. "We have a state of crisis in the American elementary and secondary schools," McGroddy said. He claimed that this still exists, "despite the fact that we have increased our funding 300 percent. ...

"We need to completely revamp our education system, making education a higher priority," he said. He suggested that industry could help by aiding the development of teaching methods, showing teachers how to use computer technology, developing a measurement for the quality of the educational process, and supporting women and minority students.

At the ACM Special Interest Group on Computer Science Education's award lecture, a panel of accomplished computer scientists discussed the formulation of Curriculum '68, the earliest national model for a computer-science curriculum. But instead of focusing on how computer science has changed and addressing the problems that academia faces today, the SIGCSE panel's discussion centered around the 20-year-old curriculum and its inception. It ignored the fact that Curriculum '88 has been stalled in committee for two years. Instead, details about the progress of the joint ACM/IEEE curriculum was relegated to

Continued on p. 117

SOFTWARE **PRODUCTS**

Product news and reviews to help you choose.

Editor: Sorel Reisman
Management Science Dept.
California State University
Fullerton, CA 92634
Compmail+: s.reisman, Bitnet: lreisman@csuf
Internet: s.reisman@compmail.com

CD-ROM: Technology whose time is coming (soon)

Sorel Reisman, Software Products Editor

Multimedia computing, a term coined by Apple Computer, has really arrived, but its potential will likely be realized on the platforms of IBM and IBM-like computers.

February saw the announcement of several extremely important multimedia products for the IBM families of computers:

- Early in the month, IBM announced its M-Motion products, a Micro Channel board with support software that accepts input from conventional video sources like camcorders, VCRs, and videodisc players. With OS/2, the Presentation Manager displays digitized, full-motion video in resizable, moveable VGA windows on the fly. It can also play accompanying audio. Supporting software also lets you still-frame the video in a window and capture the digitized image and store it to a file for subsequent display. These products represent IBM's transition from its analog-only, videodisc-based Infowindow System to a hybrid analog/digital product that will eventually evolve into fully digitized full-motion video and audio.
- An equivalent product available from Videologic offers a superior function set for both ISA (the traditional PC) or MCA machines — at the same price as IBM's products. Videologic's products also run under Microsoft Windows.
- Both IBM and Videologic also offer Infowindow emulation software to soften the blow to users who have made significant investments in Infowindow videodisc materials. Unfortunately, Infowindow's touch-screen function is not yet supported by PS/2 products.

Unsubstantiated rumors circulating on the floor of the Orlando meeting of the Society for Applied Learning Technologies and later at the Microsoft CD-ROM Conference in San Francisco suggested that IBM would soon be announcing a modular touch-screen attachment for PS/2s. Another product that will also soon be announced is a PS/2 with an integrated CD-ROM drive.

While there are other manufacturers that offer PC products with integrated CD-ROM drives (especially the products from Headstart Technologies, a division of North American Philips), this feature represents a major commitment by IBM: its (re?)entry into the consumer market.

DVI is basis for multimedia move. This strategy becomes more apparent in the context of other announcements that were made at the Microsoft CD-ROM Conference, an exposition that views analog video as simply a bridge to products that support only digital forms of text, graphics, audio, and full-motion video. IBM's commitment to this is via the enabling technology of digital video interactive.

Apple may have coined the term 'multimedia,' but it is in the IBM world where this technology is being realized.

DVI, as it is called, was originally developed at the RCA Sarnoff Labs but is now owned by Intel (with a substantial financial commitment from IBM). The storage requirements for digitized audio and video are significant, so DVI is a CD-ROM-dependent technology. Last year, Intel, IBM, and Microsoft announced a cooperative effort that would result, by 1992, in the availability of yet-to-be developed Intel DVI chip sets on IBM PS/2 motherboards supported by Microsoft operating systems. Interim milestones were also announced then.

This year's CD-ROM conference highlighted the progress made toward those objectives. Intel announced the availability this month of the Action Media Capture adaptor card to develop fully digitized DVI applications (both hard-disk- and CD-ROM-based). Intel also announced the Action Media Delivery Adaptor Card to deliver the multimedia applications. Intel will sell both ISA and MCA versions of the boards; IBM will sell only the MCA versions. Each card costs about $2,000, from either company. These reflect major milestones that have significantly reduced the entry cost for DVI development and delivery.

Accessible development tools. In addition to the breakthroughs in hardware, application development has also been simplified since 1989, when the first multimedia-authoring tools required a proficiency in C. Now, however, higher level authoring languages and tools are available for application developers. One notable example is Authology:Multimedia from CEIT Systems, a company with extensive experience in interactive videodisc authoring.

