SELF-ORGANIZING SYSTEMS

The attention of many software developers has shifted to multi-user systems. These are not your multi-user systems of old, where a host controls a multitude of terminals, or even your new multi-user database, passively serving the ad hoc requests of several users, but distributed systems where individual users and applications act independently to produce results in concert.

These systems are too complex and contain too many local interactions (both inside and with the outside) to be centrally managed. Instead they must be coordinated and self-organizing, with access to distributed information -- and underneath it all, good plumbing. Self-organization is a property of systems, not of their components, but it results from the interactions of those components.

Self-organizing systems aren’t new: Contention schemes such as multi-processor scheduling algorithms and Ethernet are ubiquitous examples of non-hierarchical, coordinated systems.

Yet only a few commercial systems vendors have studied their theoretical underpinnings -- the field of self-organizing systems. This makes sense: Such systems self-organize whether developers intend them to or not. Traffic has been flowing (more or less) ever since the young hominid passed the old one on a jungle path. The issue is not the existence of self-organizing systems, but our ability to build better ones, from traffic grids and social safety nets to complex interacting network applications and groupware.

In proper self-organizing fashion, in this issue we explore some examples of self-organizing computer systems and the necessary platforms first, and only afterwards examine the theories underlying their behavior. Since vendors’ goal is to make the intricacies invisible to users and even to application developers, most current implementations are embedded systems rather than applications. Yet

ENJOY THE FOURTH!
long-run this paradigm (sorry, but that is the correct word) has profound implications for the design of large-scale unattended systems such as Reliable Water's desalination plant (page 4) and for groupware. The fundamental prescription is that these systems should be collections of agents, not centrally controlled applications that directly manage local activity, although they may well use a traditional database underneath. In other words, authority is delegated to autonomous applications (or agents) that negotiate or interact with each other. The central system can manage the data and define the rules, but it does not determine the activity of the individual agents (just like a proper laissez-faire government; page 29). One benefit to developers is that the programs they write need deal with only a subset of the whole system -- although the effects, good or bad, propagate globally.

Yet another crackpot idea?

Like art, self-organizing behavior is hard to define precisely; it's easier to discuss its characteristics and specific implementations of it. In essence, it results from the ability of interacting parts to create orderly systems out of chaos, in a world where the theory of entropy says that things inevitably get more and more disorderly. (Self-organizing systems, of course, take energy out of the environment to do their self-organizing.) You seem to get more information and structure out than you put in; seemingly simple equations or objects can produce complex, unpredictable patterns. The best way to predict them is by actually running a simulation.

Simulations will come in handy in predicting and determining how applications and modules will interact as we build both new and old modules into ever-larger interconnected (and thereby interacting) systems. Just as markets of multitudes of competing independent agents allocate resources (more or less) efficiently, so can self-organizing computer systems probably run a complex environment of computer systems more effectively than central control could. But that hardly means that any relatively stable state is optimal -- far from it. The tweaking and external controls that help to nudge a system to optimality are the basic problem here -- and a perennial concern of economists and policymakers as well as scientists. See our discussion of markets (page 17 to 19) for more on this topic. Nor do all systems of agents self-organize: Some stay inert, and others disintegrate into chaos.

The principles in action

Self-organization occurs with all kinds of objects: cells in a body, bees in a hive, people in a society, cars in traffic. The objects needn't be alike: It also applies to differentiated cells in a body, competition among species, relations between buyers and sellers or host and parasites. There are other familiar strains here: Object-oriented programming. The free market system (a special love since our return from the Soviet Union). Agents and demons. Evolution and ecology. Yield management (see Release 1.0, 88-2) -- which is airlines' attempt to participate in a market of independent customers with different strategies, mostly levels of price- and time-sensitivity, instead of treating them all as average. On the following pages we explore some more examples.

EXEMPLARY -- PAGE 4
Why it matters

An early sign of formal interest in the computer implications of self-organizing systems was the conference on emergent computation sponsored last month by the Department of Energy at Los Alamos National Labs. "Emergent," a confusing word in this context, refers to the emergence of complex, unpredictable patterns from the iterative interaction of seemingly simple rules -- that is, self-organization.

"Emergent computation" can be used to simulate natural systems, or it can constitute a working computer system itself. An exciting aspect of emergent computation is that it's cross-disciplinary, with contributions from hackers, physicists and neurosurgeons, to say nothing of demographers, statisticians and pure mathematicians. Traditionally the province of physicists and biologists, the concepts and biological metaphors of self-organizing systems offer useful models for the design and implementation of products and services to exploit the growing market for multi-user, distributed systems.

Because self-organization generates complexity and is a property of systems, not of their components, examples of self-organization don't scale down well. The best examples are things such as bee hives, ecosystems and the interacting cells that constitute living animals -- or smoothly running traffic systems such as wide-area telephone networks, the rural highway system or most Ethernet networks and other distributed systems.

Examples covered here include Borland's VROOMM (page 8), which lets interacting code modules monitor their own priority for memory management, and Corollary's 386/smp (page 9), which has a self-management approach for multiple 386es. The product that is likely to make many of these concepts clear to a mass market is SimCity (page 10), a city simulation for under $50 on the Mac that illustrates the complex interactions and feedback loops of factors such as population, business, housing, crime, taxes and pollution. It also illustrates that systems can "work" (self-organize and act coherently) without working well.

The present is an enormously fertile time for these concepts, because we are moving into a world of interconnected systems that need them, and we have the computing power (Connection Machines and such) to simulate them first. The basic question -- with many answers for many situations -- is: What simple local rules and behaviors generate the desired behavior in the larger, self-organizing system they constitute? While the best of these systems can handle change and error and accomplish complex tasks more effectively than a system that's programmed by hand, their parts still need to be programmed by hand. There's no magic!

This discussion is not so much a prescription for what you should do as for what you should think about -- a description of what is happening regardless in a world of interconnected, interacting systems. It explores the issues governing the behavior of large-scale systems -- systems where the cost of keeping track of all the components overwhelms the benefits -- and where interaction occurs on a local basis.
Of course, developers aren't all going to go off and grow software. Nor are we going to use these ideas in rebuilding existing accounting applications or databases. (Large doesn't necessarily mean complex.) But they might be used to allocate the computing resources to process these applications.

In terms of applications, computer-based self-organizing systems control only their own "environment" -- the computers they reside on and representations of the devices they control or monitor. So the obvious tasks to apply them to are network/resource management, as well as design (page 19), simulation, modeling and process control. For example.....

RELIABLE WATER COMPANY

What is a self-organizing system? The first large-scale, commercial self-organizing system we know of is the water-desalination plant recently unveiled by Reliable Water Company, RW-2. Reliable Water is a five-year-old concern co-founded by Ed Fredkin, former head of the MIT Computer Science Lab and a scientist noted for offbeat theories that confound traditional scientists by making sense.

RW-2 works without human intervention (except to turn it on). It is run on two 8-megabyte Mac IIs by REX, an integrated software system designed with the G2 process-control expert system development tool from Gensym, also co-founded by Fredkin. It operates with what Fredkin calls "machine instinct," and what we would call a bunch of autonomous cooperating agents.

REX itself would be far too complex to program as a traditional expert system, with rules tied to specific pieces of equipment, let alone in the traditional procedural way. Instead, most of the specific knowledge in RW-2 sits in a knowledge base of passive object classes such as valves, pumps, filter membranes and the water itself. (RW-2 knows enough about valves to calibrate a new one when it is added to the system and has already done so, says Fredkin.) These objects are hooked up to sensors monitoring the real thing, and supply their data to the software system. The system's layout of pipes was entered as a diagram that the tool could interpret.

In addition, there are about 100 generic rules, or active agents (a number that will continue to grow as Reliable Water's engineers keep tinkering), which manipulate the data objects and actually run the plant and keep its operations within tolerance. The agents have individual goals that combine to make the plant deliver clean water while meeting a range of constraints: safety, efficiency, maintenance of the plant, etc. They have rules for working with each other and for monitoring and controlling pressure, temperature, electrical currents, salinity and other parameters of the equipment and materials in the plant. Some care about current operating concerns; others consider the long-term impact of the short-term measures taken to keep the plant operating despite small-scale equipment failures or malfunctions. They also generate management reports and instructions for human maintenance engineers, and test the system after repairs. When readings disagree, the agents can compare notes.

While G2 (written in LISP) doesn't use the word agent, it has something called "focus," or collections of rules evoked when an unusual event happens, such as a power failure or leak. Agents performing other tasks continue routine work while the focus agents decide what to do -- stop the
plant and determine the cause of failure, fix the suspected item, test it, and so forth. Call them cooperating expert systems. There is no single point of failure in the software; each agent should be covered by several others if its task is not done properly. Says Fredkin: "It encompasses everything. An animal in the jungle doesn't go out on a paved path; it's got to cope with whatever it finds." This early commercial system, mind you, is of a kind where the worst failure would be the release of salty water into a water supply; it's not about to blow up, deliver a first-strike attack or harm humans in any serious, irrevocable way.

Nor was the system built overnight; in fact, it took five years to design, including extensive work on the reverse-osmosis process by which it desalinates brine, and programming. To get many of the agents' rules and procedures, says Fredkin, "We had to live through the accidents." One clever trick: Put a salt sensor in the floor drain; any salt loose in the system is sure to show up there pretty quickly. Moreover, RW-2 has far more machine sensors than an equivalent human-run plant would need.

The ultimate promise of self-organizing systems is not to make complex tasks simple, but rather to automate non-routine, non-simple behavior. They do not replace (or assist) a single expert, but rather model the behavior of a large group of mostly cooperating experts. By eliminating human factors such as sleepiness, illness, competition (except regulated cost-minimizing negotiation for resources), etc., self-organizing systems should operate more effectively and reliably than equivalent human systems.

