The Ecology of Computation
B.A. Huberman (editor)
© Elsevier Science Publishers B.V. (North-Holland), 1988

Markets and Computation: Agoric Open Systems

Mark S. Miller
Xerox Palo Alto Research Center,
3333 Coyote Hill Road, Palo Alto, CA 94304

K. Eric Drexler
MIT Artificial Intelligence Laboratory,
545 Technology Square, Cambridge, MA 02139*

Like all systems involving goals, resources, and actions, computation can be viewed in economic terms. Computer science has moved from centralized toward increasingly decentralized models of control and action; use of market mechanisms would be a natural extension of this development. The ability of trade and price mechanisms to combine local decisions by diverse parties into globally effective behavior suggests their value for organizing computation in large systems.

This paper examines markets as a model for computation and proposes a framework—*agoric systems*—for applying the power of market mechanisms to the software domain. It then explores the consequences of this model at a variety of levels. Initial market strategies are outlined which, if used by objects locally, lead to distributed resource allocation algorithms that encourage adaptive modification based on local knowledge. If used as the basis for large, distributed systems, open to the human market, agoric systems can serve as a software publishing and distribution marketplace providing strong incentives for the development of reusable software components. It is argued that such a system should give rise to increasingly intelligent behavior as an emergent property of interactions among software entities and people.

* Visiting Scholar, Stanford University. Box 60775, Palo Alto, CA 94306

1. Introduction

A central problem of computer science is the integration of knowledge and coordination of action in complex systems. The same may be said of society. In society, however, this problem has been faced for millennia rather than decades, and diverse solutions have been tested for effectiveness through hundreds of generations of competition. Efforts to understand the resulting institutions and to describe their principles of operation have spawned the science of economics.

Contrary to common impressions (fostered by media coverage of politics and the stock market), most economic inquiry has little to do with guessing economic trends. Economics has many branches; the branch most relevant to this paper studies the consequences of pursuing goals within the constraints of limited knowledge and resources, and studies the institutions and patterns of behavior adapted to this pursuit. This branch of economics can without embarrassment be termed a science, since it meets the criteria for a scientific discipline [1,2].

At the broadest level of abstraction, the problems of social and computational coordination are fundamentally similar. Concrete parallels, however, are rough: memory space is a bit like land, or perhaps a raw material; processor time is somewhat like labor, or like fuel; software objects are like workers, or perhaps like managers or firms. In [I] we list a number of fundamental differences between computational and human markets. For example, within a computational system, activities need produce neither pollution nor other effects on non-consenting objects; the most typical product, information, does not form a depletable physical inventory; specialized labor forces (copies of specialized objects) can be expanded almost instantly and can be cut back without human anguish.

Despite these deep differences, we argue that the fundamental parallels between the problems of social and computational organization are strong enough to motivate the wholesale importation of economic models and metaphors into the computational domain, at least on a trial basis. These differences do, however, suggest that forms of organization that fail or are rejected in one domain may prove workable and desirable in the other. For example, the ability of computational systems to establish rules as genuine constraints where an analogous human legal system can only penalize violations makes possible patterns of organization that can only be approximated in society.

1.1. Why focus on markets?

For a variety of reasons, this work explores essentially pure markets as models of economic organization for computation, supported by a minimal "legal" framework of foundational constraints. A large body of economic theory and historical experience indicates that markets are, on the whole, remarkably effective in promoting efficient, cooperative interactions among entities with diverse knowledge, skills, and goals. Historically, those entities have been human beings, but economic principles extend to decision-making agents in general and hence to software objects as well. In [I], markets are considered as ecosystems and compared to others, such as biological ecosystems. This examination shows how the distinctive rules of markets (such as the suppression of force and protection of trademarks) foster the spread of cooperation (and encourage entities to compete to be effective cooperators).

This paper argues that market ecosystems are particularly appropriate as foundations for *open systems* [II], in which evolving software spread across a distributed computer system serves different owners pursuing different goals. When also open to human society, computational market ecosystems will enable diverse authors to create software entities and receive royalties for the services they provide, and enable diverse users to mold the system to their needs by exercising their market power as consumers. Computational markets can be made continuous with the market ecosystem of human society.

1.2. Sketch of a computational market

This line of investigation leads us to propose what may be called the *agoric* approach to software systems. *Agoric* (a·gó·ric) stems from *agora* (ág·o·ra), the Greek term for a meeting and market place. An agoric system is defined as a software system using market mechanisms, based on foundations that provide for the *encapsulation* and *communication* of *information, access,* and *resources* among *objects.* Each of these notions plays a role in supporting computational markets.

Here, the notion of "object" is independent of scale and language, and includes no notion of inheritance. An object might be small and written in an object-oriented language; it might equally well be a large, running process (such as an expert system or a database) coded internally in any manner whatsoever. Objects are assumed to communicate through message passing and to interact according to the rules of actor semantics [3,4], which can be enforced at either the language or operating system level. These rules formalize the notion of distinct, asynchronous, interacting entities, and hence are appropriate for describing participants in computational markets.

Encapsulation of information ensures that one object cannot directly read or tamper with the contents of another; communication enables objects to exchange information by mutual consent. The encapsulation and communication of access ensures that communication rights are similarly controlled and transferable only by mutual consent. These properties correspond to elements of traditional object-oriented programming practice; in large systems, they facilitate local reasoning about *competence* issues—about what computations the system can perform.

Extending encapsulation to include computational resources means holding each object accountable for the cost of its activity; providing for the communication of resources enables objects to buy and sell them. In large systems, these extensions facilitate local reasoning about *performance* issues—about the time and resources consumed in performing a given computation. Computational foundations suitable for markets thus offer advantages in the performance domain like those offered in the competence domain by object-oriented programming.

For concreteness, let us briefly consider one possible form of market-based system. In this system, machine resources—storage space, processor time, and so forth—have owners, and the owners charge other objects for use of these resources. Objects, in turn, pass these costs on to the objects they serve, or to an object representing the external user; they may add royalty charges, and thus earn a profit. The ultimate user thus pays for all the costs directly or indirectly incurred. If the ultimate user also owns the machine resources (and any objects charging royalties), then currency simply circulates inside the system, incurring computational overhead and (one hopes) providing information that helps coordinate computational activities.

2. Overview of later sections

Section 3: Computation and economic order. Basic characteristics of human markets illuminate the expected nature of computational markets. This section describes some of these characteristics and sketches some of the special issues raised in the context of computation.

Section 4: Foundations. The foundations needed for agoric open systems may be summarized as *support for the encapsulation and communication of information, access, and resources*. This section describes these foundations and their role in computational markets.

Section 5: Agents and strategies. The foundations of computational markets handle neither resource management (such as processor scheduling and garbage collection) nor market transactions. This section describes the idea of *business agents* and their use both in replacing centralized resource-allocation algorithms (discussed further by [III]) and in managing complex market behavior.

Section 6: Agoric systems in the large. Large, evolved agoric systems are expected to have valuable emergent properties. This section describes how they can provide a more productive software market in human society—opening major new business opportunities—and how they can further the goal of artificial intelligence.

Section 7: The absence of agoric systems. If market-based computation is a good idea, why has it not yet been developed? This section attempts to show why the current absence of agoric systems is consistent with their being a good idea.

Appendix I: Issues, levels, and scale. Agoric open systems will be large and complex, spanning many levels of scale and complexity. This section surveys how issues such as security, reasoning, and trust manifest themselves at different levels of agoric systems.

Appendix II: Comparison with other systems. Here are reviewed works ranging from those that draw analogies between human society and computational systems to those that explore adaptive computation from an economic point of view.

3. Computation and economic order

The basic features of computational markets are best understood by comparing them with human markets. Many important tradeoffs, such as those between market mechanisms and central planning, have already been examined in the context of human society.

3.1. Market organization

Consider the awesome dimensions of the American community. . .a labor force of 80,000,000. . .11,000,000 business units. . . .Who designed and who now directs this vast production-and-distribution machine? Surely, to solve the intricate problems of resource allocation in a vast economy, central guidance is required. . . .But American economic activity is not directed, planned, or controlled by any economic czar—governmental or private.

—A. A. Alchian and W. R. Allen, 1968 [5]

Two extreme forms of organization are the command economy and the market economy. The former attempts to make economic tradeoffs in a rational, centrally-formulated plan, and to implement that plan through detailed central direction of productive activity. The latter allows economic tradeoffs to be made by local decisionmakers, guided by price signals and constrained by general rules.

The command model has frequently been considered more "rational", since it involves the visible application of reason to the economic problem as a whole. Alternatives have frequently been considered irrational and an invitation to chaos. This viewpoint, however, smacks of the creationist fallacy—it assumes that a coherent result requires a guiding plan. In actuality, decentralized planning is potentially *more* rational, since it involves more minds taking into account more total information. Further, economic theory shows how coherent, efficient, global results routinely emerge from local market interactions. (The nature and function of prices and of market mechanisms are a notorious source of lay confusion—just as Aristotle threw rocks and yet misunderstood mechanics, so people trade and yet misunderstand markets. Alchian and Allen [5] give a good grounding in the basic concepts and results of economic analysis.)

Should one expect markets to be applicable to processor time, memory space, and computational services inside computers? Steel mills, farms, insurance companies, software firms—even vending machines—all provide their goods and services in a market context; a mechanism that spans so wide a range may well be stretched further.

As will be shown, however, a range of choices lies between pure central planning and the universal fine-grained application of market mechanisms. Computational markets, like human markets, will consist of islands of central direction in a sea of trade.

3.2. Encapsulation and property

The rationale of securing to each individual a known range within which he can decide on his actions is to enable him to make the fullest use of his knowledge. . . .The law tells him what facts he may count on and thereby extends the range within which he can predict the consequences of his actions.

—F. A. Hayek, 1960 [6]

...the law ought always to trust people with the care of their own interest, as in their local situations they must generally be able to judge better of it than the legislator can do.

—A. Smith, 1776 [7]

Computer science began, naturally enough, with central planning applied to small, manageable machines. The first programs on the first computers were like Robinson Crusoe on an empty island. They had few problems of coordination, and the complexity of their affairs could (at first) be managed by a single mind.

As the complexity of software grew, programs with multiple subroutines became the equivalent of autocratic households or bureaucracies with extensive division of labor. Increasingly, however, bugs would appear because the right hand would not know what the left hand had planned, and so would modify shared data in unexpected ways.