Moving to the consumer market. However, more important than all these new products is the success of the three partners in meeting their first DVI objectives. With the ultimate goal of providing chip sets on the motherboard, it is clear that the target costs of multimedia computing will be drastically reduced by 1992. In fact, projected markets for DVI-based multimedia computing products will let the partners move into the consumer market. At the Microsoft conference, the incremental target price for multimedia-based PCs was reported to be only $500 above the price of a nonmultimedia PC.

At a press conference hosted by the DVI partners, Michael Quinlan, IBM vice president and assistant general manager of business development for personal systems, told me that IBM intends to enter the consumer market in the next two years. He was strongly affirmative. If this does happen, it is logical that IBM will announce CD-ROM-based PS/2s to support the DVI storage requirements of multimedia computing.

Multimedia companies

The following companies are involved in the CD-ROM technologies described in the accompanying text. For information on them, circle the appropriate numbers on the reader-service card.

American Interactive Media	50
Apple Computer	51
Avtex Research	52
CEIT Systems	53
Disctronics	54
Headstart Computers	55
IBM	56
Intel Princeton Operation	57
Knowledge Access International	58
Microsoft	59
Optical Disc Corp.	60
Philips Consumer Electronics	61
Philips International Interactive Media Systems	62
Pioneer Communications of America	63
Sony Corp. of America	64
Videologic	65

Competing technologies. A contending technology in the multimedia arena is compact disc interactive, a similar CD-ROM-based technology being promoted by Philips, Sony, and the Japanese electronics giant, Matsushita. Announced in 1986 as being imminently available but slated for the consumer market, CD-I has consistently failed to meet its development and delivery schedules. Despite that, CD-I has continued to attract a measure of attention, particularly at the Microsoft CD-ROM conferences.

This year, American Interactive Media, Philips's main promoter of this technology, presented a vast array of applications ready to ship with the CD-I players. (And, of course, the real success of any of these efforts will depend on application — read "software" — availability.)

A CD-I player is designed as an add-on compatible device for audio CD players and home TVs. It will be very interesting to see if the Philips' consumer strategy is ultimately successful, or whether the IBM/Intel/Microsoft PC strategy can be transformed into successful home products. The DVI team has not demonstrated any success in that market.

Also, DVI is technically exciting, but there are essentially no applications available for it. CD-I will be shipping next year, far ahead of the DVI team's ability to seed or produce an equivalent number of consumer (or even nonconsumer) applications.

Another possible, and maybe even likely, scenario is a head-to-head battle between two different formats, not unlike the Beta versus VHS videotape battle.

The Sony factor. Most noticeable about the CD-I announcements was Sony's participation in the venture. At the conference, Sony demonstrated the potential effect of its recent acquisitions of Columbia Pictures and CBS Records. CD-I applications are replete with nifty audio and video materials originally produced by those acquired companies. Such materials — and the ability to produce them — are a key element in the design of multimedia applications (software).

Another key element essential for the success of such a product set is a company's ability to bring the products to the consumer (the user). Sony has built-in distribution channels for consumer products, something the IBM/Intel/Microsoft DVI team noticeably lacks.

Sony made two other really interesting announcements that significantly enhance the functions available in CD-I, bringing CD-I closer to the functional capability of DVI:

First, Sony is the banner carrier for adherence to the ISO 9660 CD-ROM standard. Sony is committed to maintaining and extending that standard while continuing to provide compatibility for CD audio as well as for the main hardware platforms offered by IBM (and its DVI colleagues), Apple (which has no colleagues), and the CD-I group.

Last year, Sony announced CD-ROM XA, an ISO 9660 extension to file structures that permits appropriately equipped CD-ROM drives to display conventional computer output while also allowing the simultaneous output of interleaved audio. This year, Sony extended that standard to include interleaved graphics. Sony's ultimate objective seems to be to provide full-motion video from audio-CD-compatible media while maintaining full backward compatibility with existing platform manufacturers' competing technologies.

A second interesting announcement is Sony's new WORM drive that will write CD-ROMs that are "100 percent compatible" with existing CD-ROMs and can be read by any CD-ROM drive. The CD-ROM writer will be available in two models: One model will be able only to write one disc at a time; the second model will be able to write to as many as 32 discs simultaneously.

This second machine will be priced in the $30,000 range and is targeted for in-house CD-ROM publishing. Many companies, like Apple, are already using CD-ROMs for in-house distribution of product literature, technical documentation, etc. The Sony CD-ROM publisher will greatly reduce the costs of these activities.