Much of the work done by Reliable Water and Gensym at a pilot plant in the British Virgin Islands and a model in Massachusetts will be transferable to other kinds of systems, let alone clonable for an arbitrary number of desalination plants. (The company plans to field four more in the Cayman Islands and the Caribbean by December, at $350,000 to more than $1 million apiece.) The logic in REX that knows about valves and pressures should be easily transferable; so should many of its higher-level instincts/agents for keeping equipment in good repair, avoiding damage to humans, etc. Each of these must obviously be customized, but much can be done locally. (A manager can manage anything, right? Only the workers need to know much about the particular business they're carrying out.)

thINKer revisited

We wrote about a similar system of cooperating expert agents, without the conceptual context, in April 1986. That system, to manage page layout and other production processes for newspapers, was never commercially fielded because its vendor, Composition Systems Inc., was acquired and the project shut down. However, its principals, Mike Stock and Marvin Berlin, started a new company, Artificial Intelligence Technologies, which sells expert-system development tools akin to Gensym's. AIT is currently bidding on a couple of cooperating-expert systems, including a factory-automation system for a pharmaceutical company and a production system for a large publisher. "Our biggest problem in the old days was connectivity and database," says Berlin. AIT's Mercury system, however, has solved some of those problems with a tight SQL interface that lets it use distributed databases to manage information access across environments, whereas many expert system tools keep track of their own data -- with difficulty.
NEW WINE IN OLD BOTTLES: FAMILIAR TERMS WITH NEW MEANINGS

(We'd hate to defend these in a court of law or worse, an academic symposium, but they should be useful in understanding the discussion that follows.)

adaptation -- change, generally in response to circumstances, by any means. It's hard to program adaptation because the variety of circumstances is so great. (Also, if it's programmed, is it adaptation, or simply response?)

agent -- an autonomous application acting on behalf of another person or thing, with delegated authority. The agent's goals are those of the entity that created it. You could say that an agent is an active object with a mission, but agents are abstractions that can be implemented in any way, whereas an object has a fairly formal definition.

application -- a generic, vague term for almost any kind of code that executes a defined set of instructions. Or just useful software.

class -- the generic form of objects (see across). Each object instance is a member of a class from which it derives its behavior (and perhaps some class data). Each object instance has its individual identity and data, which may come from a file or a database, or may be derived at runtime.

competition -- when two or more agents with separate, perhaps conflicting goals attempt access to the same resources.

cooperation -- when two or more agents with at least one goal in common work together to achieve it.

evolution -- adaptation by means of selection, a form of self-organization. The individual components may or may not change; the system changes because of the replacement of its parts according to some criteria that cause the system gradually to change character in some way that further its survival and growth.

expert system -- generally, a data-driven, rule-based application. It executes non-procedurally but it embodies explicit, unchanging rules. (Of course, the rule could be to change its own code in such and such a way in response to certain conditions, in which case you might have a self-organizing system on your hands.) Because they need to be flexible (if not adaptable) in response to a variety of conditions, agents frequently contain expert systems. How many rules does it take to make an expert system instead of a rule-based code module? When does a procedural application contain rules, and when does an expert system contain procedural modules? It's a question of which one calls the shots.

genetic algorithms -- algorithms for simulating evolution. They direct programs composed of strings of subroutines to reproduce "sexually," generally by splicing code elements together, so that individuals of succeeding generations include code modules from both lines of ancestors.

Release 1.0 27 June 1989
The genetic algorithms also manage a selection process, so that the programs evolve to meet criteria set by the genetic algorithms.

**goal** -- a specification for long-term success. Only people have goals in the sense of mission or purpose; agents have explicit rules and constraints and instructions. Self-organizing systems seem to have goals, but that's a philosophical discussion....

**markets** -- self-organizing systems where the main means of interaction is competition and trading. While trading generally encourages more efficiency than competition without trading, markets suffer from the "tragedy of the commons," or local suboptimal solutions where everyone would be better off if all agreed on some course of action, but where no one's individual interest is so served. Cooperation needed here!

**objects** -- The traditional way of explaining an object is to say it's encapsulated (protected) data that includes procedures that are the only way of manipulating the data. But to the outside object-oriented world it looks much more like a program; it has behavior (executing code) that may have some data (parameters) attached. An object has a *protocol*, which is the kinds of messages it knows how to respond to. The ideal agent has a broad range of protocols (or situations) it can handle, and the ideal self-organizing system can respond to anything. *Active objects* stand around in memory and interact with other objects; they change things (and may do damage). *Passive objects* tend to do data collection and filtering, but they don't alter the data or change the situations they encounter. In the Reliable Water system (page 4), for example, passive objects represent the physical system elements; rule-based active objects (agents) manipulate them. Even passive objects may send messages to other objects, but in general their behavior doesn't have side-effects. (Finally, there are what you might call "dead" objects, where marketeers use the word object to describe rich data types that their systems can handle.)

Some people use "instance" to describe a particular object, and use *class* to describe a generic object. This is clearer, but it smacks overly of jargon to some.

**self-organizing system** -- a system of independent, interacting parts that shows regular patterns of behavior and adaptation generated by the parts' interactions.

**time** -- an important aspect of all these systems. You can't adapt after you're dead. Time delays in dissemination of information and feedback have important effects on the ability of systems to self-organize effectively.

**transaction** -- an atomic instance of (steps in) an application, with a defined set of procedures or events to be accomplished and a means of maintaining integrity and closure (or rollback to the original state). A transaction transforms the domain it concerns from one valid state at the beginning to another valid state at the end -- hence the obsession with managing concurrency and conflict. Just as objects may contain other objects, so may parent transactions contain (or consist of) nested child transactions.
MEMORY MANAGEMENT: BORLAND'S VROOMM

Another example on a smaller, less visible but far more mass-market scale is VROOMM, Borland's new Virtual Runtime Object-Oriented Memory Manager. This system enables large applications such as Borland's new object-oriented versions of Reflex (shipping), Quattro and Paradox (forthcoming) to operate in 640K. Rather than extend memory (which requires users to purchase additional memory), VROOMM builds a self-organizing memory management system that uses available RAM better, discarding small code modules, or objects, on a dynamically determined basis. (The data encapsulated in these objects is the code itself; any data they manipulated has been stored on disk or with the application's data structure.) Each code object keeps track of its own persistence priority. The priority is based on factors such as when it was last used, whether it was used to reach the current state of the system, and its interdependencies with other objects. For example, windows and menus are generally interdependent and loaded together, because menus are a kind of window and require the window code for much of their behavior. As memory fills up, the system kernel queries the objects to determine their persistence ranking, and sends the lowest-ranking object into the cache. (When the cache fills, the lowest-ranking item is deleted.)

Most applications with memory management swap in or discard huge pre-determined overlays up to 64K in size, which take lots of time to reload and contain much more code than the particular functions the user needs at any one point. By contrast, VROOMM lets the system load code in tiny, targeted increments. Performance degrades gracefully; as there's less space in RAM for code, swapping gets more and more frequent, but still takes only a second or two each time. Meanwhile, the application data can grow to take up almost all the space not required by the kernel. Obviously, such a system can work only if your code is divided into handy little object modules -- 2 to 4K in this case, which required complete rewrites of Reflex and Quattro (Quattro was redone anyway to incorporate technology acquired with Surpass). Paradox was already quite object-oriented under the covers and took to the new scheme easily, says Paradox designer Richard Schwartz.

But it certainly wasn't easy, says Borland vp product management Rob Dickerson. Borland can be quite open about the general techniques used, he says, because it took a lot of testing and debugging to develop the precise algorithms that would work effectively and the development tools to use them in a general way in other Borland products and for resale with language products. Exactly how much weight should you give to an element used four cycles ago versus one used eight cycles ago? Which pieces of code should be loaded together? Defining the objects and the class collections (all the related objects typically used for a particular task) is a key part of the job, and one that's specific to each application. Borland originally considered having the system retain information from session to session (to create "personalized software"), Dickerson says, "but it turned out that the biggest win was interdependencies. If a certain user rarely uses graphics, then it never loads the graphics-oriented modules, and they never show up."

It's hard to quantify the benefits from VROOMM, notes Dickerson, precisely because it moves you from a no-choice situation where the software simply dumps and reloads code overlays to one where performance degrades gracefully in a granular trade-off as data takes over memory, and more and more small
chunks are removed in a sequence selected dynamically to minimize the number that need to be loaded back.

Borland intends to use VROOMM in OS/2 versions of its applications and offer its facilities in language toolkits as well. OS/2 doesn't give you more memory; it simply gives you access to more memory if you have (or buy) it. As people start running several applications simultaneously in OS/2 with finite amounts of physical RAM, memory management will continue to be important. The possibility of optimizing and sharing objects across applications -- not a function of VROOMM but an extension of its approach that will have to be coordinated with OS/2 -- is exciting.

MULTIPROCESSING: COROLLARY'S 386/smp

One frequent use of self-organizing coordination schemes is in bus management, multi-processor scheduling and other distributed-systems tasks. OEM supplier Corollary is using such a scheme in its 386/smp, an add-in multi-processor system used by Zenith and other companies in their own high-end UNIX systems. To applications and programmers they look like fast 386 systems that run SCO Xenix, but inside, the 386/smp replaces SCO's UNIX kernel (while keeping all its utilities and drivers) with a new kernel that sets up a self-organizing scheduling system among up to ten 386 processors. "There is no way for a central controller to know what's going on fast enough" to allocate work efficiently, says Corollary president George White. Formerly with Computer Automation, Western Digital and Texas Instruments, White founded the company in 1985. His goal, he says, is to resist the temptation to put his computer into a box and to continue selling boards to other vendors. That way he can concentrate on performance rather than positioning, and use whatever schemes work underneath the hood without having to explain them to end-users, as follows:

To maximize coordination and minimize overhead and private use of code and data, the processors all share a global 64-megabyte memory on a high-speed bus (which doesn't work for geographically dispersed systems). Each processor looks to the system run list for the next available task when it finishes or suspends work. To minimize traffic across the bus, each processor has its own 64K cache for the data it's working on, and has a bias to tasks it's done before so it doesn't need to load new code or data. Then, in a feature called coherent caching, each processor board contains an agent to watch the bus and intercept any calls for the data in its cache, so that the call can be diverted to that cache rather than get the old version from the main memory. The 386/smp includes one processor attached to an AT bus to handle any AT peripherals the user may have. In addition, the multiple processors may have their own peripherals directly connected, and the system can be configured on installation so that any given processor favors tasks that use its particular peripheral.

This sounds rather complex -- but that's the whole point. By observing simple rules, the processor agents can carry out complex overall behavior that allocates the system resources efficiently.

Release 1.0 27 June 1989
EMERGENT TROJAN HORSE: SimCity

SimCity is fixing to be this year's hot game -- and perhaps the application that illustrates (rather than explains) the principles of self-organizing systems to the mass market. Positioned, and eminently enjoyable, as a game, it lets the user construct a city or play with any of seven model cities included in the package. The user has a budget and has to cope with a variety of familiar challenges, as well as optional monsters and unpredictable disasters. The user can build industrial areas, raise or lower taxes, hire police forces, replace roads with mass-transit systems -- and query the system at any time for a citizens' poll on his performance. But nothing he does is simple -- high taxes cause businesses and residents to move out; economic growth puts pressure on resources; police protection costs money. Quite apart from random events the developer threw in (including nuclear plant explosions although no nuclear power -- as opposed to weapons -- plant has ever exploded), the system is unpredictable because of all these dependencies -- a/k/a/ object interactions.

Experience seems to be that players initially enjoy zapping their cities -- and then quickly discover that it's more fun trying to keep them alive. "It's like looking at an ant farm and poking it with a stick," says developer Will Wright, 29, who spent three years developing the program to its current state and has studied countless books and papers on urban planning. "Pretty soon you decide that's no fun. There's no challenge. You want to build a city; that's a challenge." The basic message is that cities are complex, living things and there's no simple way to score performance. Wright asks, "Why do you want me to tell you what the score is? But maybe you could put in your goal when you start playing, and measure against that -- city size, or the poll of residents..."