To combat this problem, modern object-oriented programming (to paraphrase) "secures to each object a known space within which it can decide on its actions, enabling the programmer to make the fullest use of his knowledge. Encapsulation tells him what facts he may count on and thereby extends the range within which he can predict the consequences of his actions". In short, motivated by the need for decentralized planning and division of labor, computer science has reinvented the notion of property rights.

Central direction of data representation and processing has been replaced by decentralized mechanisms, but central direction of resource allocation remains. Rather than "trusting objects with the care of their own interest, in their local situations", the systems programmer attempts to legislate a general solution. These general solutions, however, provide no way to make tradeoffs that take account of the particular merits of particular activities at particular times.

3.3. Tradeoffs through trade

...a capacity to find out particular circumstances...becomes effective only if possessors of this knowledge are informed by the market which kinds of things or services are wanted, and how urgently they are wanted.

—F. A. Hayek, 1978 [8]

...the spontaneous interaction of a number of people, each possessing only bits of knowledge, brings about a state of affairs in which prices correspond to costs, etc., and which could be brought about by deliberate direction only by somebody who possessed the combined knowledge of all those individuals....the empirical observation that prices do tend to correspond to costs was the beginning of our science.

—F. A. Hayek, 1937 [9]

Trusting objects with decisions regarding resource tradeoffs will make sense only if they are led toward decisions that serve the general interest—there is no moral argument for ensuring the freedom, dignity, and autonomy of simple programs. Properly-functioning price mechanisms can provide the needed incentives.

The cost of consuming a resource is an *opportunity cost*—the cost of giving up alternative uses. In a market full of entities attempting to produce products that will sell for more than the cost of the needed inputs, economic theory indicates that prices typically reflect these costs.

Consider a producer, such as an object that produces services. The price of an input shows how greatly it is wanted by the rest of the system; high input prices (costs) will discourage low-value uses. The price of an output likewise shows how greatly it is wanted by the rest of the system; high output prices will encourage production. To increase (rather than destroy) value *as 'judged' by the rest of the system as a whole,* a producer need only ensure that the price of its product exceeds the prices (costs) of the inputs consumed. This simple, local decision rule gains its power from the ability of market prices to summarize global information about relative values.

As Nobel Laureate F. A. Hayek observes, "...the whole reason for employing the price mechanism is to tell individuals that what they are doing, or can do, has for some reason for which they are not responsible become less or more demanded....The term 'incentives' is often used in this connection with somewhat misleading connotations, as if the main problem were to induce people to exert themselves sufficiently. However, the chief guidance which prices offer is not so much how to act, but *what to do."* [8] This observation clearly applies to the idea of providing incentives for software; the goal is not to make software sweat, but to guide it in making choices that serve the general interest.

These choices amount to tradeoffs. With finite processing and memory resources, taking one action always precludes taking some other action. With prices and trade, objects will have an incentive to relinquish resources when (and only when) doing so promises to increase their net revenue. By trading to increase their revenue, they will make tradeoffs that allocate resources to higher-value uses.

3.4. Spontaneous order

> Modern civilization has given man undreamt of powers largely because, without understanding it, he has developed methods of utilizing more knowledge and resources than any one mind is aware of.
>
> —F. A. Hayek, 1978 [10]

Will prices, trade, and decentralized tradeoffs be valuable in computation? This depends in part on whether central planning mechanisms will be able to cope with tomorrow's computer systems.

Systems are becoming available having performance tradeoffs that are nightmarishly complex compared to those of a von Neumann machine running a single program. The world is becoming populated with hypercubes, Connection Machines, shared-memory multi-processors, special-purpose systolic arrays, vectorizing super-computers, neural-net simulators, and millions of personal computers. More and more, these are being linked by local area networks, satellites, phones, packet radio, optical fiber, and people carrying floppy disks. Machines in the personal-computer price range will become powerful multi-processor systems with diverse hardware and software linked to a larger world of even greater diversity. Later, with the eventual development of molecular machines able to direct molecular assembly (the basis of *nanotechnology)* [11], we can anticipate the development of desktop machines with a computational power greater than that of a billion of today's mainframe computers [12,13].

One might try to assign machine resources to tasks through an operating system using fixed, general rules, but in large systems with heterogeneous hardware and software, this seems doomed to gross inefficiency. Knowledge of tradeoffs and priorities will be distributed among thousands of programmers, and this knowledge will best be embodied in their programs. Computers are becoming too complex for central planning, with its bottlenecks in computation and knowledge acquisition. It seems that we need to apply "methods of utilizing more knowledge and resources than any one mind is aware of." These methods can yield a productive spontaneous order through decentralized planning—through the application of local knowledge and local computational resources to local decisions, guided by non-local market prices. Instead of designing rules that embody fixed decisions, we need to design rules that enable flexible decisionmaking.

Markets are a form of "evolutionary ecosystem" [I], and such systems can be powerful generators of spontaneous order: consider the intricate, undesigned order of the rain forest or the computer industry. The use of market mechanisms can yield orderly systems beyond the ability of any individual to plan, implement, or understand. What is more, the shaping force of consumer choice can make computational market ecosystems serve human purposes, potentially better than anything programmers *could* plan or understand. This increase in our power to utilize knowledge and resources may prove essential, if we are to harness the power of large computational systems.

3.5. Command and price mechanisms

> An economist thinks of the economic system as being co-ordinated by the price mechanism....Within a firm, the description does not fit at all....It is clear that these are alternative methods of co-ordinating production....if production is regulated by price movements...why is there any organization?
>
> —R. H. Coase, 1937 [14]

Coase asks, "Why are there firms?". Firms are economic organizations that typically make little use of market mechanisms internally. If reliance on market forces always produced more efficient use of resources, one would expect that systems of individuals interacting as freelance traders would consistently out-compete firms, which therefore would not exist. In reality, however, firms are viable; analogous results seem likely in computational markets.

Market transactions typically incur higher overhead costs than do transactions inside firms [14,15]. These transaction costs (in both human and computational markets) are associated with advertising, negotiation, accounting, and problems of establishing adequate trust—typically, inside a firm, matching consumers with producers does not require advertising, instructions do not require negotiation, movement of goods does not require invoices and funds transfer, and coworkers share an interest in their joint success. Firms lower the high overhead cost of market transactions among numerous small entities by bundling them

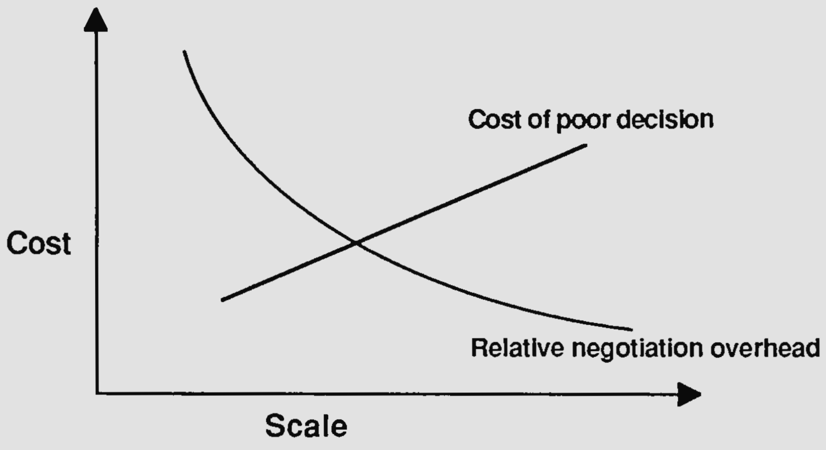


Figure 1: Scale and transaction costs. *As entities and transactions grow larger, the cost of making a poor decision grows, while the relative cost of market negotiation falls. If market decisions are better, but costlier, than central direction, they will be preferred by larger entities.*

together into fewer, larger entities. Not only does this save costs on what are now internal transactions, but by creating larger entities, it raises the size of typical transactions, making relatively fixed per-transaction overhead costs a proportionally smaller burden. (For small enough transactions, even the simplest accounting would be too expensive.)

Similar considerations hold among computational objects. For small enough objects and transactions, the cost of accounting and negotiations will overwhelm any advantages that may result from making flexible, price-sensitive tradeoffs. For large enough objects and transactions, however, these overhead costs will be small in percentage terms; the benefits of market mechanisms may then be worth the cost. At an intermediate scale, negotiation will be too expensive, but accounting will help guide planning. These scale effects will encourage the aggregation of small, simple objects into "firms" with low-overhead rules for division of income among their participants.

Size thresholds for accounting and negotiations will vary with situations and implementation techniques [16]. Market competition will tune these thresholds, providing incentives to choose the most efficient scale on which to apply central-planning methods to computation.

3.6. Can market objects be programmed?

The objects participating in computational markets must initially be much simpler than the human participants in human markets. Can they participate successfully? Human markets are based on intelligent systems, but this does not show the impossibility of basing markets on simple objects—it merely shows that the argument for agoric systems cannot rest on analogy alone. Explicit attention must be paid to the question of the minimal competence and complexity necessary for an object to participate in a market system. (These issues provide another motivation to form computational "firms" and to open computational markets to human participation.)

Experimental double-auction markets on a laboratory scale [17] give some indication of the requirements for market participation. Though involving human beings, some of these experiments have excluded most of the range of human abilities: they have excluded use of natural language (indeed, of any communications channel richer than simple bids and acceptances) and they have replaced goods with abstract tokens, excluding any cultural or historic information about their value. The participants in these markets have performed no sophisticated calculations and have lacked any knowledge of economic theory or of other players' preferences. Yet in this informationally-impoverished environment, these markets rapidly converge to the prices considered optimal by economic theory. Spencer Star [18] has successfully run double-auction markets among software entities employing simple decision procedures, and has achieved comparable efficiency.

Another reason for confidence in the applicability of market mechanisms to computation is the existence of primitive market mechanisms (outlined in this paper and presented in [III]) able to cope with such recognized software problems as garbage collection and processor scheduling. With evidence for the workability of market mechanisms both at this low level and at the sophisticated level of human society, there is reason to expect them to be workable at intermediate levels of sophistication as well.

3.7. Complexity and levels

Large computational ecosystems linked to the human market will have many parts, many aspects, many levels, and great complexity. Failure to recognize the differences among these levels will open many pitfalls. The field of biology suggests how to approach thinking about such systems.