The single CD-ROM writer will be priced in the $5,000 range. It offers DVI and CD-I application developers the opportunity to easily test newly developed applications on a preproduction-level CD-ROM. Today, developers must use 700-Mbyte (or larger) hard-disk drives to emulate the capacity of a CD-ROM. The new CD-ROMs can also be used as the source medium for CD-ROM premastering and eventual large-volume production, thereby eliminating the tape drives now needed to do this.

However, for DVI-developed applications, the quality of the video generated in this desktop fashion will be inferior to the quality of video offered by Intel's tape-based, off-line compression services.

Vendors catch up. An interesting dimension to the Microsoft CD-ROM Conference has been the dichotomy between session announcements and exhibited products. In the past, session announcements focused on such matters as DVI, CD-I, and CD-ROM XA, while exhibitors have demonstrated a variety of conventional CD-ROM drives, CD-ROM databases, and enabling services and processes. That dichotomy was much less evident this year with many exhibitors displaying real products and services that support some of the newer technologies.

For example, there is no doubt that for the foreseeable future CD-ROM drives will be an integral part of multimedia computing. But if you want to buy a non-Japanese CD-ROM drive, you will have to travel to an unknown (and probably distant) planet to find one. And, except for one exception, all the drives are practically identical in function and price.

That exception is Pioneer Communications' Minichanger, which can manage as many as six CD-ROMs, each as a separate MS-DOS or Macintosh drive, for

the incredibly low price of $1,295. I can't imagine why anyone would buy a single CD-ROM drive for $800 to $1,000 with this product now shipping. Some of Pioneer's competitors were aghast when I told them about this.

Unlike other trade shows, most of the real action at the CD-ROM conference takes place during the formal sessions. But one general meeting each year is a two-hour industry-announcement session where vendors have five to 10 minutes to present the highlights of their products and services. Even while the giants of the emerging multimedia industry touted the future, smaller firms continued to build the industry base by marketing their CD-ROM hardware, databases, retrieval software, and manufacturing services.

Recommended products. Some notable examples include companies like Avtex Research, which provides complete concept-to-implementation assistance for companies wanting to develop CD-ROMs. At the other end of that spectrum are products that you can use in a do-it-yourself desktop-publishing mode.

For example, if you are new to CD-ROM and want to try a low-cost and low-risk experiment, I strongly recommend Knowledge Access International's Disk Publisher and Retrieval Systems. For around $1,000, this PC-based package can let you easily experiment with the methodologies of text and image indexing and even let you customize retrieval software. And the best part of these products is that you need only a PC with a hard drive to test out your concept. But this package is not just for fooling around. If you like what you get, you can use your work for full-scale disc production.

If you do want or need to move into the volume pressing of discs, services like those offered by companies like Disctronics of Anaheim, Calif., were at the show advertising their services.

In fact, if you are searching for your own volume disc-pressing facility, you can window-shop from companies like Optical Disc Corp., which was founded by several of the engineers that pioneered the techniques used by almost all optical disc pressers. Its customers include many of the major disc manufacturers and several film and television editing houses. Rumor has it that the company is interested in partnering a disc-pressing installation with an educational customer that might view optical technology as the "new papyrus."

Glimpsing the future. The Microsoft CD-ROM Conference is evolving into an annual event that in many ways parallels the importance of Fall Computer Dealers Exhibition. Just as Comdex is a forum for "products" that will be marketed in the following 12 months, the Microsoft CD-ROM Conference is a forum for "concepts" that will be developed in the next 12 months with products that may follow in another 24 to 36 months. In this regard, the conference focused on the evolution of computing through 1990, and onto the end of this century.

Today, it seems that we are moving into a decade that will see the user interface continue to evolve so it has full-motion digitized video with exceptionally high-quality audio as a key part.

You could see testimonials to this in the major manufacturers' multimedia presentations. Integral to this is the emerging role of the video industry — the source of the new digitizable data. Although conference participants were mostly concerned with technical issues, the influence of Hollywood in the new technology was very apparent.

Consider, for example, that the chairman of American Interactive Media, Gordon Stulberg, is an alumnus of the executive ranks of the motion-picture and recording industry. Or consider Sony's use of Columbia Pictures' film footage and CBS Records' audio in its new CD-ROM applications. Or of Children's Television Workshop's CD-I applications that use *Sesame Street* animation. Or consider the surprise presence at the conference of television-pioneer Walter Cronkite!

Something very dramatic is happening to personal computing. And while controversies about MCA, EISA, and ISA bus architectures, or Windows versus Presentation Manager, or OS/2 versus Unix, are integral to these developments, it's still fun sometimes to have a user's view of where all this is leading. The Microsoft CD-ROM Conference certainly supplies that view.

SOFT **NEWS**

Continued from p. 114

a status report given during the concurrent sessions later that day. The effort remains stalled.