In about 130K of C code, SimCity contains algorithms for 13 to 14 spatial arrays (maps) for data such as crime rates, land values, pollution and population, as well as 300 or 400 other interacting variables; call them objects. The system is especially complex because of this two-dimensional aspect: You don't just have a crime rate, but a map of where the criminals hang out and how they're affected by the presence of police, parks and other spatial elements. Many of these factors also have feedback loops: Crime drives down land values and attracts further crime -- unless you do something about it. Says Wright: "We encode the behavior between the elements and then we just let it run and the system's behavior emerges."

All things considered, 130K for the simulation is a small amount of code -- and initial complexity. It's the behavior that's complex. Imagine trying to code such a system procedurally or even with forward-chaining expert-system* techniques. It takes almost twice as much additional code -- and as much CPU time -- to manage the graphics and animation and other features.

Response to the game has generally been favorable even among professionals and academics. "Most of them want access to the algorithms so they can tweak it a little," says Wright. "It would be great to have it used by people who really know what they're doing. We're trying to Trojan-horse the education aspect, and it seems to be working."

Available at a store near you! On a Mac!! Under $50!!!
TRAFFIC CONTROL: TEXAS INSTRUMENTS' ComMUTE

One of the obvious uses of simulated self-organizing systems is in traffic control. The application is not using the systems to control the traffic, but using them to model and simulate the results of various local rules so as to design better traffic-flow systems. One such project, ComMUTE (for Computationally Metabolic Utilitarian Transportation Ecosystem), is underway at Texas Instruments' Dallas AI Lab, where two researchers are working to "get over the bottlenecks of central management, explicit control and top-down design with emergent computation," says researcher Marek Lugowski.

He and colleague Duke Buster are building simulated vehicles ("rho-dents," using the Greek letter for r) that crawl over grids to reach their destinations. Armed with about a page of LISP code (the environment takes a lot more), they start with two coordinates each and modify those coordinates to simulate travel. Their "goal" is to reach the destination coordinates, and they may not move to spaces occupied by other vehicles, which they can't see coming until they're in the next square in the grid. Over time, they learn to do this in a relatively efficient way. They work with some rules about avoiding occupied spaces and build an internal representation of the grid -- a map of their environment -- based on previous collisions. The information in the map decays, so that it can be updated with more recent experience and singular experiences wash out. In one simulation, with the destinations lined up along an axis of the grid, the vehicles learn to approach their destinations head-on in a way that minimizes collisions. Better yet, they readjust within 10 to 30 iterations after the destinations are shifted.

So far, ComMUTE is strictly experimental, but TI is funding it in the hopes that something commercial may come of it. (That's how Speak'n'Spell got started.) Lugowski, a four-year veteran of this area, has visions of airports without control towers, but TI will probably apply the approach first in such mundane (and less risky) uses as directing forklifts in a warehouse. Says Lugowski: "The problem with traditional AI as a solution is that it doesn't scale up. Things get too complicated and vulnerable to flaws." Simulations show that simple rules such as the vehicles' work independently of the total system's size. (The local ratio of rodents per grid area, however, has an impact on performance: You can't simply increase the number of rodents without giving them more room to run around in.) This makes intuitive sense too: Since all the vehicles know about and interact with is their local environment, the overall size of the system couldn't have any impact on their performance in avoiding collisions. But will they be as effective in getting to their destinations on long trips?

SELF-ORGANIZING OPERATING SYSTEM: KEY LOGIC

Last year we wrote at length about Key Logic's KeyKOS, an object-oriented operating system that provides extremely high security and efficient execution by packaging all its functions -- including emulation of traditional OSes such as UNIX and IBM's MVS -- as objects. After some financial and management commotion and a venture refinancing, the company has regained its bearings. Although still in start-up mode after four years, it is now calling on commercial customers jointly with IBM under a Complementary Marketing Agreement. It has also licensed the software to a large software vendor that plans to use it as the foundation of a communications program for resale. For further details, see Release 1.0, 88-5.

Release 1.0 27 June 1989
SOME PLATFORMS

For now, most vendors are simply working on creating the kinds of multi-system environments and tools that will let multiple user and agent applications interact, rather than building self-organizing systems per se. (Multi-tasking is de rigueur, so all these agents can operate concurrently.) The three major categories are databases of information about the components and state of the system (FileShare and Legato), cross-systems communications (Transarc and the Open Software Foundation's request for technology for a distributed computing environment) and programming tools that will help to build autonomous agents/transactions (Cooperative Solutions). Soft·Switch and Hewlett-Packard's NewWave environment will ultimately provide some of all three. NewWave's Object Management Facility, a platform that will span (at least) DOS, OS/2 and UNIX, has garnered some support from the Object Management Group, a growing would-be standards group led by Hewlett-Packard (a key OSF founder).

Overall, the platform/activity or central/local dichotomy sounds much like the division of labor between a database and the applications that surround it; the added idea is that the applications interact with each other. Distributed databases, because they have solved all these problems for themselves, give you the ability to build cooperative applications across platforms without worrying about these issues -- but only if you adhere to the database tool's restrictions. It's easy to push data around, harder to execute commands or send messages remotely.

WHO'S ON FIRST? SAROS FILESHARE AND LEGATO SYSTEMS

Saros and Legato Systems exemplify the differences between the OS/2 and UNIX mindsets. Although they have similar goals in mind -- building a system information substrate to manage system-resource objects and possibly application objects -- they have come up with two entirely different approaches. Initially, Saros will be selling a cross-network file manager primarily to network administrators for use by end-users in office-automation environments, whereas Legato will first sell a front-endless product to OEMs and programming teams who can appreciate the intricacies of its products. Both of them will eventually operate in the shadow of -- or support interfaces to -- X.500, the forthcoming (don't hold your breath) international standard for directory services, a sort of global database that will enable users and applications to keep track of system resources across wide-area networks.

Saros FileShare is one of the first OS/2 applications to show up -- and an SQL Server application to boot. But in fact it's systems software that uses the Sybase engine to accomplish the task of managing DOS and OS/2 files across a network. If you think of the world in the traditional form of layers you have to wonder how systems software can sit on top of all the other layers, but the modular approach puts such facilities side by side.

FileShare was originally designed to be a general object manager by a highly technical crew of people mostly from Tandem and Microsoft. But its founders discovered that their customer set was uneasy enough at the notion of adopting OS/2 servers, and didn't want to hear about objects and such. The customers had a real-world problem -- giving users access to files they should have access to, and keeping them away from files they shouldn't see.

Release 1.0 27 June 1989
FileShare allows users or applications to manage files by attributes -- author, user, date, etc. -- as well as by as many key words as they care to use (and search). Because it relies on a database that manages the files (as well as other objects, such as users, workgroups, etc.), user administrators can perform set operations -- delete all files last used before 14 July 1988 that do not include the key word "IRS." Or transfer all files used by the WonderWidget workgroup to the archives; the project has been canceled. Individual users can control their own private files, but the system manages group access to them in ways specified by the administrator.

FileShare thus lets almost any application function transparently as a multi-user application (aside from copyright issues) because FileShare takes over management of concurrent access to files and file-locking. In the short term, some vertical-market software vendors will be using it to manage files for their applications. As more and more application vendors support FileShare (which isn't a big technical challenge), getting to FileShare files from within an application will be almost as easy as it is now for DOS files, but FileShare does make the network visible in a way that network operating systems that let you use files off a server do not. The user has to find the file and store it locally before gaining access to it.

FileShare's file-management services are extensive, giving users ways to catalogue and selectively retrieve files, and allowing system administrators to control access, back-ups, version management, archiving and other tasks with virtually all the power of SQL Server. At the user's option, a single file can be copied by multiple users and manipulated locally (with notice to other users that this is happening). When any user returns the file, the new copy is kept as a version of the original, so you have, say, 3A, 3B, and 3C. At any point, those three can be logically merged in FileShare: the user has to do the work -- redlining and reconciling documents, say -- and then FileShare creates a single new Version 4 from the result.

On beyond files

Because it manages files, FileShare is the way dBASE used to be: An entire file is off-limits when any part of it is restricted; in other words, you can't lock records within a file, but just entire files. Of course, that's the province of a database manager, not a file manager anyway. Yet FileShare is really an object manager under its skin: With proper integration between an application and its underlying capabilities, you could just as easily manage objects for an application and provide smaller-grained control, including locking records if they are so defined. (It's more likely to be used for database views or applications, notes president Wayne Carpenter; leave the records to the databases.) Other possible tasks include managing system resources such as printers, mailboxes or application modules themselves anywhere there's an SQL Server to help keep track of things.

For such extensions, the product will need to add application logic to deal with the particular services and operating environments it's working in, but the fundamental idea of managing system resources will stay the same. Of course, it's a feature of OS/2 (and other advanced systems such as UNIX) that applications are composed of modules. Only a few of the modules need be task-specific; the rest are more general utilities that accomplish other parts of the job; and yet other modules are specific to the system platform. Typical of what we expect of many "OS/2" applications, such as Lotus Notes, FileShare is actually an OS/2 back-end linked to DOS front-ends.

Release 1.0

27 June 1989
Overall, we think FileShare will be successful if Saros can find a good way to sell and support it. It offers PC users (and their administrators) the same services taken for granted in other multi-user environments but not yet fully addressed by PC LAN and operating system vendors.

The UNIX approach: Legato

FileShare operates on OS/2 servers only. Although it can operate across multiple servers transparently (so that only the network administrator needs to know where things are physically, and then only at configuration time), it is limited for now to tightly linked OS/2 servers (despite its foundation on the widely ported SQL Server) and has code specific to OS/2 and DOS file structures and LAN security. Legato Systems, a 10-person VC-funded firm recently started by a group of former Sun employees addresses similar problems for the higher-end, more performance-sensitive UNIX world.

Their goal is similar to FileShare's—to provide an underlying facility that will enable users and applications to ignore the intricacies of network layout and accessibility. Legato, however, is initially going after programmers and developers rather than end-users, since that's where the action is. Its first product is Prestoserve, a server performance-enhancer that only a techie could love (or install).

Like FileShare, Legato's future products will use a database, but one modeled after the hierarchical system used in X.500. Legato is interested in being able to fit easily into the standards-sensitive UNIX world, while Saros is relying on Microsoft and Sybase for standards for now (although it uses mostly SQL Server's transaction and integrity services, and manages a lot of storage directly for efficiency). Long-run, Legato co-founders Rusty Sandberg and Bob Lyon expect they will be providing a kind of system resource manager that will work with rather than replace X.500. It will extend it by providing tools for system resources to register themselves so that applications and users can find the most efficient resources that meet specific criteria. In essence, it will provide a medium for publishing persistent information—such as the services available from a particular printer, or the applications available on a particular server—as well as a language for real-time inquiries, such as "Are you busy now?" and "Do you have corporate stationery in one of your bins?"