Biological ecosystems obey physical law, but to understand them *as ecosystems* requires abstractions different from those used in physics. The existence of physics, chemistry, cell biology, physiology, and ecology as separate fields indicates that the concepts needed for understanding biological systems are naturally grouped according to the scale and complexity of phenomena to which they apply. Such a grouping may be called a *level*. Some issues arise repeatedly at different levels. For example, cells, organs, organisms, and hives all expend effort to maintain a boundary between their internal and external environments, and to bring only selected things across that boundary.

The concepts needed for understanding agoric open systems may likewise be grouped according to different levels, ranging from computational foundations through increasingly complex objects to market systems as a whole. As in biology, there are issues which appear in some form at all levels. Appendix I examines some of these issues, including security, compatibility, degrees of trust, reasoning, and coordination. In considering these issues in computational markets, it will be important to avoid misapplying concepts from one level to a very different level—that is, to avoid the equivalent of trying to analyze biological ecodynamics in terms of conservation of momentum.

The next three sections of this paper examine computational markets at successively higher levels, examining first foundations, then decision-making entities, and finally the emergent properties of large systems.

4. Foundations

Computation takes place in a context that determines what sorts of events can and cannot occur; this context can be viewed as the foundation of the rest of the system. Computational markets will require foundations that permit and forbid the right sorts of events. To simplify this discussion, the following explores foundations that provide for a basic uniformity in the nature of objects and their interactions despite differences in complexity and scale. In real systems, uniform foundations will ease the process of changing scale-dependent decisions and will make possible a unified set of conceptual and software tools spanning different scales.

It should be emphasized, however, that implementation of an agoric system will not demand adoption of a standard programming language. So long as certain constraints are met at the interfaces between objects coded by different parties, the language used inside an object can be freely chosen. The necessary constraints can be imposed by either the language or the operating system.

Computational foundations are frequently expressed in the form of programming language or operating system designs. Programming languages have evolved chiefly to provide abstractions for organizing computation on a small scale—the computation occurring inside a single machine and serving a single user. This has unfortunately led many programming language designers to make decisions (such as providing for global variables or access to arbitrary memory addresses) that make these languages unsuitable for organizing computation on a very large scale. The Actor languages, Argus, the concurrent logic programming languages (such as FCP), and the Mach operating system are examples of systems which have been designed to be extensible to large, open systems. These are covered in this book respectively in [IV], [V], [IV], and [VI]. All these projects have arrived at broadly similar models of computation from different directions, suggesting that their common elements will be of general value. This section briefly outlines some of the properties they share—properties which seem important for the implemention of computational markets.

4.1. Information and access

As indicated in Figure 2, the system capable of supporting open computation all share support for the encapsulation and communication of information and access. Communication of information is fundamental to computation that involves more than a single object. Encapsulation of information involves separating internal state and implementation from external behavior, preventing one object from examining or tampering with the contents of another.

In conventional practice, encapsulation of information increases modularity and conceptual clarity during design, a feature of considerable value. In agoric systems, though, secure encapsulation will be essential during operation. Without security against examination, theft of proprietary information would be rampant, and the rewards for the creation of valuable code

	Encapsulation of:			Communication of:		
	Information	Access	Resources	Information	Access	Resources
Dataflow, CSP, Occam	●	●		●		
Old Timesharing	●	●	●	●		
Actors, FCP, Argus, Mach	●	●		●	●	
Xanadu			●	●	●	●
FOCS	●	●	●	●	●	●
Agoric Systems	●	●	●	●	●	●

Figure 2: Comparison of foundations. *CSP is the "Communicating Sequential Processes" language of C.A.R. Hoare [19]. Occam is a related language for the Transputer [20]. FCP is Flat Concurrent Prolog, a concurrent logic programming language [IV], [21]. FOCS [22] is an operating system concept designed for resource ownership and service provision. Xanadu [23] is a hypertext publishing system described briefly in Appendix II, "Comparison with other work".*

and information would be reduced or destroyed. Without security against tampering, objects could not trust each other's future behavior, or even their own. Encapsulation provides a sphere within which an object may act with complete control and predictability.

Encapsulation and communication of access—capability security—ensures that the ability to communicate with an object can only be obtained in certain ways, such as through deliberate communication. With capability security, object A can get access to object B only by:

(1) being born with it, when object A's creator already has that access;
(2) receiving it in a message (from an object that already has that access); or
(3) being the creator of object B.

Capability security is a common foundation for protection in operating systems. It appears to be a flexible and general mechanism able to support a wide variety of policies for providing access protection. In an open system without capability security, every object would have to verify the nature and legitimacy of every message it received, imposing unacceptable overhead on small, simple objects. With capability security, simple objects can "assume" that their messages come from legitimate sources, because their creators can implement policies that limit access to trusted parties.

Together, the above properties yield security while preserving flexibility. Despite the Turing-equivalence of most programming languages, they can nevertheless differ formally and absolutely in their ability to provide for security [25]. How can this be, if one can write an interpreter for a secure language in an insecure one?

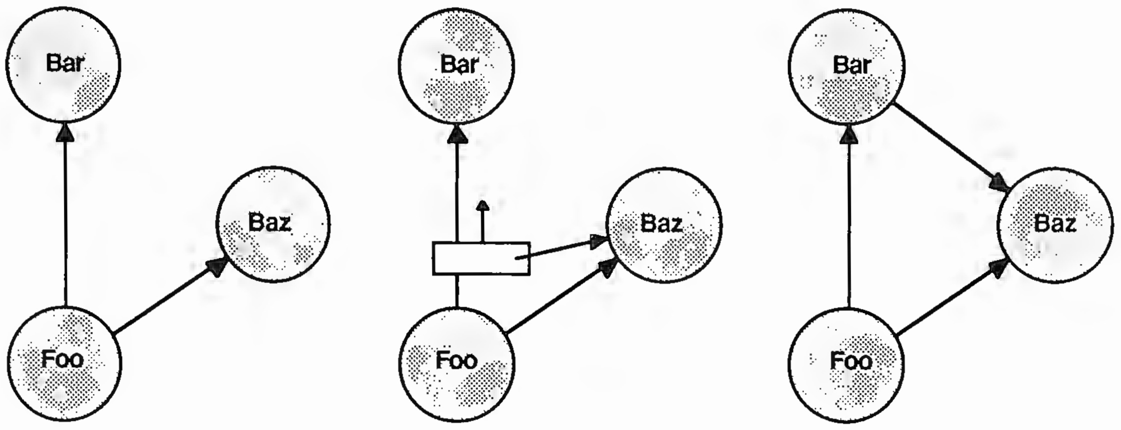


Figure 3: Communication of access. *Foo sends a message to Bar containing a copy of Foo's access to Baz. Upon receiving the message, Bar has access to Baz. (Adapted from [24].)*

Turing-equivalence describes the *abilities* of a system, but security rests on *inabilities*—on the inability to violate certain rules. Adding an interpreter on top of a system cannot subtract abilities from the system itself (even if the interpreted language consists of nothing but inabilities, as can easily be arranged). Thus, adding interpreters cannot establish the inabilities needed for security.

The question is not "what functions can be computed?", but "given that I am a computational object, what is my relationship to an already populated computational environment?". Let us call a set of computational objects coded in an insecure programming language "reference level objects", and those which exist on top of a reference-level interpreter "interpreted objects". If the interpreter implements a secure language, then the interpreted objects are protected from each other. Reference level objects, however, can simply ignore the interpreter and wreak havoc on the interpreted objects.

4.2. Ownership and trade

As software systems have evolved toward greater modularity, encapsulation of information and access have become more clean, uniform, and reliable. As has been discussed, encapsulation in software serves the same crucial function as property rights in human affairs: it establishes protected spheres in which entities can plan the use of their resources free of interference from unpredictable external influences. This enables entities to plan and act despite the limited, local nature of most knowledge; it thus permits more effective use of divided knowledge, aiding the division of labor. The value of protected spheres and local knowledge has thus far been the sole motivation for giving software modules "property rights" through encapsulation.

In economic systems, property rights also enable economic entities to accumulate and control the results of their efforts, providing the basis for an incentive system having the desirable evolutionary properties outlined in [I]. In agoric systems, encapsulation will begin to serve this function as well.

Agoric systems also require the encapsulation and communication of computational resources, such as a memory block or a processor time slice. This prevents the evolution of parasitic objects [I], confines the costs of inefficiency to inefficient objects and their customers, and (in suitable implementations) makes performance information available locally. Encapsulation and communication of resources correspond to ownership and voluntary transfer, the basis of trade.

A familiar systems programming construct which violates encapsulation of resources is the round-robin scheduler. In such a scheduler, the amount of processing power allocated to a process depends simply on the number of other processes. The processing power allocated to a given process will be reduced whenever some other process decides to spawn yet more processes. Under a round-robin scheduler, the processor is treated as a commons; given a diversity of interests, the usual tragedy is to be expected [26].

Artsy's paper on "The Design of Fully Open Computing Systems" (FOCS) [22] discusses an approach for an operating system design having the desirable properties specified above. Artsy's use of the term "fully open computing systems" corresponds to what would here be termed "extreme separation of mechanism and policy", where the mechanism is the support of protected transfer of ownership and the verification of ownership on access. All other resource allocation is then provided as user-level policy. Thus, schedulers and memory allocators are completely outside the secure operating system kernel and operate via an ownership-and-trade model. One can, for example, own and trade time-slices of a particular processor. Scheduling is performed at the user level by exchanging such commodities.

Starting from direct ownership of physical computational resources, more abstract models of ownership can be built. For example, a deadline scheduler can be viewed as follows: When a task is to be scheduled in a hard real-time application (*i.e.,* one that must meet real-time deadlines), it should be known beforehand how long it will take and by what time it must be done. When a process wishes to insure that it will be able to schedule a set of such tasks, it can purchase "abstract future time slices"—not specific time slices, but rights to a time slice of a certain duration within a certain period. Since this gives the seller of time slices greater flexibility with respect to other clients, such time slices should cost less than concrete ones. This is like a futures market, but with guaranteed availability—an honest seller of time slices will not obligate himself to sell time slices he may not be able to get. (See also [27].)