Competitiveness and cooperation. Part of the US's problem in promoting technology, IBM's McGroddy said, was "the lack of a [presidential] cabinet-level position [that] is responsible for the success of technology — an incredibly important position." Noting that the US has cabinet-level spokesmen for industries like tobacco (through the Agriculture Dept.) and energy (through the Energy Dept.), McGroddy asked, "Where's the spokesperson for technology?"

He also warned of the dangers of become complacent about US computer technology and suggested that "others," referring to the Far East, "are running away from us." He likened US technology to a ship that needs continual movement: "A boat can also sink by rusting around the waterline, by sitting around in the water. We need to worry about that, not worrying about running into an iceberg, but sinking below the waterline."

McGroddy's call for cooperation was echoed the next day by Tadao Ichikawa, an information-science professor at Hiroshima University. In his talk, Ichikawa stressed the need for cooperation between what he called a highly productive Japan and a highly creative US. Describing Japan as productive but not creative, he said Japan is best at "turning ideas into applications." Looking into the future, Ichikawa said, "we expect the West to become more creative; we expect the Japanese to become more productive." He then asked, "Why can't we be more cooperative in producing a better performance?"

Inconsistent compilers. William Kahan, a computer-science professor at the University of California at Berkeley who won this year's ACM Turing Award for his work spearheading the IEEE 754/854 standards for floating-point arithmetic, wowed the audience with examples of how Cray and Sun compilers, which have incompatibilities with the standard, can generate incorrect results even though the source-level mathematics is correct.

Referring to the Sun compiler, Kahan said the problem stems from the fact that "technicians competent in floating-point were prohibited from influencing the compiler writers, from a decision on high." He said that very few compilers conform to the standards, listing only Borland International's Turbo C for MS-DOS PCs and Lightspeed C for the Macintosh as shining examples of compatible C compilers.

BOOK REVIEWS

Spanning software's diverse theories, practices, and philosophies.

Editor: Michael Lutz
Rochester Institute of Technology
c/o 29 Concord Pl.
Pittsford, NY 14534
Compmail: m.lutz
Internet: mjlics@ultb.isc.rit.edu

Doing the right thing, and doing things right

The Macintosh Way by Guy Kawasaki (Scott, Foresman, Glenview, Ill., 1989, 210 pp., $19.95)

D. Brent Chapman, Chapman Consultants

Guy Kawasaki's new book, *The Macintosh Way*, should be required reading for the managers and executives of any company in the microcomputer software industry, particularly those producing Macintosh software. The book, subtitled "The Art of Guerilla Management," is a funny, wickedly pointed guide to Kawasaki's philosophy of "doing the right thing and doing things right," which he calls "the Macintosh way."

Kawasaki is a veteran of Apple's Macintosh effort, and is now the president of Acius, which writes database software for the Macintosh. He started at Apple in the Macintosh division shortly before the Mac was introduced to the public, with the title of "software evangelist" and the seemingly impossible mission to "get software" for the Macintosh. Two days before he left Apple to form Acius, he was promoted to director of software-product management. He thus speaks with the knowledge of an insider with a ring-side seat throughout the unfolding of the Macintosh phenomenon.

What he speaks about is not so much *what* Apple did in developing and marketing the Macintosh (although there's plenty of that), but *how* and, most importantly, *why*. In his treatment of various aspects of development and marketing, he constantly comes back to two key themes: doing the right thing and doing things right. After a short introduction and history lesson, the remainder of the book is divided into a two sections, one each on these central topics.

In the section on doing the right thing, Kawasaki tells how to create a first-class company by creating the right work environment, great products, exceptional support, and innovative marketing. Kawasaki emphasizes that "people are the most important asset of a company" and that "the right environment is created by passion and sustains action, risk-taking, and fun."

His second aspect of doing the right thing is having great products, and he explains what he believes are the qualities of great products, the right product-development process, and the right development people.

The third aspect is support, beginning with a simple case study of "exceptional support" (shopping at Nordstrom's department store), continuing with examples of "the way it shouldn't be (but usually is)," and concluding with "the way it should be (but usually isn't)."

Kawasaki pulls no punches when presenting his opinions of Apple, its competitors, or any of the other objects of his criticism.

The final aspect is the marketing of technology (finding the right people and getting the right information into their hands, and letting them sell themselves), as opposed to the technology of marketing (today's modern advertising and PR campaigns).

The second key to success for a "Macintosh way" company, according to Kawasaki, is doing things right. This involves such diverse things as taking advantage of user groups, causing products to be *pulled* through distribution channels (rather than pushed), working with the trade and general press, working with your parent company, giving good demos, making solid presentations to harried managers, using trade shows effectively, and driving your competitors crazy. He devotes a chapter to each of these topics.