Although the underlying service is general, Legato intends initially to sell specific versions of it for functions such as printing, communications and "a decent back-up-and-restore service across heterogeneous CPUs—emphasis on restore rather than backup," says Lyon. "We want to turn everyone into an operator just like they're an operator of the telephone, by making it all easy and consistent." It's likely to be a lot easier to sell than object management.

Long-run, both these companies will probably be providing object management for self-organizing systems. In a world of many interacting objects/agents, it may be hard to find the right one to send a message to without the services of some such global directory. Legato's language will be one means of addressing objects that manage peripherals such as printers, fax machines and new devices of the future. (Of course, in the far distant future X.500 will probably advance to having an object-oriented database underneath.)
In the self-organizing object-oriented systems of the future, active agents will be performing real work (rather than just searching information services for interesting articles), and developers and users will want some way of defining and ensuring the integrity of the work they perform, i.e. a way to define transactions. Although the system as a whole may easily adjust to the failure of a transaction or two, the "owner" of the agent is unlikely to be as accommodating.

These agents can represent users trying to accomplish a high-level task, or manage system agents that manage resources, deliver mail or perform computations. (See glossary, page 6.) Just as an object is an encapsulated piece of data with some methods or behavior accessible externally, so is a transaction an encapsulation of a unit of work that transforms a domain from one valid state to another.

The transaction's behavior is visible externally, and it may send messages to other objects and agents anywhere on the systems to accomplish its work, but it maintains integrity of action over time just as an object maintains the integrity of its data. The transaction either commits or is rolled back. Thus, transactions are atomic activity, with logic to check and enforce integrity. No outside events are allowed to interfere with its behavior (just as objects respond only to messages they understand).

**Time-stretched transactions**

But the traditional notion of a transaction sees it as an instantaneous event: The state of the world outside is assumed to remain constant. In this sense, it's much like a batch job, notes SoftSwitch president Mike Zisman. (SoftSwitch sells multi-platform E-mail tools and systems.) In fact, transactions can go on for days (also an issue for object-oriented databases; Release 1.0, 88-12); examples include delivery of mail, editing changes to a document or design, multi-person approval cycles and other tasks that require input from several people (i.e. groupware). The trick will be to design transactions that can accomplish their work successfully and with integrity in the face of a changing outside world and possibly conflicting transactions. The simple approach would be to lock everything, but reality requires more granularity so that the world can keep moving while many transactions are going on concurrently.

E-mail is a combination of a front-end and a transaction-oriented back-end that manages the location and transfer of high-level information. This kind of back-end capability will be vital for agents communicating asynchronously. E-mail treats the transfer as an atomic transaction with facilities for error-recovery, integrity, reporting, etc. In the case of SoftSwitch and some others, it may also include data translation from one environment's format to another's. In the case of the Coordinator, it may include some content-based activity, monitoring the type of message and maintaining a record of transaction types and open items. (Who promised a report to Juan but hasn't delivered it yet? Whom did Alice delegate the customer's complaint to?) And there are nested transactions. Delegating a piece of work is a transaction, but the parent transaction-- completion of the work-- is still unfinished. Another E-mail-style transaction application is the grov-
ing area of EDI (electronic data interchange), with defined transactions for
purchase orders, invoices and the like.

In order to build agents, people will in fact be building transactions --
atomic applications that accomplish a specific task and manage its integ-

rity. One company addressing this issue is Cooperative Solutions, a start-
up whose principals left Tandem last summer and started the 11-person firm
this January after a long vacation. The company's aim is to enable users to
generate transactions for the new environment where "applications may be
smeared all over the network and have to adapt to resources -- database,
agents, mail -- moving around too," says co-founder Kim Worsencroft. "And
they still have to be atomic." Initially these will be mostly database-
based transactions, but Cooperative Solutions will attempt to be database-
and environment-independent -- unlike most current development tools.
"Object-oriented and transaction-oriented people come from different worlds,
but we're trying to join the two," adds co-founder Dennis McEvoy.

(The name of Tandem crops up quite frequently; it is one of the first com-
panies to have truly understood the issues of cooperative processing,
redundancy, transactions, etc. Indeed, one of the first independent-agent
ideas was designed by Tandem's Joel Bartlett into its Guardian OS in 1975 --
paranoid democracy. If a processor failed and another processor noticed,
the second processor would send a message to the administrator and blackball
the failed processor. However, it wouldn't tell the other processors; it
would let them notice for themselves -- which they would, eventually. The
reason? What if part of a failing processor's behavior was to broadcast a
message that another, healthy processor was failing? This scheme ensured
that processors couldn't falsely denounce each other to the system.)

OPEN WORLD: OSF AND TRANSARC

The Open Software Foundation is well aware that selecting UNIX and a user
interface isn't enough; the world needs a "Distributed Computing Environ-
ment" to standardize upon. In fact, you might well argue that the distrib-
uted computing platform is more important than the local operating system.
It's important only that local systems can communicate, rather than that
software from one system be able to execute on another one. (See Release
1.0, 88-11.) OSF's request for technology is seeking facilities for cross-
system communications, allowing applications and agents to interact freely,
including remote procedure call services, naming and authentication services
(part of X.500), presentation services (data translation between environ-
ments) and a distributed file system. These components will probably come
from several vendors and will be sewn together by OSF next year just like
the components of its Motif interface.

A possible contender for the OSF's blessing is a forthcoming product from
Transarc, a company recently funded by IBM among others and formed to
commercialize the technology underlying the Andrew File System long in use
at Carnegie-Mellon. The AFS implementation is also likely to be adopted by
NeXT and of course by IBM, especially if IBM puts its support behind the
NextStep interface/development environment. Transarc clearly has the tech-
nology for its first product, which will be a commercial version of AFS.
Future products will handle such tasks as implementing transactions over
networks, with special attention to security and nested transactions, based
on CMU's Camelot, for CArnegie-Mellon Low-Overhead Transaction system.

Release 1.0

27 June 1989
SOME EXPERIMENTS

Large-scale self-organizing software systems are still pretty rare. With the exception of Reliable Water Company's REX (page 4), the only ones we know of are research efforts. Two of the more prominent are the work of Danny Hillis at Thinking Machines, the company he founded, and Bernardo Huberman at Xerox PARC. Hillis is using the capabilities of the powerful Connection Machine he designed to model population dynamics and evolution and trends such as the spread of disease, while Huberman is more directly concerned with the use of self-organizing market-style systems to manage computer resources -- or what he terms "the ecology of computation" (the title of a book he recently edited on the topic).

What's the difference between a marketplace system and an evolutionary one? One is probably simply the time scale; the other is that in evolution the agents are replaced over time, while in the market it's the goods or services traded and used by the agents that go in and out of existence depending on demand for them (consider demand the criterion of survival in the market, but it's only the resources and market structure, not the agents, that evolve. (Of course, most systems are hybrids over the long run.)

In the Hillis model, adaptation of individual elements is the focus, whereas in the Huberman approach the issue is the organization of the system and the allocation of resources. Interactions between the elements in Hillis's model are simply competition for resources and high scores; in the Huberman model, agents negotiate each other in structured auctions for a variety of resources. (Note: these are not different approaches to the same problem, but different problems and different approaches that are both part of the broad field of self-organizing systems.)

BERNARDO AND THE MARKETS

Perhaps the most visible simulated market system is Spawn, the work of Huberman and his colleagues at PARC. Huberman is a former physicist who worked in chaos theory early, and has now moved on to self-organizing systems. Spawn is a computational "ecology," running mostly on networked Sun workstations, which has attracted considerable press attention of late. It enables PARC to make use of idle computer time through market allocation of processor time -- and of course to study the effects and effectiveness of a variety of algorithms for doing so.

Huberman takes a rewarding physicist's approach, focusing on experiments and measurements, and he's getting interesting results. The system has been up just a few months and produced the results described here, but it's not in general use: It's still easier to simulate than deal with the network's cryptic reporting interface and occasional failures. (Guess we need a self-organizing system to maintain as well as allocate resources.) The initial observations are hardly surprising: It takes only a few agents competing for resources to establish "rational" prices: That is, time on a machine is proportional to its performance, so that price-performance is relatively constant. Prices rise in response to shortages or high demand. And so forth. The efficiency of allocation is close to optimal.
But more interesting effects emerge when you start looking at issues like use of resources when there's a delay in feedback: What happens when (as is normally the case) you're dealing with delayed or imperfect information? If too many agents use a single resource, its performance goes down, but they don't find out until it's too late. (Sounds like Highway 101 to us!)

The situation here is that the decision is made on the price at one time, but the actual price paid is higher or lower depending on actual use of the resource. What happens when you introduce agents who make predictions? As it turns out, it's good to have a couple of agents making predictions, but if too many do so, chaos reigns again (cf. October 17, 1987, when people predicting a market collapse helped make it happen). A draft paper ("Collective behavior of anticipatory agents," by J. O. Kephart, T. Hogg and Huberman) and the chart below explain it succinctly:

"Initially, as analysts are introduced into a system of naive agents, the performance of the naive agents improves because the oscillation amplitude [deviation from the optimum] is reduced ("trickle-down effect"). The overall system performance improves as well. However, when the percentage of analysts increases beyond a certain point, the behavior becomes pathological, exhibiting complicated periodic or chaotic oscillations. The analysts start to perform even worse than the naive agents, and the overall system performance plummets. The cyclical analysts outsmart themselves, because their presence in the system induces a qualitative change in the dynamics which they do not take into account themselves [in this particular model]. This suggests that game theorists might be more successful..."

Initial work shows that adaptive game theorists do do better, but that's only because this market is relatively easy to predict....

Issues are availability of information, cross-impacts of behavior, the inability even of adaptive game theorists to predict prices correctly in any reasonably complicated model, and the importance of diverse strategies. For example, if everyone avoids the Bay Bridge because the radio says there's a traffic jam, it might be in a single person's interest to take the Bay Bridge. But if everyone figures that out, then what? If everyone wears PC...
Forum golf shirts, they lose their cachet. If Juan follows Alice's lead and becomes a manager, who's left to do the work? Some resources become more valuable if everyone uses them, like a popular movie (as long as you can get in) which is easier to talk about if others have seen it, while others become less valuable: "It's so crowded no one goes there anymore."

But most troubling is the so-called "tragedy of the commons," the fundamental problem we attempt to solve with government actions. This is the issue of shared resources: Everyone is better off if everyone pays, but each individual agent is better off if he doesn't pay and mooches off the other. In other words, how do we get global optimization when it's in contradiction with local optimization. The same issues apply to long-term research in the face of stock-market pressures, competition among firms in the face of external threats from other countries, the confusing interplay between safety nets and incentives in our welfare system and most current puzzles in public policy. (To see some of them played out, try playing SimCity, page 10.)