4.3. Resource ownership and performance modularity

The activity of a running program may be analyzed in terms of *competence* and *performance.* Competence refers to what a program can do given sufficient resources, but without explicit consideration of these resources. Competence includes issues of *safety*—what the program will not do, and *liveness*—whether the program will eventually do what it is supposed to, or will instead infinitely loop or deadlock. Performance refers to the resources the program will use, the efficiency with which it will use them, and the time it will take to produce results—precisely those issues ignored by competence. Both these issues have been the subject of formal analysis: the competence aspects of a programming language may be formal-

	Formal Analysis	Modularity
Competence, Safety and Liveness	Semantics, Correctness proofs	Object-oriented programming, Message passing
Performance, Efficiency	Complexity theory, Proofs of response time	Computational markets, Prices

Figure 4: Markets and performance modularity. *Issues of program competence and performance can be dealt with using the conceptual tools of formal analysis and modularity. Computational markets provide leverage for modularizing performance issues like that of object-oriented programming for competence issues.*

ized as a programming language semantics and used to analyze safety properties via proofs of partial correctness and liveness properties via proofs of termination. The performance aspects of a program may be formally analyzed via complexity theory and proofs of response time (for real-time programming).

Formalization alone, however, is insufficient for dealing with these issues in large programs—a complex non-modular program in a formalized language will often resist formal (or informal) validation of many important properties; modularity is needed to make analysis tractable. Modularization proceeds by separating interface from implementation, allowing concern with what a module *does* to be somewhat decoupled from concern with *how it does it*. Object-oriented programming and abstract data types aid modularization of competence issues, with message protocols serving as an abstract interface for competence effects [28]. Similarly, computational markets will aid modularization of performance issues, with prices serving as an abstract interface for resource costs.

4.4. Currency

For a broad market to emerge from these foundations, a system must provide for ownership and trade not only of basic computational resources, but also of virtual, user defined resources. Such resources can serve as tokens for establishing a system of currency. Public key communications systems [29] enable implemention of a secure banking system; within a mutually trusted hardware subsystem, capability-based security plus unforgeable unique identifiers are sufficient for establishing a public key system without resorting to encryption [25].

Accounting mechanisms have been used in software to some extent. Old time-sharing systems are one of the more familiar models—a fact which may raise grave concerns about the desirability of agoric systems. But using an agoric system would not mean a return to the bad old days of begging for a grant of hundreds of dollars of computer time and storage to edit a medium-sized document late at night, or to perform some now-inexpensive computation. The cost of computers has fallen. It will continue to fall, and personal computers will continue to spread. Aside from overhead (which can be made small), accounting for the costs of computation will not make computation more expensive. Making human beings pay for computer time is not the goal of computational markets.

In agoric systems, objects will charge each other and the machine will charge the objects. Given low enough communications costs and the right sorts of demand, a personal computer could earn money for its owner by serving others, instead of remaining idle. A machine's owner need not pay to use it, since the internal charges and revenues all balance. In a stand-alone computer, currency will simply circulate, incurring a computational overhead but providing internal accounting information which can guide internal decisions.

Inside one machine, one could have the foundations establish an official currency system. No secure way has yet been found to do so between mutually distrustful machines on a network without relying on mutually-trusted, third-party machines serving as banks. In accord with the goal of uniformity, such banks are here suggested as the general model for transfer of currency [25,30]. These banks can maintain accounts for two parties; when party A transfers money to party B, the bank can verify for B that the money has been transferred. (The cost of verification provides an incentive for A and B to establish enough trust to make frequent verification unnecessary.) In this model, it is unnecessary and perhaps impossible to establish any one currency as standard. There will instead be a variety of local currencies with exchange rates among them; it has been argued that this will result in greater monetary stability, and hence in a more efficient market, than one based on a single currency [31].

4.5. Open problems

At the foundational level, many open issues remain. Actors and FCP seem to be clean, simple open-systems programming languages, but they have no evident mechanism for dealing with machine failure. Argus is an open-systems language able to deal with this problem, but only by directly providing distributed abortable transactions as a basic mechanism. While such transactions provide much leverage, they are quite complex. A promising line of investigation is the design of a language having the simplicity of Actors or FCP, but which provides *mechanisms* for failure-handling that enable user-level *policy* to support Argus-style transactions. Even more satisfying than such a design would be a demonstration that Actors or FCP already have sufficient mechanism.

More central to agoric systems is adequate resource accounting. There is as yet no open-systems language which provides for ownership and trade of basic computational resources while preserving semantic uniformity and supporting the emergence of charging and prices. It seems this has been accomplished in the realm of operating systems design [22], but unfortunately in a way which is not yet amenable to distributed systems. It would be exciting to apply Artsy's work to open-systems oriented operating systems like Mach [IV].

5. Agents and strategies

When a problem needs to be solved frequently, but no single solution is right in all situations, it is desirable to permit the testing and use of many solutions. To allow this freedom, one seeks to separate mechanism and policy, designing foundations that support a wide range of policies via general-purpose mechanisms.

Foundational mechanisms for ownership and trade of information and computational resources allow the choice of policy to be delegated to individual objects; these objects may in turn delegate their choices to other objects, termed *agents*. Even policies for such fundamental processes as processor scheduling and memory allocation can be so delegated. The following argues that agents at a higher level can accomplish adaptive automatic data structure selection, guide sophisticated code transformation techniques, provide for competition among business agents, and maintain reputation information to guide competition.

5.1. Resource allocation and initial market strategies

Systems programming problems such as garbage collection and processor scheduling have traditionally been addressed in the computational foundations, casting the architect in the role of omniscient central planner. In this approach, the architect imposes a single, system-wide solution based on global aggregate statistics, precluding local choice. In the market approach, however, these problems can be recast in terms of local negotiation among objects. Solutions in this framework also provide objects with price information, allowing them to make profitable use of the resulting flexibility.

This enables programmers to provide objects with specialized resource allocation strategies, but it need not force programmers to attend to this. Objects can delegate these strategic issues to business agents, and a programming environment can provide default agents when the programmer does not specify otherwise.

The companion paper "Incentive Engineering for Computational Resource Management" [III] describes and analyzes *initial market strategies* which, if followed by a set of business agents, result in distributed algorithms for allocation of processor time, memory space, and communication channels. Initial market strategies (whether these or others) will play a key role in establishing agoric systems: from a traditional programming perspective, they will provide initial policies for garbage collection and processor scheduling; from a market perspective, they will help provide initial resource prices and thus an environment in which more sophisticated entities can begin to operate. Thus, they will help bridge the gap between current practice and computational markets. As markets evolve, the scaffolding provided by initial market strategies may be largely or entirely replaced by other structures.

The initial strategies for processor scheduling are based on an auction in which bids can be automatically escalated to ensure that they are eventually accepted. The initial strategies for memory allocation and garbage collection are based on rent payment, with objects paying retainer fees to objects they wish to retain in memory; objects that are unwanted and hence unable to pay rent are eventually evicted, deallocating their memory space. These approaches raise a variety of issues (including the threat of strategic instabilities stemming from public goods problems) that are addressed in our companion paper. Together, these strategies provide a proof of concept (or at least strong evidence of concept) for the notion of decentralized allocation of computational resources.

Initial market strategies will provide a programmable system that generates price information, enabling a wide range of choices to be made on economic grounds. For example, processor and memory prices can guide decisions regarding the use of memory to cache recomputable results. Given a rule for estimating the future rate of requests for a result, one should cache the result whenever the cost of storage for the result is less than the rate of requests times the cost of recomputing the result (neglecting a correction for the overhead of caching and caching-decisions). As demand for memory in a system rises, the memory price will rise, and these rules should free the less valuable parts of caches. If the processing price rises, caches should grow. Thus, prices favor tradeoffs through trade.

5.2. Business location decisions

Price information can guide a variety of other choices. Many of these resemble business location decisions.

Main "core" memory is a high-performance resource in short supply. Disk is a lower performance resource in more plentiful supply. In an agoric system, core memory will typically be a high-rent (business) district, while disk will typically be a low-rent (residential) district. Commuting from one to the other will take time and cost money. An object that stays in core will pay higher rent, but can provide faster service. To the degree that this is of value, the object can charge more; if increased income more than offsets the higher rent, the object will profit by staying in core. Treating choice of storage medium as a business location problem takes account of considerations—such as the relative value of prompt service—that traditional virtual memory algorithms do not express.

Small objects would have an incentive to "car-pool" in disk-page sized "vehicles". But given the issues described in Section 3.5, a typical object buying resources on the market may occupy many pages. Instead of deciding whether it should be completely in or out of core, such an object might decide how many of its pages should be part of an in-core working set, perhaps relying on a traditional algorithm [32] to dynamically select the in-core pages.

The variety of types of memory also suggests a need for more flexibility than the traditional two-level approach provides. The many kinds of memory differ in many ways: consider fast RAM cache, write-once optical disk, and tape archived in multiple vaults. Memory systems differ with respect to:

- Latency
- Storage cost
- Access cost
- Reliability
- Transfer rate
- Locality structure
- Predictability of access time
- Security

Tradeoffs will change as technology changes. To be portable and viable across these changes, programs must be able to adapt.

Much has been written about the need to migrate objects in a distributed system in order to improve locality of reference [30,33,34]. Again, this resembles the problem of choosing a business location. Machines linked by networks resemble cities linked by highways. Different locations have different levels of demand, different business costs, and different travel and communications costs. Various traditional approaches correspond to:

- staying put and using the phone (as in Mach [VI] and the V kernel [35]),
- commuting to wherever the momentary demand is (as in Apollo [36]),
- moving only when there are no local customers (as in the Bishop algorithm [37]),
- coordinating multiple offices (as in Grapevine [38] and in [39]),
- and moving where labor costs are lower (load balancing, as in [40]).

If limited to any one of these methods, human societies would suffer terrible inefficiencies. One expects the same for large, diverse software systems. If a system's mechanisms support a range of policies, different objects can select different approaches.

The notion of location in a space is still rare in object-oriented programming (for an exception see [41]). All memory in an ideal von Neumann computer is effectively equidistant, and many real computers approximate this ideal, but in a widely distributed system, differing distances are of real importance. When objects are given an account of the costs of communicating and commuting, they gain a useful notion of distance for making economic decisions.

5.3. Business agents

In a market containing sophisticated, potentially malicious objects, how can simple objects hope to negotiate, compete, and survive? One answer would be to shelter simple, mutually-trusting objects within large, sophisticated objects, building the latter out of the former. This model, however, would preclude turning loose small objects as service-providers on the open market. Other means are required for giving small objects the market sophistication they need.