Another chapter covers the unorthodox concept of evangelism (getting your customers and suppliers to do your work for you, since they're sold on your vision).

Finally, there's a chapter entitled "The Macintosh Guide to Dating and Marriage," which has little to do with the subject of making a successful software company but is lots of fun.

The book is light in tone, humorous, and easy to read, in spite of the wealth of ideas and information it contains. The numerous footnotes include many of the most humorous comments in the book; for example, when he first mentions the Laserwriter printer that let Apple help create the desktop-publishing market and the Postscript language from Adobe that controls the Laserwriter, Kawasaki uses a footnote to define Postscript as "the technical term for a lucrative royalty stream from Apple to Adobe."

Also scattered throughout the text are many thought exercises designed to make you realize the significance of what Kawasaki is saying. For example, to drive home the effects of desktop publishing, the exercise is "Connect a daisy-wheel printer to a Macintosh. Create a newsletter with Pagemaker [what Kawasaki calls "an act of God specifically intended to save Apple"]. Print the newsletter."

Kawasaki pulls no punches when presenting his opinions of Apple, its competitors, or any of the other objects of his criticism. Occasionally, he descends into a diatribe that the book would be better off without, but on the whole these brief digressions are more than balanced by the quality and content of the rest of the book.

All in all, the book is informative, entertaining, quick to read, and well worth its price and the time to read it for anyone interested or involved in microcomputer software development and marketing.

RS 100

Text-compression book is anything but dense

Text Compression by Timothy C. Bell, John G. Cleary, and Ian H. Witten (Prentice-Hall, Englewood Cliffs, N.J., 1990, 318 pp., $43.20)

Peter G. Anderson,
Rochester Institute of Technology

The length of compressed text should be equal to its entropy, which depends on its probability, which assumes estimates arising from models deriving from the study of similar pieces of text and, perhaps, of earlier parts of the current text. My Pascal style differs from Shakespeare's poetic style; chapters in a book differ; and the end of an article may differ from its beginning.

You can represent models with simple statistical tables (like one reflecting that 9.76 percent of typical English text is the letter "e"), Markov or finite-state models (reflecting that the letter "u" almost always follows "q"), or, for data of a different kind, that human speech nearly follows a behavior that permits linear extrapolation and that natural scenes contain objects that are, by definition, continuous in extent. These models lead directly to data-compression schemes like Morse code, Grade 2 Braille, linear predictive encoding of speech, and quadtrees of images.

This is the major theme of Timothy Bell and colleagues' *Text Compression*: Text compression depends first on modeling the statistics of the phenomenon and then on encoding a given piece of text against the model. A model may be determined and frozen once and for all or it may adapt itself to the statistics of the current piece of text.

Morse code, with its terrible encoding of numeric data, and the bulk of well-known (the authors call them ad hoc) models are examples of the former. Breakthrough text-compression work of the last 15 years (including work by the book's authors and by J. Ziv and A. Lempel) are of the adaptive sort, where the encoder's and decoder's code books evolve based on the statistics of the text as it is encoded and transmitted.

In the adaptive approach, if the text turns out to be other than English (say, Fortran or Finnish), its regularities and peculiarities will be duly noted and cleverly encoded, with very little penalty extracted for not having a static model optimized for the new language. In fact, there will often be benefits because individual styles will be recognized and accounted for in the adapting model. (The authors have carefully quantized these observations and have provided the needed formulas, theorems, and proofs.)

The book presents optimal coding theory, covering the developments from Shannon-Fano, through Huffman, to arithmetic coding, which can overcome the limitations of the first two (stemming from rounding all encodings up to an integral number of bits per message). In arithmetic coding, you encode messages as subintervals [x,y) of [0,1) and transmit them in the form of a single representative number z in [x,y). The authors show with pseudo-C code how you may implement this. The example includes the adaptive steps, which also overcome some of the Shannon-Fano and Huffman schemes' requirement to maintain data structures for sorting messages by frequency.

The style of this book is very easygoing, although the authors have not skipped the necessary mathematics or rigor. Particularly valuable are the presentations of this technology's origins, its continued growth and development, why it continues to be necessary (actually, integral) to the continued developments of computer processing power, memory, and communication, and the components of trade-off evaluations for various approaches. The book has both a glossary and an extensive list of references covering the classics and many recent (from the last 15 years) entries.