There is much interesting discussion of market systems in the work of Mark Miller and Eric Drexler in Huberman's and Drexler's books cited on page 33. Miller is currently chief architect at Xanadu, now part of Autodesk. He is working there on building a back-end text database that will ultimately play host to a variety of front-end agents that will not just search it but also annotate it, link cross-references and publish editorial assessments of its contents. More on this next issue.

<table>
<thead>
<tr>
<th>SQL and prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>The contrasts between SQL, the language of databases, and prices, the language of markets, may be illuminating. SQL is Boolean and deterministic and discrete: Get me such and such records, that meet such and such precise criteria. Pricing, by contrast, regards things as fungible: I want so much of whatever meets, more or less, these general criteria. The particular identities don't matter, and you can trade $10 of this for $10 of that without theoretically changing my happiness (ex transaction costs). SQL is about identity; prices are about functionality. SQL is digital; prices are analogue. SQL regards the world as tables; prices regard it as a fluid, shifting mass where one thing can replace another.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GENETIC ALGORITHMS: PROGRAMMING BY EVOLUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolution is a means of adaptation: A population evolves as its members are replaced by new and different ones; the individual elements don't necessarily adapt. Evolution works by selection: Successful elements (and their traits and behaviors) survive and reproduce. (Success may be determined by ability to get food and sexual partners, or to earn a living, or to acquire a specified level of currency or resource access in a computational system.) Unsuccessful elements die out, taking their traits and behavior with them. Over the long run, the successful elements and traits predominate. There is a seeming difference between traditional evolutionary systems, where individual elements are created and die out according to their success in meeting goals, and systems where the individual elements persist, while their behaviors or problem-solving techniques die out or flourish. But in</td>
</tr>
</tbody>
</table>

Release 1.0 27 June 1989
fact, it's merely a matter of mechanisms and metaphors. In a market-based software system, certain algorithms and resources might get used frequently, and others might "die" of neglect.

Will management then pour resources into making the widely used algorithms even more efficient, or making popular application programs available to more users? At this point, evolution in the software industry is working mostly at the macro level in this way, determining the marketplace success of operating systems, platforms, hardware architecture and applications. (Criteria for success include things such as marketing and management stability as well as the technical capabilities of the products.)

Market systems are exotic and scary enough to the traditional object-oriented programmer, who has to relinquish control of the executing application. A related but further-out notion is the idea of genetic algorithms, which leaves even the design of the programs up to a self-organizing system.

Genetic algorithms are algorithms for building sequences of subroutines (analogous to genes) that reproduce and mate, resulting in ever more efficient solutions for a given task (or for changing tasks, for that matter). This results in evolution, where the solutions change by replacing themselves over time. The genetic algorithms are not the solution algorithms themselves, but the programs that manage the rules of reproduction, genetic recombination and selection. In other words, they're the program running the simulation.

Typically, the program treats the solution algorithms as strings of text (code subroutines) with designated breakpoints where splices can occur. "Children" are created by joining spliced halves from two parents. Selection is made by actually running the child and scoring its solution of the problem -- typically some kind of optimization problem. Most genetic algorithms produce succeeding generations of solution strings of the same general form and length, but relaxing those constraints with so-called "messy algorithms" tends to produce more rapid, effective evolution towards superior performance, says Dave Goldberg, an expert in the field at the University of Alabama. "We've taken our cue from nature, where evolution has carried us from a small number of genes per organism to many with good results."

Stuffed software

The implications for software development long-run are promising; GE Corporate Research is using genetic algorithms in jet turbine design and reported promising results at the recent genetic algorithm conference -- reportedly a 2 percent increase in efficiency. But genetic algorithms are unlikely to see much use outside the design lab for a while; few dp managers are going to trust algorithms of uncertain parentage. Emergent computation and genetic algorithms in particular merge the notions of design and runtime, since it's a permanently incremental approach where the system is always in flux (or alive). But there's no reason that you can't use emergent principles for a period called "design," and then remove the genetic capacity to create a runtime system (regardless of what the purists might think). If you do it right, you then end up with inanimate but still executable software.
EVOLUTION: DANNY DISCOVERS SEX AND PARASITES

Danny Hillis, founder of Thinking Machines, turns out to have built an ideal machine for running genetic algorithms. He has been using his own Connection Machine and his limited free time to build self-organizing evolutionary simulations, modeling other people's ideas that ran too slowly elsewhere and also coming up with a few of his own. First, he has built a system that can evolve a sorting algorithm that comes within one step (60 vs. 61) of the best humans can do for sorting a list. (The algorithms consist of 64 sequenced instructions to compare and exchange a particular item in the list to be sorted. Each individual gets a random list to sort, and is scored on how well it has sorted the list by the end of the 64 steps.) That's pretty impressive. The whole thing took 100,000 generations, at about 10 generations per second for this model.

There are different ways to score to determine survival, and to generate successive generations, which both have a strong impact on the model. Scores on the problem represent the environment, in the sense that it's what the individuals are adapting to. The simplest method is to take an arbitrary percentage of the population with the highest scores. Another is to assign arbitrary survival probabilities to specific traits, which is how population biologists generally do it.

The problem with either of these scoring methods, of course, is that they're static. They end up producing a population specialized for a particular environment. But a changing environment is closer to reality, and adaptability, rather than specialization, is key to survival over time. (That's certainly the case with humans, which aren't particularly good at anything except intelligence, which equals learning, which equals adaptability.)

Populations may get stuck in a local adaptation (specialization) that's counterproductive in the long run. It turns out that parasites -- other objects that evolve more rapidly because their generations turn over faster -- can counteract the tendency to inertia. Any time a particular type gets too prevalent, it becomes an especially appealing host for parasites and is prevented from overtaking the population entirely, leaving enough diversity in the system for it to respond to new conditions. In other words, parasites keep populations from complacency, and may shake them out of suboptimal solutions to further improvements.

There's also the question of just how to elicit and spread a diversity of elements for the selection to work on. It turns out that exchange of genetic matter is generally far more important than mutation in keeping a population of complex things diverse. Mutations are less and less likely to be beneficial as an individual gets complex, whereas sex allows variations that work to be passed along and spread throughout a population. Sex is a means of what Hillis calls "idea-sharing." Sexual reproduction is simulated by recombining genes (represented as sections of the code strings) according to some specification to generate a new generation. (Although the pairing of any two individuals, or gene pools, is likely to result in a more "average" individual than the more extreme one, it also spreads its extreme characteristics more broadly than would simple cloning, or asexual reproduction.)

Many variations may sit around in the population "doing" nothing, but when conditions change somewhere there's likely to be a set of individuals better
suited to the new condition. As they mate with less favored individuals, they may help the population survive. Or if they stick to themselves, this small group may flourish and grow larger as the rest of the population flounders. Mating behavior -- random, pick your relatives or pick someone with similar string patterns are common strategies -- is another important characteristic of a model.

Gaps in the tape?*

But most interesting are Hillis's theoretical insights extending the work of others in the field, as described in his unpublished paper "Punctuated equilibria due to epistasis in simulated populations," which has traveled widely on the samizdat circuit. (A fine example of the cross-disciplinary nature of this field). In essence, it offers explanation for the strange gaps in fossil records, which indicate that evolution is not a slow, steady process, but rather a sequence of quick swings from one prevalent phenotype (trait or behavior) to another, or "punctuated equilibrium." There's only a short intermediate period as the majority of a population switches over.

The reason, as shown by Hillis's simulations, is that phenotypes aren't the result of a single gene, but of a number of genes acting in concert. Earlier simulations which looked at genes and phenotypes as equivalent (which used a single gene rather than several as a criterion for survival) missed this point. But simulations that use survival criteria based on the co-occurrence of several genes (e.g., subroutines that work well together) mimic the reality of punctuated equilibrium. Those genes may be lurking around individually in the population with little noticeable effect on survival rates. Chance combinations of the favorable genes (with high survival rates) slowly raise their frequency in the overall population until they reach a critical mass -- and suddenly blossom. "Any small increase in the frequency of one ... will greatly enhance the selective values of the others, and vice versa," writes Hillis (with a great deal of supporting mathematics).

Theatre of the emergent

As biology, Hillis's simulations lack a representation of spatial migration -- and no doubt other items too. But they can show some exciting patterns of reality without modeling all its aspects.

As theater, they vary in what they show on the screen. The computer can create populations of an arbitrary number of elements (call them objects), execute their behavior and then, based on the results, decide whether they survive or not. You can display them as locations on the screen (turning pixels on or off), or with colors so you can see characteristics as colors sweeping over the screen and through a population. The results are dramatic. As noted, if you have a static measure of success, you generally reach a stable state fairly quickly -- although not necessarily at the optimal level. This is an example of hill-climbing; once you've got to the top of the hill, you can't necessarily see another, higher hill elsewhere. It takes change and diversity -- the equivalent of turning the terrain into a moving ocean instead of static hills -- to make sure that the climbing continues. In real life, of course, circumstances are always changing.

*Release 1.0 27 June 1989
SOME PATTERNS

As you look at self-organizing systems, some interesting insights "emerge."

- Although systems self-organize by themselves (by definition), there's still lots of work left for the designers of their components; minute variations and nuances in the components can cause large, unpredictable variations in the resulting systems. See page 25.

- Scale and the definition of system boundaries are vital but complex questions. See page 26.

- As these systems get closer and closer to modeling and interacting with humans, they will raise ethical and sociological questions. The very world "agent" sets off complicated emotional responses. See page 27.

- Diversity is valuable (and irrepressible). See page 30.

- "Think globally; act locally!" translates to providing global data access, but letting applications make their own "decisions." The essence of self-organization is distributed control -- even with centralized information. (TI's ComMUTE, for one, uses only local information; see page 11.) The cost of maintaining central control rapidly overcomes its benefits as systems scale up, while market-style allocation schemes become more efficient (with exceptions). See pages 4 to 11.

- Self-organizing systems need reliable platforms for execution and communication. See pages 12 to 16.

- Careful attention must be paid to the quality and latency (timeliness) of information. See pages 17 to 19.

- Systems can evolve, but only if inefficient components are allowed to "die" (or terminate). See pages 17 to 22.

On the next few pages we try to describe self-organization and address some of these points not covered earlier.

As noted, self-organization (or emergent behavior) is a property of systems and results from the interaction of their individual elements. The best way to predict the pattern generated by their interaction is to simulate it. This "behavior" is not explicit (then it could be defined precisely), but a system's ability to maintain itself and to adapt to unforeseen events -- everything from painless assimilation of a new computer into a network to repudiation of spurious messages from a marauding hacker.

On beyond neural nets

Self-organization, however, is different from mere flexibility or adaptability; it's more akin to intelligence. SATs to the contrary, intelligence is not how much you know. It's how quickly (or whether) you can learn it. So in this sense emergent computing systems have the potential to exhibit far more intelligence than traditional expert systems, which merely execute rules rotely, or programmed recognition systems.