Just as delegation of tasks to other objects can enable a small, simple object to offer sophisticated services, so delegation can enable it to engage in sophisticated market behavior. In this work's terminology, an object can delegate competence-domain actions to a *subcontractor;* this corresponds to the normal practice of hierarchical decomposition, which originated with the subroutine. An object can likewise delegate performance-domain actions to an *agent;* this seems likely to be a normal practice in agoric systems. Simple objects then can make their way in a complex world by being born with service relationships to sophisticated agents (which themselves can be composed of simple objects, born with...). Initially, human decisions will establish these relationships; later, specialized agent-providing agents can establish them as part of the process of creating new economic objects. The initial market strategies mentioned in Section 5.1 could be provided by simple agents.

One might object that a simple object and its collection of agents together constitute a complex object. But these objects, though complex in the performance domain, can remain extremely simple in the competence domain. Further, large agents need not burden a simple object with enormous costs; in general a large number of objects would share the agents and their overhead. The object-and-agent approach thus can enable entities of simple competence to compete in the open market.

5.3.1. Data-type agents

In object-oriented programming, one can supply multiple implementations of an abstract data type, all providing the same service through the same protocol, but offering different performance tradeoffs [28]. An example is the lookup table, which may be implemented as an array, linked list, hash table, B-tree, associative memory, or as any of several other devices or data structures. In an object-oriented system, code which *uses* such an abstract data type is itself generally abstract, being independent of how the data type is implemented; this provides valuable flexibility. In contrast, code which *requests an instance* of such an abstract data type is usually less abstract, referring directly to a class which provides a particular implementation of that type. The resulting code embodies decisions regarding implementation tradeoffs in a relatively scattered, frozen form.

In a market, agents can unfreeze these decisions: instantiation requests can be sent to a data-type agent, which then provides a suitable subcontractor. In human markets, someone seeking a house can consult a real-estate agent. The real-estate agent specializes in knowing what is available, what tradeoffs are important, and what to ask clients regarding those tradeoffs. Similarly, a lookup table agent could know what lookup table implementations are available, what tradeoffs they embody, and (implicitly, through its protocol) what to ask clients regarding those tradeoffs (*e.g.*, a request might indicate "I will often be randomly indexing into the table"). The agent could also "ask questions" by providing a trial lookup table that gathers usage statistics: once a pattern becomes clear, the agent can transparently switch to a more appropriate implementation. Long term, sporadic sampling of usage patterns can provide a low-overhead mechanism for alerting the agent to needed changes in implementation.

An agent can do more. For example, the relative price of memory and processor time may vary with the time of day or with the state of technology; depending on the cost of different implementations and the cost of switching among them, a change may be profitable. Likewise, the table-user may desire faster responses; again, a change may be profitable.

If a newly-invented lookup table implementation is superior for some uses, it could be advertised (by its advertising agent) to prominent lookup table agents. "Advertising" could include paying these agents to test its performance under different patterns of use, enabling them to determine which of their clients could benefit from switching to it. The new table would soon be used by programs that were coded without knowledge of it, and which started running prior to its creation.

Unrelated agents can interact synergistically. Consider the case of a lookup table with distinct read and write ports and distributed users. As long as there are writers, this lookup table chooses to exist on only one machine (in order to preserve serializable semantics without the complexity of distributed updates). This implementation imposes substantial delays and communication costs on the readers: if all objects dropped access to its write port, the lookup table could transmit static copies of itself to all readers, lowering these costs. The table can represent this cost by charging an artificially high retainer fee for the write port, giving users

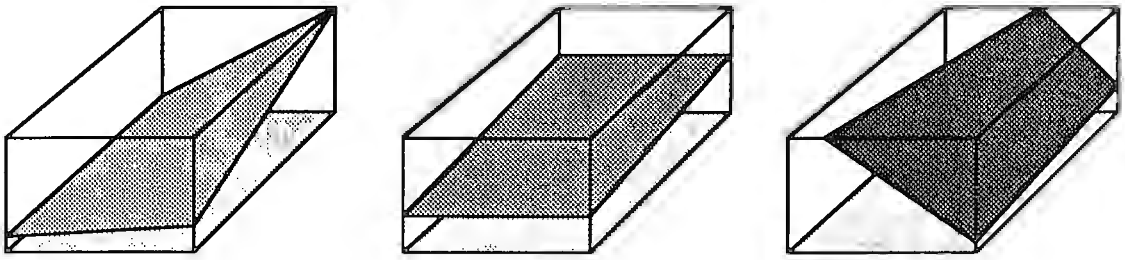
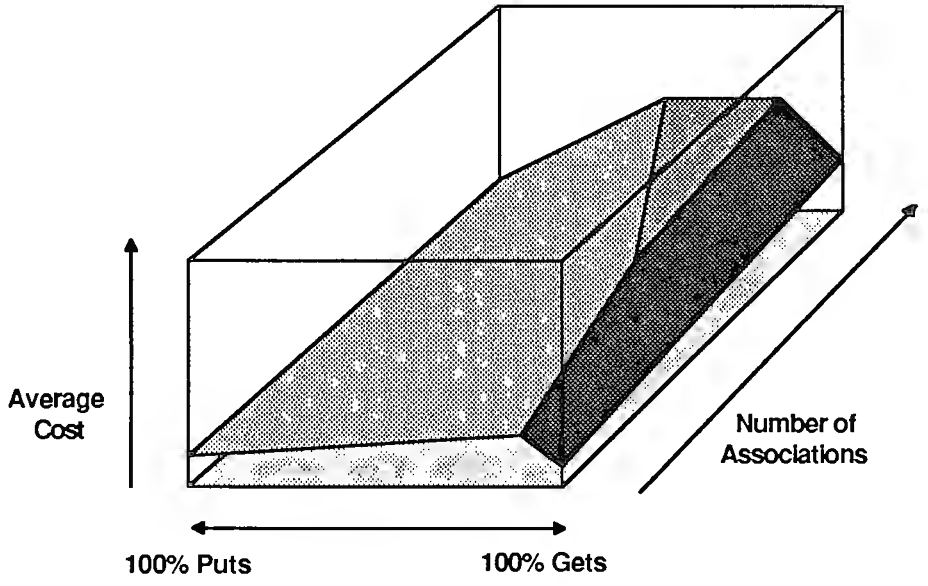


***Figure 5: Lookup table tradeoffs.** Graphs (a), (b), and (c) show processing costs for hypothetical implementations of a lookup table serving multiple sites (see 5(d) below for graph axes). Each puts key-value associations, and gets the value associated with a given key. The average processing cost of a request depends on the mix of get and put operations, and the number of associations stored. Figure 5(a) shows a linked-list implementation which puts in constant time, but gets using a linear search. Figure 5(b) shows a balanced binary tree whose costs scale as the log of the number of associations. Both 5(a) and 5(b) are centralized data structures—a table exists at only one site and all requests must travel there, adding constant overhead. Figure 5(c) shows a distributed table, replicated at each client site: gets are inexpensive, requiring no external communication, but puts are costly, requiring locking, updating, and unlocking all copies of the table.*



(d) Lookup table agent

***Figure 5(d): Lookup table agent.** Given the above choices, clients with differing patterns of use should patronize different implementations. Figure 5(d) shows costs given that a client always makes the minimum-cost choice. An ideal lookup table agent can present this cost function (plus a small cost for expenses and profit margin), relieving the client of the need make this choice.*

an incentive to drop this capability and permit a shift to the less expensive implementation. This illustrates how local economic decisions can encourage appropriate performance tradeoffs involving such distinct aspects of the system as garbage collection, network traffic, and representation of data structures.

Given sufficiently powerful partial-evaluation agents [42], a data-type agent could offer to extend an object with a new protocol. For example, the user of a lookup table might frequently look up the values of the first N entries following a particular key. Rather than doing so by running a procedure using the existing protocol, it could offer to pay for partially evaluating the procedure with respect to the lookup table, and add a lookup-next-N request to the table's protocol. This would typically make servicing such requests more efficient; a portion of the resulting savings could paid as dividends to the object that invested in the partial evaluation.

5.3.2. Managers

Different agents of the same type will have different reputations and pricing structures, and they will compete with each other. An object must select which of these agents to employ. Just as an object may employ a lookup table agent to decide which lookup table to employ, so an object may employ an agent-selection agent to decide which agent to employ. Agent-selection agents are also in competition with each other, but this need not lead to an infinite regress: for example, an object can be born with a fixed agent-selection agent. The system as a whole remains flexible, since different objects (or versions of a single object) will use different agent-selection agents. Those using poor ones will tend to be eliminated by competition.

A generalization of the notion of an agent-selection agent is that of a manager. In addition to the functions already outlined, a manager can set prices, select subcontractors, and negotiate contracts. To select good agents and subcontractors, manager-agents will need to judge reputations.

5.3.3. Reputations

A reputation system may be termed *positive* if it is based on seeking objects expected to provide good service, and *negative* if it is based on avoiding those expected to provide bad service. Negative reputation systems fail if effective pseudonyms are cheaply available; positive reputation systems, however, require only that one entity cannot claim the identity of another, a condition met by the identity properties of actors [4,43] and public key systems [29]. Accordingly, computational markets are expected to rely on positive reputation systems.

It would seem that new objects could not acquire positive reputations ("Sorry, you can't get the job unless you show you've done it before."), but they need not have *given* good service to make one *expect* good service. For example, a new object can give reason to expect good service—thereby establishing a positive reputation—by posting a cash bond guaranteeing good performance. (This requires, of course, that both parties to the contract trust some third parties to hold the bond and to judge performance.) Despite the idea that software entities cannot make commitments [44], contracts with enforceable penalty clauses provide a way for them to do so.

The demand for reputation information will provide a market for reputation services, analogous to credit rating agencies, investment newsletters, Underwriters Laboratories, the Better Business Bureau, and *Consumer Reports*. When the correctness and quality of the service can be judged, it seems that an effective reputation service could work as follows. A reputation server approaches a service provider, offering money for service. (The server funds these purchases by selling information about the results.) The reputation agent has an incentive to appear to be a regular customer (to get an accurate sample), and regular customers have an incentive to appear to be reputation agents (to get high quality service). A restaurant reviewer has been quoted as saying "If someone claims to be me, he's not!" [45]. Given unforgeable identities, the best either can do is maintain anonymity. Service providers then have an incentive to provide *consistently* good service, since their reputation might be at stake at any time. This scenario generalizes to tests of reputation services themselves.

5.3.4. Compilation

Tradeoffs in compilation can often be cast in economic terms. For example, the best choice in a time-space tradeoff will depend on processor and memory costs, and on the value of a prompt result. Another tradeoff is between computation invested in transforming code *versus* that spent in running the code; this is particularly important in guiding the often computation-intensive task of partial evaluation.