This book is a volume in the Prentice-Hall's so-called "advanced rereference series," but give it to your students anyway! The authors appear to have had fun writing this book, and I had fun reading it. **RS 101**

The calculus of concurrency

Communication and Concurrency by Robin Milner (Prentice-Hall, Englewood Cliffs, N.J., 1989, 260 pp., $40)

Andrew Kitchen,
Rochester Institute of Technology

This book presents the most recent version of a semantic theory of processes originally introduced in Robin Milner's influential *A Calculus of Communicating Systems* (Springer-Verlag, 1980). In the years since the original publication, Milner and others have strengthened and refined this theory, which Milner now prefers to call "process calculus," and have explored its relationships with other semantic models of concurrency, notably, Anthony Hoare's CSP, the Meije language of Gerard Berry and others, and Edward Brinksma's LOTOS.

Milner's goal in developing the process calculus was to find a semantic theory for which communication is the primitive notion. Hoare, working at about the same time, developed a model based on the same idea (see *Communicating Sequential Processes*, Prentice-Hall, 1985). Milner's and Hoare's work, although superficially similar, follow very different paths.

Milner bases his model on pairwise communication between processes, one process being the sender and the other the receiver. Hoare's approach allows synchronization between any number of processes.

However, the most significant difference between the two theories lies in their approach to semantic equivalence. Milner's theory uses the notion of *bisimilarity*. Two processes are bisimilar if, to an observer, their (external) behavior always appears to be the same. Hoare's model is based on what he calls failures equivalence: Two processes are failures-equivalent if their sets of possible execution traces are identical and any environment that offers one the opportunity to deadlock after generating a given trace will provide the same opportunity to the other after it has generated the same trace.

The distinction between these is subtle. Hoare's is the weaker; you can characterize it as the weakest equivalence that never equates a process having the potential for deadlock with one that cannot deadlock. On the other hand, you can characterize bisimilarity as the strongest equivalence based on observable behavior.

I strongly recommend that you read Milner's new book hand in hand with Hoare's 1985 book. Hoare's presentation is a little more accessible to a reader unfamiliar with the field, but both are well written and do not require any more background in formal methods than that acquired in a typical undergraduate program in computer science. They offer a unique experience: the opportunity to listen in on a dialogue between two major figures in the field.

Milner begins his book by analyzing some simple concurrent systems. His goal is to show how "communication and concurrency are complementary notions" and to provide the rationale for the small set of operators from which his calculus is built. Unlike parallel composition, the sequential composition of processes is not a primitive operation in his theory

(nor in Hoare's, either). This is an intriguing trend in the theory of parallelism.

The second chapter begins with a formal description of process calculus. Milner then presents a transitional semantics that specifies how processes evolve as they participate in synchronization events.

Subsequent chapters contain a careful development of the concept of bisimilarity. Milner proves some partial results on the soundness and completeness of a set of equational laws for observation congruence (the weakest congruence consistent with bisimilarity) and illustrates the power of the theory with some nontrivial examples.

He gives the semantics for a simple imperative concurrent programming language by translating it into the process calculus, which lets him both illustrate the specification of variables and procedure calls in his calculus's purely synchronization-based semantics and model the nondeterminism inherent in concurrent execution in a shared-memory environment.

The remaining chapters cover other process calculi (including the work of Hoare and his associates), the extension of the calculus to handle logical as well as algebraic properties, and the issues in designing concurrent systems that can be guaranteed to behave deterministically.

Milner's book is appropriately paced for a graduate-level course or for an advanced undergraduate seminar in the theory of parallelism. Numerous exercises are scattered throughout the text, and they are placed to engage the reader actively in the development of the material. This is a stimulating introduction to an important model for the semantics of concurrency. **RS 102**

Interface specification made simpler

IDL: The Language and Its Implementation by John R. Nestor, Joseph M. Newcomer, Paola Giannini, and Donald L. Stone (Prentice-Hall, Englewood Cliffs, N.J., 1990, 560 pp., $35)

Warren R. Carithers,
Rochester Institute of Technology

One of the thornier problems facing developers and maintainers is the task of ensuring that groups of interacting programs share a common view of the universe in which they operate. Compounding this are the facts that these programs may be very large and may have been developed by teams of programmers working over long periods of time. Under these circumstances, some formal specification method to describe the interface between modules is extremely useful.

The Interface Definition Language was developed to address just these issues. IDL provides a specification mechanism to describe data structures used by groups of cooperating programs. You can use a collection of tools to automatically generate the portions of such programs that deal with the shared data structures, virtually eliminating interface incompatibilities.

John Nestor and colleagues' *IDL: The Language and Its Implementation* is an attempt to present all aspects of IDL Version 3 in a useful, understandable format.

One initially daunting characteristic of this book is the sheer volume of information contained between its covers. In one book, the authors provide an introductory treatment of the language, a full reference manual for IDL, a complete implementation presentation, a guide to the use of a widely available IDL implementation, and a formal denotational semantic definition of the language. In all, 27 chapters and seven appendixes greet you upon opening this book.