Release 1.0

27 June 1989
Neural nets, by contrast, are simple emergent systems which can learn to recognize patterns. Current computer based ones can't learn very much. A neural net is a relatively simple emergent system, because its nodes have very little visible behavior; they simply switch on or off. (Internally, the nodes adjust the weights of what may be extremely complex equations, and may produce extremely complex structures.) The emergent system that results is capable of recognizing faces or poor credit risks or (so some hope!) stock market patterns.

By contrast, a large object-oriented system consists of a lot of complex objects, with a variety of goals that may not bear any visible relation to the ostensible goal of the system they're part of. They pass complex messages back and forth, whereas neural nets have fixed connections and nodes that pass only binary signals or at best values to each other.

Will it take?

Of course, groups of objects don't always self-organize. There may not be sufficient interaction, in which case the system will remain relatively static; each node will do its thing without instigating the sort of chain reaction that constitutes a self-organizing system. At the other extreme, there may be chaos, in both the old and the new senses of the world. There can be so much irregularity that no useful patterns emerge, or they may be so extreme as to be useless for our purposes.

But in the happy medium the interactions result in a relatively stable system with periodic oscillations around steady states or evolution from state to state. Patterns emerge -- patterns that seem to hold far more information than the objects that created them did. How can millions of Chinese, for example, suddenly join to foment revolution -- while millions of Russians languish in sullen despair? This self-organized movement, unhappily,

---

1 Discussions of AI can get bogged down about what is "intelligent" and what isn't; it includes areas as disparate as explicit rule-based expert systems and neural-net-based pattern-recognition systems. But emergent computation concerns one specific feature of computational systems -- their ability to self-organize. All this raises the question of why life is necessarily carbon-based. What makes something alive? Is it the chemicals it's based on? Or how those chemicals are arranged? Or the fact that those chemicals arrange themselves? From a (relatively) short sequence of genes you can create a whole human being. By contrast, the things robots build are simpler than the robots themselves.

2 Yes, neural nets recognize patterns (see Release 1.0, 87-7), but they don't actually do anything; that's the province of expert systems and applications. Expert systems can be programmed to make decisions based on the patterns recognized by a neural net; applications do whatever the expert system decides. Of course, ideally all these components are technological approaches that are combined in a comprehensive, composite system that does everything, with the partitions invisible to the casual user. Objects are hybrids that could just as easily contain neural-net algorithms as the programs and data of "normal" objects.

Release 1.0 28 June 1989
turns out to be part of an even larger pattern of repression interspersed with easing. What’s the difference between smooth traffic flow and paralyzing gridlock? Between fast disk access and overloaded thrashing?

Content counts

Can emergent computation actually build systems more complex than what you put into them? That’s the hope.

What is the content of the agents that constitute them and generate such complex interactions? Theoretically, the behavior of the individual components should be easier to specify because they are (relatively) simple compared to the whole. If you are building a system modeled along human interactions -- for office automation, say, or the water plant described on page 4, you can start with the explicit interactions among people as your model. Just as good expert systems don’t mimic the brain but take a piece of its behavior (the use of explicit rules) and model that, and neural nets model excitations of brain cells, these systems simply take some specific rules of behavior and model those to produce systems capable of handling problems of overwhelming complexity.

But you still have to build the logic of the local diagnostic expert system or decision maker or data expert into your agent. And interactions among people are frequently not explicit, after all (part of the reason for some people’s discomfort with transaction-oriented groupware; Release 1.0, 88-6, and page 29). For example, what is the true meaning (to another computer agent, say) of the simple expression, "Let’s have lunch"? How do you model the interaction it represents? The hope is that by modeling only explicit behaviors, you may actually get systems that run more smoothly than human ones. But there will also be a lot missing.

The benefit of the model agents may be not just simplicity, but also the selection only of behaviors that are relevant and useful. You don’t want your agents to have emotions and moods, necessarily, but you want them to have socially useful goals and the behavior to achieve them. So you can model negotiating behavior, goals, etc., without having to model whole people. (But that means that your models, like those of "model" economies, may be of limited use in predicting actual behavior of a real system.)

Design for changing

How do you give a program/agent rules for changing itself? "Most correct and well-written programs will simply break if you make even a trivial change," says arch-programmer Bill Joy. That’s why you need a lot of them in a self-organizing system, so that they can cover for each other. And we’re not (except for genetic algorithms) talking about random changes: You need a "grammar" for making changes, to maximize the proportion of beneficial changes and minimize the harmful ones. (Most are probably immaterial.)

At the simplest level, you can have rules concerning data, which change the actions an agent takes but leave its overall behavior intact. At the highest level, a language like LISP, which treats the code itself as data, makes it easy to write self-changing programs. (Of course, these programs are themselves a set of interacting objects.) In the middle is a grey area, where event-triggered rules change the structure of the system in which the
agent operates, but the agent itself retains its original form. (In such a
system, individual agents may die out from lack of use or resources, but
they don't change. The result is evolution by replacement of components --
the traditional biological model.) In short, you can't define emergent be-
behavior by looking at the individual agents, but only at their interactions.

Levels and extents: Choose your -cosm

A basic issue in self-organizing systems is picking the level of abstrac-
tion: Are you looking at a whole system, or at components thereof? An ap-
plication consists of a set of functions, which consist of processes... and so forth. Should programs themselves self-organize? Or only systems?
Where should the active objects fit in, and where the data or passive ob-
jects? From a pragmatic point of view, if it's simpler to build something
than to grow it, then that's what you should do. The goal of using emergent
computation is to grow systems more complex than what you could build.

Aside from levels, how widely does it extend? Are people part of the sys-
tem, or are they external forces who simply program the agents and set them
in motion? At some point, the system currency allocated to certain of these
resources may indicate to management that it's time to buy a fourth printer,
or run a more cost-effective line between two workgroups. Is management
part of the emergent system? In this context, yes. Management's behavior
is affected by interaction with its components, and management's behavior in-
turn affects the very nature of the system.

The spirit is emergent, but the flesh needs a platform

As indicated just above, the extent of a self-organizing system depends on
your perspective. The system platform can be an interacting part of a self-
organizing system, or it can be the environment. Emergent software systems,
like any other software, are heir to traditional systems requirements. What
While emergent systems can be used to allocate system resources, they them-
selves operate on top of traditional operating systems and communication
facilities -- their environment.

The environment may comprise a database, networking software, etc. The
self-organizing systems, applications or system-resource managers, are the
flexible part that can usually override the imperfections and errors of the
platform. In a small, fixed system, errors are fatal, whereas in a larger,
adaptive one they can be overcome. The emergent (you might call it living)
system is error-tolerant; it overcomes small errors and repudiates big ones.

More and more of the typical computer system is becoming virtual hardware
-- that is, fixed or standard. The hardware used to be the only fixed part, a ni
and included the software, which was wired to perform particular tasks in
Then the software became "soft," and was able to give the same hardware a
variety of personalities. Eventually there was a separation between the
system software and the applications, and the system software became virtu-
ally "hard" once again. In some cases, it is now burned into ROM (as with
the Macintosh).

As indicated above, one could call these images of computer
functionality "virtual" ones, and the actual images as
imperfect.
platform. The problem with this is that they lose their capacity to evolve. They may respond, but they don't change themselves. (Without our necessarily being conscious of it, we are part of an evolutionary ecosystem where; these systems and software are constantly evolving; we call it the market.)

Distributed and central control. To add our credit card, we have to enter the number and press a button. Today, card data can even be embedded into our smart cards. Self-organizing systems are truly object-oriented, as opposed to systems called object-oriented because they contain objects: In the latter, a traditional procedural application calls objects as it needs them: A database application tool, for example, may use graphical objects to build a screen design, or a word-processing application may treat paragraphs and words as objects. But generally either the user or the application logic manipulates the objects; they don't manipulate each other in cascading series of interactions. Of course, there are degrees of this: Do the objects contain procedural code, or does the procedural flow occasionally set off object activities? But in principle the distinction is clear: Only a system of active objects with distributed control exhibits true emergent behavior. Take, for example, a system such as the MacroMind animation tools. Although they create objects with their own behavior, the system is itself fairly procedural: MacroMind's Director stage-manages the objects; they do not interact with each other, but rather each runs under the control of the central program. By contrast, with a simulation (as opposed to animation) program such as SimCity, the objects within the system interact with and affect each other. Difficult as it is for programmers to give up control as they design a system, it's much more difficult to accept that not only will the objects run under their own control, but that the system will actually redesign (not just reconfigure) itself.

Agents and authority. Reopening a vigorous discussion that took place at the November annual meeting of Computer Professionals for Social Responsibility, a panel at the recent conference on Expert Communication (sponsored by the Graphic Communications Association) addressed the nature and role of intelligent agents. The original discussion was kicked off by an airing of Apple's Knowledge Navigator video, well-known by now to all but computer-industry troglodytes. Generally, many people find this tape off-putting for reasons ranging from excessive marketing zeal to sexism to incredulity. Our reaction is mostly aversion to anthropomorphism: We like our computers to act like computers, and our friends to act like people. Computers' attempts to pass as people are either insulting to one's intelligence (if ineffective) or dangerously deceptive (because the illusion will inevitably break down).

What is an intelligent agent? We can't think of one other than a person such as George Smiley. On the other hand, a plain old computer agent can autonomously carry out reliably any number of explicit rules and instructions. Properly programmed, it can react in useful fashion to a large range of individually unpredictable situations, carry out the goals of its creator and adjust in specified ways to a variety of circumstances. Collections of such agents may even "learn." (Yes, a collection of agents thus might constitute something close to an "intelligent agent.")
What's the difference between an application and an agent? Probably the same as the difference between a person and an intelligent agent. In other words, an agent is an application with a mission -- although the software agent's mission is not its own.

In self-organizing systems, there are lots of agents that seem to be working on behalf of the system itself rather than for any user in particular, and there are lots of agents interacting, rather than a few agents interacting only with the system as environment.

Don't manipulate; delegate!

Any application can be set up as an agent, doing a user’s bidding. The deeper connotation of agent is an autonomous, discrete application that will go off and do something independently; it's the opposite of direct manipulation, where the user directs the application interactively. Instead, the user gives the application instructions -- whether with a language or through some kind of user-friendly fill-in form, prompted script or other tool -- and sends it off to work on its own. An agent can do its work once, periodically, or sit patiently in the background waiting for a triggering event. It can be as simple as an E-mail collector, or as complex as a junk-mail detector/filter or a sales manager's agent that gathers data from a variety of databases, generates a report and sends customized, personalized E-mail to the salespeople based on the contents of the report. (So what does the sales manager do? She can spend her time motivating the salespeople, calling on problem accounts and providing customer feedback to the product development group.)