Investment in code transformation is much like other investments in an economy: it involves estimates of future demand, and hence cannot be made by a simple, general algorithm. In a computational market, compilation speculators can estimate demand, invest in program transformations, and share in the resulting savings. Some will overspend and lose investment capital; others will spend in better-adapted ways. Overall, resources will flow toward investors following rules that are well-adapted to usage patterns in the system, thereby allocating resources more effectively. This is an example of the subtlety of evolutionary adaptation: nowhere need these patterns be explicitly represented.

Current programming practice typically sacrifices a measure of structural simplicity and modularity for the sake of efficiency. Some recent compilation technologies [42,46] can make radical, non-local transformations that change not only performance, but complexity measure. Use of such technology could free programmers to concentrate on abstraction and semantics, allowing the structure of the code to more directly express the structure of the problem. This can reduce the tension between modularity and efficiency.

As we later argue, computational markets will encourage the creation of reusable, high-quality modules adapted for composition into larger objects. The resulting composite objects will typically have components paying dividends to different investors, however, imposing internal accounting overhead. Again, there is a tension with efficiency, and again it can be reduced by means of compilation technology. Much compilation involves (invisibly) "violating" internal boundaries—compiling well-separated components into a complex, non-modular piece of code. In an agoric system, a compilation-agent can do the same, while also analyzing and compiling out the overhead of run-time accounting.

A *Pareto-preferred compiler* is one which performs this transformation so as to guarantee that some component will be better off and none will be worse off. This can be achieved even if the resulting division of income only approximates the original proportions, since the total savings from compilation will result in a greater total income to divide. The expectation of Pareto-preferred results is enough to induce objects to submit to compilation; since multiple results can meet this condition, however, room will remain for negotiation.

5.4. The scandal of idle time

Current resource allocation policies leave much to be desired. One sign of this is that most computing resources—including CPUs, disk heads, local area networks, and much more—sit idle most of the time. But such resources have obvious uses, including improving their own efficiency during later use. For example, heavily-used programs can be recompiled through optimizing compilers or partial evaluators; pages on disk can be rearranged to keep files contiguous; objects can be rearranged according to referencing structure to minimize paging [47], and so forth. In a computational market, a set of unused resources would typically have a zero or near-zero price of use, reflecting only the cost of whatever power consumption or maintenance could be saved by genuine idleness or complete shutdown. Almost any use, however trivial, would then be profitable. In practice, contention for use would bid up prices until they reflected the marginal value of use [5]. Idle time is a blatant sign of wasteful resource allocation policies; one suspects that it is just the tip of a very large iceberg.

Terry Stanley [48] has suggested a technique called "post-facto simulation" as a use of idle (or inexpensive) time. It enables a set of objects to avoid the overhead of fine-grained accounting while gaining many of its advantages. While doing real work, they do no accounting and make no attempt to adapt to internal price information; instead, they just gather statistics (at low overhead) to characterize the computation. Later, when processing is cheap and response time demands are absent (*i.e.,* at "night"), they simulate the computation (based on the statistics), but with fine-grained accounting turned on. To simulate the day-time situation, they do not charge for the overhead of this accounting, and proceed using simulated "day" prices. The resulting decisions (regarding choice of data structures, use of partial evaluation, *etc.*) should improve performance during future "days". This is analogous to giving your best real-time response during a meeting, then reviewing it slowly afterward: by considering what you should have done, you improve your future performance.

6. Agoric systems in the large

In describing the idea of market-based computation and some of its implications, this paper has implicitly focused on relatively isolated systems of software performing relatively conventional functions. The following examines two broader issues: how market-based computation could interact with existing markets for software, and how it could be relevant to the goal of artificial intelligence.

6.1. Software distribution markets

An agoric open system would provide a computational world in which simple objects can sell services and earn royalties for their creators. This will provide incentives that differ from those of the present world, leading to qualitative differences in software markets.

6.1.1. Charge-per-use markets

> Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way.
>
> —R. W. Hamming, 1968 [49]

Consider the current software distribution marketplace. Producers typically earn money by charging for copies of their software (and put up with extensive illegal copying). Occasional users must pay as much for software as intense users. Software priced for intense users is expensive enough to discourage purchase by occasional users—even if their uses would be of substantial value to them. Further, high purchase prices discourage many potentially frequent users from trying the software in the first place. (Simply lowering prices would not be more efficient if this lowers revenues for the sellers: with lower expected revenue, less software would be written, including software for which there is a real demand.)

Now consider trying to build and sell a simple program which uses five sophisticated programs as components. Someone might buy it just to gain access to one of its components. How large a license fee, then, should the owners of those components be expected to charge the builder of this simple program? Enough to make the new program cost at least the sum of the costs of the five component programs. Special arrangements might be made in special circumstances, but at the cost of having people judge and negotiate each case. When one considers the goal of building systems from reusable software components, with complex objects making use of one another's services [50], this tendency to sum costs becomes pathological. The peculiar incentive structure of a charge-per-copy market may have been a greater barrier to achieving Hamming's dream than the more obvious technical hurdles.

In hardware markets, it can be better to charge for the use of a device than to sell a copy of it to the user:

> Why was [the first Xerox copier] so successful? Two thing contributed to the breakthrough, McColough says. . .technical superiority. . .and equally important, the marketing genius of the pricing concept of selling [the use of the copier], not machines. 'One aspect without the other wouldn't have worked,' he said. '. . .we couldn't sell the machines outright because they would have been too expensive.'
>
> —Jacobson and Hillkirk, 1986 [51]

Agoric systems will naturally support a charge-per-use market for software. In any market, software producers will attempt to extract substantial charges from high-volume users. With charge per use, however, the charges to be paid by high-volume users will no longer stand in the way of low-volume users; as a result, they will use expensive software that they could not afford today. At the same time, high-volume users will experience a finite marginal price for using software, rather than buying it and paying a zero marginal price for using it; they will cut back on some of their marginal, low-value uses. The overall benefit of numerous low-volume users making high-value use of the software will likely outweigh the loss associated with a few high-volume users cutting back on their low-value uses, yielding a net social benefit. It seems likely that some of this benefit will appear as increased revenues to software producers, encouraging increased software production.

In a charge-per-copy market, users face an incentive structure in which they pay nothing to keep using their present software, but must pay a large lump sum if they decide to switch to a competitor. A charge-per-use market will eliminate this artificial barrier to change, encouraging more lively competition among software producers and better adaptation of software to user needs.

By enabling small objects to earn royalties for their creators, charge-per-use markets will encourage the writing, use, and reuse of software components—to do so will finally be profitable. Substantial improvement in programming productivity should result; these improvements will multiply the advantages just described.

6.1.2. Hardware encapsulation

This charge-per-use scenario presents a major technical problem: it depends on the ability to truly protect software from illicit copying. True encapsulation would ensure this, but true encapsulation will require a hardware foundation that blocks physical attacks on security. Two approaches seem feasible: either keeping copies in just a few secure sites and allowing access to their services over a network, or developing a technology for providing users with local secure sites to which software can migrate.

In the limit of zero communication costs (in terms of money, delay, and bandwidth limitations), the disincentive for remote computation would vanish. More generally, lower communication costs will make it more practical for objects located on remote machines to offer services to objects on user machines. Remote machines can provide a hardware basis for secure encapsulation and copy protection—they can be physically secured, in a vault if need be. This approach to security becomes more attractive if software can be partitioned into public-domain front-ends (which engage in high-bandwidth interaction with a user), and proprietary back-ends (which perform sophisticated computations), and if bandwidth requirements between front- and back-ends can be minimized.

One system that might lend itself to this approach is an engineering service [13]. The user's machine would hold software for the representation, editing, and display of hardware designs. The back-end system—perhaps an extensive market ecosystem containing objects of diverse functionality and ownership—would provide computation-intensive numerical modeling of designs, heuristics-applying objects (perhaps resembling expert systems) for suggesting and evaluating modifications, and so forth.

Two disadvantages of separating front- and back-ends in this way are communications cost and response time. If hardware encapsulation can be provided on the local user's machine, however, software can migrate there (in encrypted form) and provide services on-site. Opaque boxes are a possible design for such secure hardware:

Imagine a box containing sensors and electronics able to recognize an attempt to violate the box's integrity [52]. In addition, the box contains a processor, dynamic RAM, and a battery. In this RAM is the private key of the manufacturer's public-key encryption key pair [29]; objects encrypted with the public key can migrate to the box and be decrypted internally. If the box detects an attempt to violate its physical integrity, it wipes the dynamic RAM (physically destructive processes are acceptable), deleting the private key and all other sensitive data. All disk storage is outside the box (fast-enough disk erasure would be too violent), so software and other data must be encrypted when written and decrypted when read. The box is termed *opaque* because no one can see its contents.

Internally, the opaque box would require encapsulation among software objects. This can be done by using a secure operating system [VI], by using capability hardware [53,54,55], or by demanding that objects be written in a secure programming language and either run under a secure interpreter or compiled by a secure compiler [IV,56]. Among other objects, the box would contain one or more branches of external banks, linked to them from time to time by encrypted communications; these banks would handle royalty payments for use of software.

Will greater hardware cost make opaque boxes uncompetitive for personal computer systems? If the added cost is not too many hundreds of dollars, the benefit—greater software availability—will be far greater, for many users. Opaque boxes can support a charge-per-use market in which *copies* of software are available for the cost of telecommunications. CD-ROMs full of encrypted software might be sold at a token cost to encourage use.

An intermediate approach becomes attractive if opaque boxes are too expensive for use as personal machines. Applications could be split into front and back-ends as above, but back-ends could run on any available opaque box. These boxes could be located wherever there is sufficient demand, and linked to personal machines via high-bandwidth local networks. People (or software) would find investment in opaque boxes profitable, since their processors would earn revenue. With high enough box-manufacturing costs, this approach merges into the remote-machine scenario; with low enough costs, it merges into the personal-machine scenario.

6.1.3. Inhibiting theft

As society embodies more and more of its knowledge and capabilities in software, the theft of this software becomes a growing danger. An environment that encourages the creation of large, capable, stand-alone applications sold on a charge-per-copy basis magnifies this problem, particularly when the stolen software will be used in places beyond the reach of copyright law.