The first section provides an introduction to IDL. The authors present major concepts of the language with simple examples. They then provide a complete example of the use of the Mini IDL implementation, which helps reinforce the relationships between the various IDL constructs and helps fill in some of the gaps in the necessarily light treatment of some language features in the introduction.

The book's second section provides a complete IDL reference manual. Each chapter begins with a short paragraph summarizing the IDL component described in the chapter. As with most reference documents, this part of the book is not intended for casual browsing but for use in bits and pieces as the need for information arises. Each chapter is concise yet complete.

Virtually anyone using the IDL language will find a significant amount of useful information here.

The third — and largest — section, which covers the Carnegie Mellon University/Tartan implementation of IDL, may be the most interesting part for many readers. The authors provide a look at the inner workings of a team designing a large software product: the IDL system. In the authors' words, this section shows the collection of "well-documented techniques, unwritten folklore, and on-the-spot inventions we used to build our particular implementation" and as such brings a human element to what is a case study in software engineering.

The authors describe the decisions made throughout the design process and often appraise those decisions critically, providing a remarkably candid view of the development process. The anecdotal evidence and conclusions provided in chapter 21 ("Rationale and Retrospective") should especially strike familiar chords in anyone involved in the design of a large software product.

The fourth section provides an implementation and usage guide the Mini IDL system. It contains much valuable information for both users of Mini IDL and people interested in IDL implementations in general.

The final section is a formal definition of IDL through the use of denotational semantics. The treatment is self-contained; some exposure to denotational description will help you understand this section, but it is not mandatory. Formal language theorists and those considering developing IDL implementations may find this section to be more useful than the typical IDL user will. The material is presented in a logical sequence that makes it accessible even to those readers who do not have significant experience with formal semantic definitions of this type.

Seven appendixes provide information of widely varying types, from a full Backus-Naur-form description of the language and initial coverage of what a standard implementation of IDL should contain to musings on the history and possible directions of IDL. In addition, users of IDL Version 2 will find appendix D to be particularly helpful in easing the transition to IDL Version 3.

This book deals with IDL Version 3 in enough ways that virtually anyone using the language will find a significant amount of useful information. In addition, the authors provide valuable insight into the development process that should be useful even to those practitioners not directly involved with IDL. **RS 103**

BUILD BETTER SOFTWARE

With these Breakthrough Guides from McGraw-Hill

Available Now at These Fine Stores

ALABAMA

Walden Software
Madison Square Mall
Huntsville, AL 35805

CALIFORNIA

Scholar's Bookstore
El Segundo, CA 90245
(213) 322-3161

OPAMP Technical Books
Los Angeles, CA 90038
(213) 464-4322

Stacey's Bookstore
Palo Alto, CA 94301
(415) 326-0681

Stanford Bookstore Palo Alto
Palo Alto, CA 94301
(415) 327-3680

San Diego Technical Books
San Diego, CA 92111
(800) 346-0071

Stacey's Bookstore
San Francisco, CA 94105
(415) 421-4687
FAX (415) 777-5017

Computer Literacy
San Jose, CA 95131
(408) 435-1118

Stanford Bookstore
Stanford, CA 94305
(415) 329-1217

Computer Literacy
Sunnyvale, CA 94086
(408) 730-9955

B. Dalton Bookseller
Torrance, CA 90503
(213) 370-5735

BookStar
University City, CA 92122
(619) 457-7561

COLORADO

Auraria Book Center
Denver, CO 80204
(303) 571-0265

Tattered Cover Book Store
Denver, CO 80206
(303) 322-7727
(800) 833-5327

R & R Technical
Littleton, CO 80160
(303) 794-4518

United TechBook Company
Longmont, CO 80501
(303) 651-3184
(800) 247-4808

D.C.

B. Dalton Bookseller
1776 K Street, N.W.
Washington, DC 20006

Reiter's Scientific
Washington, DC 20006
(202) 223-3327

FLORIDA

Walden Software
Merritt Square Mall
Merritt Island, FL 32952

Bookstop
Miami, FL 33156
(305) 598-7292

Walden Software
Palm Beach Mall
West Palm Beach, FL 33401

GEORGIA

Engineers Bookstore
Atlanta, GA 30313
(404) 892-1669

Oxford Bookstore
Atlanta, GA 30305
(404) 262-3333

INDIANA

Walden Software
Lafayette Square
Indianapolis, IN 46254

ILLINOIS

Kroch's & Brentano's
Chicago, IL 60603
(312) 332-7500
Outside Chicago
(800) 833-BOOK
FAX (312) 332-6074