A scenario

In the future, this agent might negotiate with other agents to plan meetings, set sales objectives, or even bargain. (The instructions could be: "Add 20 percent to whatever each salesperson proposes as a sales quota, and tell him or her that's next year's quota. Then take the average of that and her counteroffer as the final figure. If any salesperson refuses to confirm, set up a meeting (priority 3) and send me an exception message."

Obviously, once a salesperson's agent figured out the salesmanager's agent's rules, the sales manager would have to refine her agent's rules -- by starting with a floor offer of 110 percent of last year’s sales, for example. You can see how eventually the behavior of interacting agents -- especially when customers get into the act -- could become quite intricate, as envisioned by the team at PARC (page 17).

These agents have some conditional behavior, which could be expressed as if-then rules. Are they expert systems? As much as any other rule-oriented system. Because agents act for people, and are autonomous, they will generally include many data-driven actions rather than operate strictly procedurally. But that doesn't make them "expert" or "intelligent." It simply makes them capable of acting within a specified range of likely conditions. The last rule of any agent's instructions should run along the lines of "If you encounter unforeseen conditions, get back to me."
...and groupware

Groupware comes in two forms -- information-sharing systems, such as databases and bulletin boards, and more active, transaction-oriented systems such as the Coordinator and Coordination Technology's forthcoming product (see Release 1.0, 88-6). Many people are leery of the second form because they incorrectly assume it's authoritarian and rigid. In terms of this discussion, they think it's filled with government agents. But in fact groupware is about user agents interacting with each other. A user, of course, may be a representative of top management, but in general it's old-fashioned hierarchical systems that have government agents, in the form of hard-coded rules and procedures not subject to coordination and negotiation.

Systems and societies: Micro- and macrocosms

You can look at communities or even the entire population of the world as complex systems of individual people or communities. All these are relevant to some kinds of inquiry. We suspect that some of the world's trade problems are due to the propensity of some countries to act like single companies, and of some large companies to take on the trappings of government bureaucracies.

While we were in the Soviet Union, we saw a system which was programmed and controlled by the state -- and which didn't work. A large system lacks the flexibility to respond to local information. You can build some of that flexibility in, but the goal is to build in flexibility of a higher order -- a system that changes itself, not just its responses, in the face of new information.

By modeling these systems, we can hope to determine the optimal size of, say, business organizations: At what point does the overhead of internal coordination so overwhelm things that little work gets accomplished?
IN PRAISE OF DIVERSITY

Diversity is the equivalent of change across a system's parts at a single moment, instead of across time, the usual medium of change.

Thinking about self-organizing systems highlights the dangers of the lack of diversity represented by standards. Although you can build upon and extend standards, they enforce stasis and the end of progress in the specific areas they define. (Although you can build on standards, sometimes it's worth it to start over.) Aside from retarding improvement in a fixed environment, lack of diversity also leaves us vulnerable to external change -- everything from computer viruses to AIDS, global weather changes and the arrival of more efficient competitors. From a global perspective, diversity -- the fact that "we" included both small and large, both warm-blooded and cold-blooded, animals -- enabled us, the vertebrates, to survive in the face of whatever killed off the dinosaurs. Diversity leads to adaptability, because it offers a range of solutions.

In recent days, the Internet virus attacked (with exceptions) only machines running BSD 4.3 -- compelling testimony for why we don't want everyone using the same system. If you want to get worried, perhaps AIDS is the factor that will prove how similar all humans are. Mammals may survive, but this particular species may not. Similarly, the name UNIX may survive, but the particular variants espoused by OSF or UNIX International may not. (And it may keep the name but little else.) On the other hand, it's nice to have a couple to choose from, and OS/2 is spurring both of them along from outside.

Finally, change or no, you don't want each element of a system to have the same goals and behavior: For example, it's fine to have newsletter writers competing to be better, but if everyone were a newsletter writer there wouldn't be much for them to write about, no matter how finely competition honed their skills. Likewise, in a software system, you want specialized modules for specific tasks that can be called as needed. And you may want modules optimized for speed or for memory conservation, to be used as the availability of resources warrants.

Diversity and SAA

From this perspective, IBM may be moving in the wrong direction with SAA. Rather than let diversity flourish, it's attempting to obliterate the differences among its three broad product lines, with the result that none of them is optimized for any particular environment. And underlying it all, there's still the notion of the host controlling everything. IBM's notion of cooperative processing is a PC talking to a mainframe, rather than two consenting PCs.

The real issue isn't to be able to run all software elements everywhere, as SAA would have it, but to allow all software elements to talk to one another. That is, messages should travel freely, but they should be executed differently by different recipients. Objects with common protocols may have different internal implementations, with no one the wiser.

The correct balance (for now) is probably a variety of operating environments connected by a variety of communications protocols and even some facilities such as object protocols. The Open Software Foundation, for ex-
ample, is currently soliciting technology for a distributed computing environment (page 16). This would not legislate the structure of the individual elements, and in fact could be the most useful part of the OSF suite, long after the local stuff -- UNIX or whatever sits at an individual desktop -- is gone. While the communications stuff itself could also improve, in this context it's the environment rather than the self-organizing system itself.

Diversity in business

The same principles can be applied to companies themselves. Competition and diversity are good -- but at the right level: between companies not within them. This leads ineluctably to the conclusion that a company is frequently better off funding a start-up than taking on an innovative, counter-culture project itself. This philosophy lies behind Lotus's recent decision to "restructure" its relationship with Lotus Notes -- a wonderful product that Lotus finds itself incapable of selling or supporting properly. The issue isn't the quality of the product or the size of Lotus's salesforce, but simply the company's ability to twist its mindset to the service of a product so different from the one it was built upon. Lotus's heritage is numbers and individual productivity; Notes is about text and communication and collaboration.

Likewise, Cray Research's decision to split the company in two was a straightforward, clear-eyed recognition of the company's inability to support two conflicting efforts managerially and emotionally. Finances were beside the point.

This honest realism is refreshing in a world where everyone wonders how IBM can support both its proprietary SAA and the "open" UNIX standard (and within the UNIX camp, OSF's Motif as well as NextStep). Elsewhere, you could question the wisdom of Sun Microsystems' three product lines, and Hewlett-Packard's ability to absorb the Apollo product line and culture. Should Sun have bought TOPS? (On the other hand, Apple is doing fine with the Apple II and the Mac -- perhaps because they do not overlap at all anymore. But Apple was certainly smart to spin out Claris, for both external and internal reasons.)

In fact, stockholders of IBM would probably be richer right now if IBM had been split up by the Justice Department ten years ago. Meanwhile, former AT&T stockholders are doing well with their investments in the regional Bell operating companies carved out of the old AT&T, although the parent company is still fighting the aftereffects of its size and inflexibility.

Among smaller companies, the most common malady is lack of focus, not lack of funds. All too often, going for market share or exploiting opportunities means a diversity of products for a variety of customers -- a great idea in a mature market, but a dangerous dilution of effort where deep technology rather than breadth of product line is the key to success. Many start-ups die from internal competition for attention and resources long before they get big enough to attract competition from outside. (No examples because their names won't mean anything...)

By contrast, Cray Research's John Rollwagen and the new Cray Computer's Seymour Cray decided the original company's two parts would be better off if each could be single-minded in pursuit of one system architecture.
The country benefits too. Now it has two supercomputer companies instead of one, plus the promise of a start-up founded by yet another Cray alumnus, Steve Chen, and funded by IBM.

All this flies in the face of the current obsession with standards. In fact, the supercomputer race is far from over, and the last thing it needs is premature standards that would bring innovation and progress to a halt, or half-hearted competition between two parts of a strife-torn company. If the government doesn't know which effort to fund (or "protect"), so much the better. Let them fight it out in the marketplace. And let the government help them both by buying and using machines rather than with subsidies.

This applies, of course, to other markets besides supercomputers. The best thing the government can do anywhere is encourage businesses to compete. Although Cray welcomes help against Japanese trade barriers, it's heartwarming to see how eager it is to foster competition at home. (There's that -cosm issue again.)
RESOURCES & PHONE NUMBERS

Mike Stock, Artificial Intelligence Technologies, (914) 347-6860
Rob Dickerson, Borland International, (408) 439-1649
Gary Chapman, Computer Professionals for Social Responsibility, (415) 322-3778
Tolly Holt, Roger Moody, Coordination Technology, (203) 268-4045
George White, Corollary, (714) 250-4040
John Rollwagen, Cray Research, (612) 333-5889
Dennis McEvoy, Kim Worsencroft, Cooperative Solutions, (408) 377-0300
Eric Drexler, Foresight Institute, (415) 948-5830
Robert Moore, Lowell Hawkinson, Gensym, (617) 547-9606
Ann Hardy, Mary Wills, Key Logic, (408) 496-1090
Ruann Ernst, Hewlett-Packard/OMG, (408) 447-1341
Rusty Sandberg, Carol Realini, Bob Lyon, Legato Systems, (415) 329-7880
Doyne Farmer, Los Alamos National Lab, (505) 667-9186
Marc Canter, John Scull, MacroMind, (415) 442-0200
Will Wright, Jeff Braun, Maxis Software, (415) 848-0229 or 376-6434
Jonathan Gossels, Doug Hartman, Open Software Foundation, (617) 621-8700
Ed Fredkin, Reliable Water Company, (508) 670-2300
Wayne Carpenter, Mike Kennewick, Saros Corporation, (206) 646-1066
Mike Zisman, Soft•Switch, (215) 640-9600
Bill Joy, Sun Microsystems, (415) 960-1300
Danny Hillis, Thinking Machines, (617) 876-1111
Philip Lehman, Transarc Corporation, (412) 338-4400
Duke Buster, Marek Lugowski, Texas Instruments, (214) 343-4136
David Goldberg, University of Alabama, (205) 348-1618
Mark Miller, Xanadu Operating Company (Autodesk), (415) 856-4382
Bernardo Huberman, Xerox PARC, (415) 494-4147

For further reading:

"Grammatical Man," by Jeremy Campbell. This book is rather old, but it reaches back even further for a lucid exposition of (Claude Shannon's) information theory, and looks forward towards the generation of information by self-organizing systems.

"The Ecology of Computation," assembled and edited by Bernardo Huberman, North-Holland, 1989. (Thanks, Brian!) This is the emergent computation crowd's core text. Parts of it are more readable than others, but its parts self-organize into a useful survey of the field.

"The Blind Watchmaker," by Richard Dawkins (also author of "The Selfish Gene"), W.W. Norton, 1986. This is the basic (non-reference) text on the modern theory of evolution. Occasionally cute (or just English?), it's consistently thought-provoking.

"Engines of Creation: The coming era of nanotechnology," by K. Eric Drexler, Anchor Books, 1987. This was a speaker gift at the 1988 PC Forum. It has more ideas per page than almost anything else we've read; emergent computation is only part of it! Other topics include hypertext publishing and nanotechnology. Drexler is co-author of several of the papers in Huberman's book.