A charge-per-use environment will reduce this problem. It will encourage the development of software systems that are composites of many proprietary packages, each having its security guarded by its creator. Further, it will encourage the creation of systems that are distributed over many machines. The division and distribution of functions will make the problem faced by a thief less like that of stealing a car and more like that of stealing a railroad. Traditional methods of limiting theft (such as military classification) slow progress and inhibit use; computational markets promise to discourage theft while speeding progress and facilitating use.

6.1.4. Integration with the human market

It has been shown how an agoric system would use price mechanisms to allocate use of hardware resources among objects. This price information will also support improved decisions regarding hardware purchase: if the market price of a resource inside the system is consistently above the price of purchasing more of the resource on the external market, then incremental expansion is advantageous. Indeed, one can envision scenarios in which software objects recognize a need for new hardware, lease room for it, and buy it as an investment.

It has been shown how objects in an agoric system would serve human needs, with human minds judging their success. Similarly, when objects are competent to judge success, they can hire humans to serve their needs—for example, to solve a problem requiring human knowledge or insight.

Conway's law states that "Organizations which design systems are constrained to produce systems which are copies of the communications structures of these organizations" (from [57] as quoted in [58]). If so, then software systems developed in a distributed fashion can be expected to resemble the organization of society as a whole. In a decentralized society coordinated by market mechanisms, agoric systems are a natural result.

6.2. The marketplace of mind

Artificial intelligence is unnecessary for building an agoric open system and achieving the benefits described here. Building such a system may, however, speed progress in artificial intelligence. Feigenbaum's statement, "In the knowledge lies the power", points out that intelligence is knowledge-intensive; the "knowledge acquisition bottleneck" is recognized as a major hindrance to AI. Stefik has observed [VII] that this knowledge is distributed across society; he calls for a "knowledge medium" in which knowledge contributed by many people could be combined to achieve greater overall intelligence.

Agoric systems should form an attractive knowledge medium. In a large, evolving system, where the participants have great but dispersed knowledge, an important principle is: "In the incentive structure lies the power". In particular, the incentives of a distributed, charge-per-use market can widen the knowledge engineering bottleneck by encouraging people to create chunks of knowledge and knowledge-based systems that work together.

Approaches based on directly buying and selling knowledge [VII,23] suffer from the peculiar incentives of a charge-per-copy market. This problem can be avoided by embodying knowledge in objects which sell knowledge-based services, not knowledge itself. In this way, a given piece of knowledge can be kept proprietary for a time, enabling producers to charge users fees that approach the value the users place on it. This provides an incentive for people to make the knowledge available. But in the long run, the knowledge will spread and competition will drive down the price of the related knowledge-based services—approaching the computational cost of providing them.

Agoric open systems can encourage the development of intelligent objects, but there is also a sense in which the systems themselves will become intelligent. Seeing this entails distinguishing between the idea of *intelligence* and the ideas of individuality, consciousness, and will. Consider the analogous case of human society.

It can be argued that the most intelligent system now known is human society as a whole. This assertion strikes some people as obvious, but others have a strong feeling that society should be considered less intelligent than an individual person. What might be responsible for these conflicting views?

The argument for the stupidity of society often focuses not on the achievements of society, but on its suboptimal structure or its slow rate of structural change. This seems unfair. Human brains are presumably suboptimal, and their basic structure has changed at a glacial pace over the broad time spans of biological evolution, yet no one argues that society is worse-structured than a brain (what would this mean?), or that its basic structure changes more slowly than that of a brain. Great intelligence need not imply optimal structure, and suboptimal structure does not imply stupidity.

Other arguments for the stupidity of society focus on the behavior of committees, or crowds, or electorates. This also seems unfair. Human beings include not only brains but intestines; our intelligence is not to be judged by the behavior of the latter. Not all parts need be intelligent for a system to be so. Yet other arguments focus on things individuals can do that groups cannot, but one might as well argue that Newton was stupid because he did not speak Urdu. A final argument for the stupidity of society focuses on problems that result when a few individuals who are thought to somehow *represent* society attempt to direct the actions of the vast number of individuals who actually *compose* society—that is, the problems of central planning, government, and bureaucracy. This statement of the argument seems an adequate refutation of it.

The argument for society's intelligence is simple: people of diverse knowledge and skills, given overall guidance by the incentives of a market system, can accomplish a range of goals which, if accomplished by an individual, would make that individual a super-human super-genius. The computer industry is a small part of society, yet what individual could equal its accomplishments, or the breadth and speed of its ongoing problem-solving ability?

Still, it is legitimate to ask what it means to speak of the "intelligence" of a diverse, distributed system. In considering an individual, one commonly identifies intelligence with the ability to achieve a wide range of goals through complex information processing. But in agoric systems, as in human society, the component entities will in general have diverse goals, and the system as a whole will typically have no goals [59]. Nonetheless, a similar concept of intelligence can be applied to individuals, societies, and computational markets.

Individuals taking intelligence tests are judged by their ability to achieve goals set by a test-giver using time provided for the purpose. Likewise, the intelligence of a society may be judged by its ability to achieve goals set by individuals, using resources provided for the purpose. In either case, the nature and degree of intelligence may be identified with a combination of the *range* of goals that can be achieved, the speed with which they can be achieved, and the efficiency of the means employed. By this measure, one may associate kinds and degrees of intelligence not only with individuals, but with corporations, with *ad-hoc* collections of suppliers and subcontractors, and with the markets and institutions that bring such collections together at need. *The idea of intelligence may thus be separated from the ideas of individuality, consciousness, and will.*

The notion of intelligence emerging from social interactions is familiar in artificial intelligence: Minsky [60] uses the society metaphor in his recent work on thinking and the mind; Kornfeld and Hewitt [61] use the scientific community as a model for programs incorporating due process reasoning [II]. Human societies demonstrate how distributed pieces of knowledge and competence can be integrated into larger, more comprehensive wholes; this process has been a major study of economics [8] and sociology [63]. Because these social processes (unlike those in the brain) involve the sometimes-intelligible interaction of visible, macroscopic entities, they lend themselves to study and imitation. This paper may thus be seen as proposing a form of multi-agent, societal approach to artificial intelligence.

7. The absence of agoric systems

Market-style software systems are a fairly obvious idea and have received some attention. However, in considering any fairly-obvious idea with (allegedly) great but unrealized potential, it is wise to ask why that potential has in fact not been realized. When an idea of this sort neither lends itself to formal proof nor to small, convincing demonstrations, the difficulty of making a case for it grows. Support from abstract arguments and analogies can be helpful, as can an examination of the practical issues involved. But in addition, it helps to see whether the idea has been tested and found wanting. Considering this major category of possible negative evidence is an aspect of due-process reasoning.

Why have agoric open systems not been implemented already? In part, because the software community has lacked an immediate, compelling need. Advances have been made, through better programming environments and methodologies (including the encapsulation and communication of information and access), and through tools for making larger structures visible to programmers [64]—all without building markets. These environments and methodologies have extended the programmer's conceptual span of control, enabling one mind or a few closely-coordinated, mutually-trusting minds to build ever larger and more complex programs. These advances have decreased the urgency of enabling extensive cooperation *without* mutual trust or extensive communications.

Another problem has been the scale-sensitivity of the market approach. In small systems, the overhead of accounting and negotiations is unjustified; further, incremental increases in scale have thus far been possible without markets. Robust service-trading objects must have a certain minimum complexity, or have access to trusted business-agents of a certain minimum complexity. The virtues of markets are greatest in large, diverse systems.

There has, perhaps, also been a cultural factor at work. Large, research-oriented computer networks have focused on academic and government work—that is, toward non-profit use. Further, the academic community already has an informal incentive structure that rewards the creators of useful software in an incremental way, in rough proportion to its usefulness. These reputation-based reward mechanisms facilitate the development of software systems that build on others' work; the differing incentives in the commercial community may be responsible for its greater tendency to build redundant systems from scratch.

These considerations seem sufficient to explain the lack of agoric systems today, while giving reason to expect that they will become desirable as computer systems and networks grow. In the large, open, evolving software systems of the future, the overhead of accounting will be less important than robustness and flexibility. Further, the development of automated programming systems will introduce "programmers" having (initially) a sharply limited ability to plan and comprehend. This will re-emphasize the problem of the "programmer's" span of conceptual control, and increase the need for mechanisms that strengthen localization and system robustness.

8. Conclusions

A central challenge of computer science is the coordination of complex systems. In the early days of computation, central planning—at first, by individual programmers—was inevitable. As the field has developed, new techniques have supported greater decentralization and better use of divided knowledge. Chief among these techniques has been object-oriented programming, which in effect gives property rights in data to computational entities. Further advance in this direction seems possible.

Experience in human society and abstract analysis in economics both indicate that market mechanisms and price systems can be surprisingly effective in coordinating actions in complex systems. They integrate knowledge from diverse sources; they are robust in the face of experimentation; they encourage cooperative relationships; and they are inherently parallel in operation. All these properties are of value not just in society, but in computational systems: markets are an abstraction that need not be limited to societies of talking primates.

This paper has examined many of the concrete issues involved in actually creating computational markets, from hardware and software foundations, to initial market strategies for resource management (chiefly in [III]), to the organization of systems of objects and agents able to interact in a market context. As yet, no obstacle to their realization has been found.

Distributed systems based on the charge-per-use sale of software services and computational resources promise a more flexible and effective software market, in which large systems will more often be built from pre-existing parts. With many minds building knowledge and competence into market objects, and with incentives favoring cooperation among these objects, the overall problem-solving ability of the system can be expected to grow rapidly.

On a small scale, central planning makes sense; on a larger scale, market mechanisms make sense. Computer science began in a domain where central planning made sense, and central planning has thus been traditional. It seems likely, however, that some modern computer systems are already large and diverse enough to benefit from decentralized market coordination. As systems grow in scale and complexity, so will the advantages of market-based computational systems.

Appendix I. Issues, levels, and scale

This appendix explores how various computational issues change character from lower to higher levels of a system (in the sense described in Section 3.7). Agoric open systems can most easily be developed by building up from current systems—by finding ways to make a smooth transition from current programming practice to the practices appropriate to market ecosystems. (One aspect of this is dealt with in [III].) Understanding how issues will change from level to level will aid this process and minimize the chance of misapplying concepts from one level to problems on another level.