LOUISIANA

BookStar
New Orleans, LA 70130
(504) 523-6411

MASSACHUSETTS

Boston University Bookstore
Boston, MA 02215
(617) 236-7415

B. Dalton Bookseller
Braintree, MA 02184
(617) 848-9542

SoftPro
Burlington, MA 01803
(617) 273-2919

M I T Coop
Cambridge, MA 02142
(617) 491-4230
FAX (617) 621-0856

Quantum Books
Cambridge, MA 02139
(617) 494-5042
FAX (617) 577-7282

B. Dalton Bookseller
Framingham, MA 01701
(508) 872-1264

MARYLAND

Maryland Book Exchange
College Park, MD 20740
(301) 927-2510

NORTH CAROLINA

Intimate Book Shops
Chapel Hill, NC 27514
(919) 929-0411

NEW JERSEY

McGraw-Hill Bookstore
Hightstown, NJ 08520
(609) 426-5749

Walden Software
Garden State Plaza
Paramus, NJ 07652

Princeton University Store
Princeton, NJ 08540
(609) 921-8500

NEW YORK

Walden Software
Hudson Valley Mall
Kingston, NY 12401

Barnes & Noble
105 Fifth Avenue
New York, NY 10003

McGraw-Hill Bookstore
New York, NY 10020
(212) 512-4100

Waldenbooks
57 Broadway
New York, NY 10006

Walden Software
Poughkeepsie Galleria
Poughkeepsie, NY 12601

Total Information Inc.
Rochester, NY 14613
(716) 254-0621
(800) 876-4636
FAX (716) 254-0153

OHIO

Walden Software
Rolling Acres Mall
Akron, OH 44320

University Bookstore
Cincinnati, OH 45221
(513) 556-1292

Borders Book Shop
Columbus, OH 43220
(614) 451-2292

OREGON

Powell's Technical
Portland, OR 97209
(503) 228-3906
(800) 225-6911

TENNESSEE

Davis-Kidd Booksellers
Nashville, TN 37215
(615) 385-2645

TEXAS

Bookstop
Austin, TX 78752
(512) 453-7297

University Co-op
Austin, TX 78705
(512) 476-7211

Bookstop
Dallas, TX 75230
(214) 363-5744

PRNT Bookshop at Infomart
Dallas, TX 75207
(214) 746-3625

Taylor's Technical Books
Dallas, TX 75240
(214) 239-TECH

Bookstop
Houston, TX 77098
(713) 529-2345

Brown Book Shop
Houston, TX 77002
(713) 652-3937
FAX (713) 652-1514

Bookstop
San Antonio, TX 78230
(512) 697-0588

VIRGINIA

Walden Software
Manassas Mall
Manassas, VA 22110

Walden Software
Tysons Corner Center
McLean, VA 22103

Walden Software
Patrick Henry Mall
Newport News, VA 23602

WASHINGTON

Tower Books
Bellevue, WA 98004
(206) 451-1110

Elliott Bay Professional
Seattle, WA 98104
(206) 624-6658

University Bookstore
Seattle, WA 98105
(206) 634-3400

WISCONSIN

Harry W. Schwartz Bookshop
Milwaukee, WI 53202
(414) 274-6400
Outside Milwaukee
(800) 552-READ
FAX (414) 276-4577

In Canada, please contact McGraw-Hill Ryerson at (416) 321-7610.

Your Computer Series Publisher

McGraw-Hill, Inc.
Professional & Reference Division

How much longer can you afford to ignore your lack of control over...

- IMPACT ANALYSIS
- CHANGE CONTROL
- VENDOR CODE MANAGEMENT
- SOFTWARE AUDITS
- MIGRATION CONTROL
- PRODUCTION CONTROL
- MAINTENANCE BOTTLENECKS
- ADA DEVELOPMENTS
- DOD-2167A ADHERENCE

If you have these PROBLEMS... Softool has the SOLUTION!

It's called ***CHANGE AND CONFIGURATION CONTROL (CCC)***.
Softool pioneered this technology, and now the CCC product family sets the standard against which all others are measured.

Available for Digital (VMS, ULTRIX and RISC), UNIX, IBM (MVS and VM), and various other platforms.

SOFTOOL CORPORATION

"The Change and Configuration Management Experts"

digital™ Independent Software Vendor

340 South Kellogg Ave., Goleta, CA 93117
Telephone: 805-683-5777
Telex: 658334 Fax: 805-683-4105

Offices in Chicago, Dallas, Los Angeles, New York, San Francisco and Washington, D.C.
Trained representatives in Australia, France, Hong Kong, Israel, Italy, Malaysia, Singapore, Spain and the United Kingdom

Reader Service Number 18