"Intelligence as emergent behavior," essay by Danny Hillis in the Winter 1988 edition of Daedalus on artificial intelligence. Also, several unpublished papers and talks.

Release 1.0 27 June 1989


Books we plan to read:


**COMING SOON**

- Xanadu.
- A composition engine.
- CompuServe, Prodigy, MCI Mail, usenet, Internet, et al. The only way we know to get around to getting online is to commit ourselves to writing about them...
- Network navigation.
- Patents and copyrights.
- Object-oriented database status report.
- And much more... (If you know of any good examples of the categories listed above, please let us know.)

Release 1.0 27 June 1989
**RELEASE 1.0 CALENDAR**

<table>
<thead>
<tr>
<th>Date</th>
<th>Event Description</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>July 17-21</strong></td>
<td>*CASE 89 - London. Sponsored by Index Technology and a host of academic groups, including London's Imperial College. Contact: Elliot Chikofsky, (617) 494-8200, x 1989.</td>
<td></td>
</tr>
<tr>
<td><strong>July 23-25</strong></td>
<td>*Sun Expo '89 - Santa Clara. Keynotes by Bill Joy and Scott McNealy. Sponsored by Sun Observer magazine for Sun users and resellers. If there is a hot UNIX box, this is it. Come see for yourself and meet the growing Sun community. Contact: Clayton Peters, (408) 296-7111 or (800) 828-EXPO.</td>
<td></td>
</tr>
<tr>
<td><strong>July 24-26</strong></td>
<td>Computer simulation conference - Austin, TX. Sponsored by the Society for Computer Simulation. Contact: Laurel Costello, (619) 277-3888.</td>
<td></td>
</tr>
<tr>
<td><strong>July 31-August 4</strong></td>
<td>*SIGGRAPH '89 - Boston. Sponsored by the Association For Computing Machinery. Keynote by Nicholas Negroponte. The annual festival for visual, graphical thinkers. Contact Cindy Stark, (312) 644-6610.</td>
<td></td>
</tr>
<tr>
<td><strong>August 1-3</strong></td>
<td>Comdex Asia/Pacific - Sydney, Australia. Sponsored by Interface Group. Contact: Cheryl Delgreco, (617) 449-6600.</td>
<td></td>
</tr>
<tr>
<td><strong>August 6-9</strong></td>
<td>DB2 Users Group annual meeting - Chicago. Sponsored by International DB2 Users Group. Contact: Samantha Sipowicz, (312) 644-6610.</td>
<td></td>
</tr>
<tr>
<td><strong>August 8-12</strong></td>
<td>*18th international conference on parallel processing - St. Charles (near Chicago), IL. Sponsored by Pennsylvania University.</td>
<td></td>
</tr>
</tbody>
</table>

*Release 1.0*  
27 June 1989
August 8

*Software development law '89 - Boston. Sponsored by law firm Elias and Goodman to precede MacWorld and targeted at Mac developers. Not just theory; this conference will have Apple reps talking concretely about such issues as use of the Mac name, HyperCard and HyperTalk, Apple software patents, etc. Speakers include Adam Osborne, Rachel Parker, Dan Bricklin and Dave Winer. Contact: Paul Goodman, (212) 421-6022 or Wes Thomas, (516) 266-1652.

August 9-12


August 9-11

*Conference on object-oriented dbms applications - Santa Clara, CA. Sponsored by Santa Clara University. Contact: Mohammed Ketabchi, (408) 554-2731 or mketabchi@scu.bitnet.

August 22-24


August 22-26

IJCAI-89 - Detroit. The international version of AAAI. Sponsored by the American Association for Artificial Intelligence. Contact: Claudia Mazzetti, (415) 328-3123.

August 23-25

TechDoc '89 - San Jose. "Publishing in the '90s...the art of publishing and the science of information management." Technical documentation and all its ramifications, including hypertext. Keynote: Jim Barksdale of Federal Express (a real treat!). Sponsored by Graphic Communications Association. Contact: Patti Hill, (703) 841-8160.

August 24-September 1

*Eleventh World Computer Congress - San Francisco. With a focus on tools and application software this year; in the U.S. for the first time in 24 years. Sponsored by 46 IFIP member societies. Call Nancy Dana, (303) 696-6100.

September 6-9

Breakaway '89 - Orlando. Sponsored by ABCD, the microcomputer industry association (mostly dealers). Contact: Deborah Keating, (312) 240-1818.

September 7-10

Comtec '89 - Singapore. Regional micro exhibition. Sponsored Microcomputer Trade Association of Singapore. Contact: Yong Mee Hiong, Singapore 2913238; fax 2965384.

September 10-13

September 11-14  NetWorld - Dallas. Managed by H.A. Bruno. Contact: Adam Torres at (201) 569-8542 or (800) 444-EXPO.

September 13-15  *Conference on Computer-Supported Cooperative Work - London (Gatwick). Inspired by the successful U.S. events, but likely to focus even more on social issues. Contact: Lorna Meek, 011 44 (753) 73232.

September 17-21  Managing the corporate personality for the nineties - Martha's Vineyard. Sponsored by Design Management Institute. With speakers from Xerox and Danish State Railways, among others. A broadening, useful conference. Call Nancy Barry, (617) 236-4165.

September 18-20  DataStorage - San Jose. Sponsored by DISK/TREND and Freeman Associates. Call Darlene Plamondon, (408) 554-6644.


September 19  Interface Design '89 - San Francisco. Sponsored by MacWeek, in hopes of consistency if not necessarily standards. Contact: Cindy Koral at (415) 243-3315.

September 20-23  Seybold computer publishing conference - San Francisco. Sponsored by Seybold Publications. The usual extravaganza, expanded from desktop publishing to include all electronic publishing. Contact: Kevin Howard, (213) 457-5850.


Release 1.0  27 June 1989
<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>October 1-4</td>
<td>*ADAPSO Management Conference - Orlando. Mingle with your peers (and Disneyworld's nearby just in case). Contact: Sheila Wakefield, (703) 522-5055.</td>
<td></td>
</tr>
<tr>
<td>October 1-4</td>
<td>*Alex. Brown Computer Services Seminar - Baltimore. The tenth annual... Contact: Rivka Hawk or Ellen Kempler, (301) 727-1700.</td>
<td></td>
</tr>
<tr>
<td>October 2-6</td>
<td>*OOPSIA - New Orleans. Sponsored by ACM/SIGPLAN. Come meet your fellow objects and share procedures. Send a message to Carole Mann, (407) 628-3602.</td>
<td></td>
</tr>
<tr>
<td>October 2-6</td>
<td>CD-ROM Expo - Washington, DC. Sponsored by IDG Conference Group. Contact: Mitch Hall, (617) 329-8090.</td>
<td></td>
</tr>
<tr>
<td>October 2-6</td>
<td>Interop 89 - San Jose. Interoperability made tangible, with tutorials, discussions, product demos and pitches. Sponsored by Advanced Computing Environments. Contact: Mark Belinsky, (415) 941-3399.</td>
<td></td>
</tr>
<tr>
<td>October 3-5</td>
<td>PC Expo - Chicago. Sponsored by PC Expo. Contact: Steven Faher, (800) 444-EXPO or (201) 569-8542.</td>
<td></td>
</tr>
<tr>
<td>October 16-19</td>
<td>EDUCOM - Ann Arbor, MI. Call Joan Davis, (609) 520-3340.</td>
<td></td>
</tr>
<tr>
<td>October 30-November 1</td>
<td>*Seventh annual Seybold Executive Forum - Boston. Sponsored by Patty Seybold's Office Computing Group. Contact: Deborah Hay, (617) 742-5200 or (800) 826-2424.</td>
<td></td>
</tr>
<tr>
<td>November 1-3</td>
<td>*UNIX expo - New York City. Keynote by noted UNIX fan Ken Olsen of DEC. Managed by National Expositions Co. Contact: Roger Halligan or Heidi Dethloff, (312) 332-4650 or (212) 391-9111.</td>
<td></td>
</tr>
<tr>
<td>November 5-10</td>
<td>*Hypertext '89/SIGDOC 89 - Pittsburgh, PA. Much larger, for better or worse, than the first, wonderful hypertext conference in the fall of '87. Hypertext covers the first three days; SIGDOC the last three. Sponsored by ACM. Contact: Elise Yoder at (412) 327-8181 for Hypertext '89; Mike Dolhi or Adam Young at Scribe Systems, (412) 281-5959 for SIGDOC 89. (How about a joint committee on standards for the use of apostrophes?)</td>
<td></td>
</tr>
</tbody>
</table>

Release 1.0 27 June 1989
November 13-15  
**UIST** - Williamsburg, VA. Symposium on user interface software and technology, sponsored by ACM SIGGRAPH and SIGCHI. Contact: John Sibert, (202) 994-4953.

November 13-17  
*Comdex* - Las Vegas. Also including MACdex. Contact: Jane Wemyss at (617) 449-6600 or (800) 325-3330.

December 4-6  
**First International conference on object-oriented and deductive databases** - Kyoto. Sponsored by IEEE, MCC, many others. Contact: Professor Kiyoshi Agusa, 011 81 75 256-1677 or Won Kim at MCC, (512) 338-3439.

1990

January 17-19  

January 22-25  

January 28-31  
**EDventure Holdings PC (Platforms for Computing) Forum** - Tucson, AZ. Sponsored by us! Note that it's earlier this year. Contact: Daphne Kis, (212) 758-3434.

February 20-22  
**Computer science conference** - Washington, DC. "Cooperation is the theme, among processing units, technologies, disciplines." Sponsored by ACM. Contact: Barbara Kyriakakis, (703) 323-2318.

February 22-25  

March 5-9  
**Seybold Seminars '90** - Boston. ...moves east. Call Kevin Howard, (213) 457-5850.

April 10-13  
**Macworld** - San Francisco. Later this year. Call Peggy Kilburn, (617) 326-9955.

Please let us know about any other events we should include.
-- Denise DuBois

*The asterisks indicate events we plan to attend. Lack of an asterisk is no indication of lack of merit.

Release 1.0  
27 June 1989
SUBSCRIPTION FORM

Please enter my subscription to Release 1.0 at the rate of $395 per year in the U.S. and Canada. Overseas subscriptions are $475, airmail postage included. Payment must be enclosed. Multiple-copy rates on request. Satisfaction guaranteed or your money back.

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>Company</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
</tr>
<tr>
<td>City</td>
<td>State</td>
</tr>
<tr>
<td>Telephone</td>
<td></td>
</tr>
<tr>
<td>How did you hear about Release 1.0?</td>
<td></td>
</tr>
</tbody>
</table>

Please fill in the information above and send with your check payable to: EDventure Holdings Inc.
375 Park Avenue, Suite 2503
New York, NY 10152

If you have any questions, please call us at (212) 758-3434.

Daphne Kis
Associate Publisher

Release 1.0 27 June 1989