Higher levels of organization will raise issues not so much of system correctness as of system coherence. For example, while a sorting algorithm may be *correct* or *incorrect,* a large collection of software tools may be *coherent* or *incoherent*—its parts may work together well or poorly, even if all are individually correct. The notion of coherence presumes a level of complexity that makes it inapplicable to a sorting algorithm. Despite the differences between correctness and coherence, they have much in common: correctness can be seen as a formal version of coherence, one appropriate for small-scale objects. In this, as in many of the following issues, hard-edged criteria at lower levels of organization have soft-edged counterparts at higher levels.

	low level . . . high level
Economics	planning . . . spontaneous order
Security	encapsulation . . . skepticism
Compatibility	message passing . . . operability
Degrees of trust	trust . . . reputations
Reasoning	. logic . . . due process
Coordination	serialization . . . negotiation

Figure 6: Changes in character of issues across levels.

I.1. Security

Alan Kay has characterized compatibility, security, and simplicity as essential properties for building open systems. For mutually untrusting objects to interact willingly, they must be secure. Encapsulation can provide security at a low level, as a formal property of computation. With this property, one can code an object so that the integrity of an internal data structure is guaranteed despite possible nonsense messages. Security at a high level involves skepticism and the establishment of effective reputation systems. Skepticism enables an object to continue reasoning coherently despite being told occasional lies.

Encapsulation—in this case, protection against tampering—is necessary for skepticism to work. Without encapsulation, a skeptical object's intellectual defenses could be overcome by the equivalent of brain surgery.

I.2. Compatibility

Compatibility allows objects to be mutually intelligible, despite diverse origins. At a foundational level, it involves a shared message passing medium and mutual understanding of some protocol. Inside a small program written by a single programmer, objects can be carefully crafted so that any two that communicate will necessarily use the same protocol. Between large objects written by different people, or the same person at different times, checking for protocol agreement can frequently prevent disaster. For example, if an object is passed a reference to a lookup table when it is expecting a number, it may help to learn that "addition" will not be understood by the table before actually attempting it. Note that this itself relies on agreement on a basic protocol which provides a language for talking about other protocols.

In the Xerox Network System, clients and servers not only compare the type of protocol that they can speak, but the range of protocol versions that they understand [65]. If their ranges overlap, they then speak the latest mutually understood version. If their ranges do not overlap, they then part and go their separate ways. This is an example of bootstrapping from a mutually understood protocol to determine the intelligibility of other protocols. The developing field of interoperability [66] should soon provide many more.

Sophisticated objects should eventually have still broader abilities. Human beings, when faced with a novel piece of equipment, can often learn to make profitable use of unfamiliar capabilities. Among the techniques they use are experimentation, reading documentation, and asking a consultant. One may eventually expect computational analogues [67].

I.3. Degrees of trust

Security is needed where trust is lacking, but security involves overhead; this provides an incentive for trust. At a low level, a single author can create a community of trusting objects. At an intermediate level trust becomes more risky because error becomes more likely. This encourages error-checking at internal interfaces, as is wise when a team of programmers (or one forgetful programmer) must assemble separately developed modules.

At higher levels, strategic considerations can encourage partial trust. A set of objects may make up a larger object, where the success of each depends on the success of all. Here, objects may trust each other to further their joint effort [68]. Axelrod's iterated prisoner's dilemma tournament [69] (see also [I]) shows another way in which strategic considerations can give rise to trust. One object can generally expect cooperative behavior from another if it can arrange (or be sure of) appropriate incentives.

In a simple iterated prisoner's dilemma game, this requires both having a long-term relationship and paying the overhead of noticing and reacting to non-cooperative behavior. Reputation systems within a community can extend this principle and lower the overhead of using it. Some objects can gather and sell information on another object's past performance: this both provides incentives for consistently good performance and reduces the cost of identifying and avoiding bad performers. In effect, reputation systems can place an object in an iterated relationship with the community as a whole.

The idea that encapsulation is needed at low levels for security, where we also expect complete trust seems to entail a conflict. But the function of encapsulation is to protect simple objects where trust is limited or absent (as it will be, between some pairs of objects). Complete trust makes sense among simple objects that are in some sense playing on the same team.

I.4. Reasoning

Programming language research has benefited from the methodology of formalizing programming language semantics. A result is the ability to reason confidently (and mechanistically) about the properties of programs expressed in such languages. This can establish confidence in the correctness of programs having simple specifications. The logic programming community is exploring the methodology of transforming a formal specification into a logic program with the same declarative reading. The resulting logic program is not generally guaranteed to terminate, but if it does, it is guaranteed to yield a correct result, since the interpreter is a sound (though incomplete) theorem prover and the program is a sound theorem.

Deductive logic seems inadequate as a high-level model of reasoning, though there is much controversy about this. High level reasoning involves weighing pro and con plausibility arguments (due-process reasoning [II]), changing one's mind (non-monotonicity), believing contradictory statements without believing all statements, and so forth. There have been attempts to "fix" logic to be able to deal with these issues, but [70] argues that these will not succeed. A more appropriate approach to high level reasoning emphasizes coherence, plausibility, and pluralism instead of correctness, proof, and facts. (This does not constitute a criticism of logic programming: logic programming languages, like lambda-calculus languages, can express arbitrary calculations, including those that embody non-logical modes of reasoning.)

I.5. Coordination

In order to coordinate activity in a concurrent world, one needs a mechanism for serialization. Semaphores [71] and serialized actors [IV,3,4] enable a choice between processes contending for a shared resource; these primitives in turn make possible more complex concurrency control schemes such as monitors [19] and receptionists [4], which allow the protected resource to interact with more than one process at a time. Monitors in turn have been used to build distributed abortable transactions (as in Argus, described elsewhere in this volume [V]), which support coherent computation in the face of failure by individual machines.

For very large distributed systems, transaction-based coordination requires too much consistency over too many participants. Dissemination models [38,39,72], and publication models [23,73] provide mechanisms that apply to larger scales.

The Colab is another project which has extended notions of coordination control. Colab is a project to build a collaborative laboratory—a multi-user interactive environment for supporting collaborative work [74]. In the Colab, a group of people work together on a set of data and sometimes contend for the right to modify the same piece of data. Initial attempts to deal with this by simply scaling up transactions proved unsuitable. Instead, *social-coordination* mechanisms were found, such as signals to indicate someone's interest in changing a piece of data. The applicability of these mechanisms is not human-specific, but should generalize to any situation in which there is often a significant investment in computation which would be thrown away by an aborted transaction.

An essential aspect of higher-level coordination mechanisms is negotiation. When allocating exclusive access to a resource for a millisecond, it often makes sense to rely on simple serialization. When allocating exclusive access for a year, it often makes sense to take greater care. One simple form of negotiation is an auction—a procedure in which the resource is allocated to the highest bidder. Hewitt in [75] explores Robert's Rules of Order as the basis for more sophisticated negotiation procedures.

Even sophisticated negotiation mechanisms will often rely on primitive serializers. In auctions, an auctioneer serializes bids; in Robert's Rules, the chair serializes access to the floor.

I.6. Summary

This section has examined how a range of issues—security, compatibility, trust, reasoning, and coordination—may appear at different levels of market-based open systems. Certain themes have appeared repeatedly. Mechanisms at low levels often support those at higher levels, as (for example) high-level coordination mechanisms using simple serializers. Further, higher levels can inherit characteristics of lower levels, such as encapsulation and conservation laws.

Issues often blur at the higher levels—security and trust become intertwined, and may both depend on due-process reasoning. The bulk of this paper concentrates on low- and mid-level concerns which must be addressed first, but high-level issues all present a wealth of important research topics.

Appendix II. Comparison with other systems

This section, and these papers, discuss and criticize many works. We wish to emphasize that *these* works have been chosen, not for their flaws, but for their value.

II.1. The Xanadu hypertext publishing system

This paper has compared agoric systems to other systems for computation. Our first exposure to many of the central ideas of markets and computation, however, stems from our work with the Xanadu hypertext system [23]. Xanadu is a proposed on-line publishing medium for hypertext documents. A hypertext document differs from the normal notion of a document in that it contains *links,* connections between documents which readers can follow at the click of a mouse. Published documents thus form not a set of disconnected islands, but a web connected by references, quotes, criticisms, comments, and rebuttals.

How can a reader find a path in such an interconnected web? Rather than proposing that someone (somehow) create a single, official system index, the Xanadu project proposes support for decentralized indexing, and hence for pluralism. Any reader can author and publish a guide to any set of public documents. Other readers can then use this guide to sort material and orient themselves. Anyone can, of course, publish guides to other guides. Xanadu relies on the expectation that this activity will result in a spontaneous order—a richly-connected world of documents in which readers can find their way.

Why will indexing be done where needed? In part because readers will do much of the basic searching and sorting for themselves, and then publish the results (since publishing is easy). In addition, however, Xanadu provides a charge-per-read royalty arrangement to encourage publication of material for which there is a demand. Just as charge-per-use software will make it economical to assemble software from diverse components, so Xanadu's royalty arrangement is designed to encourage the assembly of documents from parts of other documents: if one document quotes another, a reader's royalty payments are split between them.

In Xanadu, documents are passive data. One way of conceiving of agoric systems is as a publishing medium for linked, active data.

II.2. Knowledge medium

Mark Stefik's "Knowledge Medium" paper [VII] paints a visionary future in which AI systems, distributed across society, are able to communicate and share knowledge. In contrast, current expert systems are seen as isolated systems rebuilt from scratch each time. A knowledge medium would enable individual systems to specialize in encoding the knowledge most relevant to them, and would provide a market for the purchase of knowledge represented elsewhere. As a result, the process of encoding knowledge is expected to accelerate through division of labor and economies of scale.

This proposal is compatible with the agoric systems vision, but has a somewhat different emphasis. Stefik's paper emphasizes representing knowledge, communicating representations, and integrating representations together. While we certainly expect (and hope) that all this would occur in an agoric system, this work emphasizes the sale of knowledge-based services.

In Stefik's vision, a "knowledge provider" responds to a request by sending a representation of the knowledge it specializes in. The consumer is then faced with the task of relating this representation to its own. This problem would create a market for "knowledge integrators". In the model sketched in this paper, knowledge is "represented" by embodying it in objects that apply their knowledge to provide services. Consumers would then be integrating the *results* in order to provide further services.

Because of the copying problem, a market for services should be more effective than a market for representations. Once knowledge is transmitted, it will often spread without further rewarding its creators. This reduces the incentives for knowledge creation.

II.3. Enterprise Net

Enterprise [VIII], by Malone, provides decentralized scheduling of tasks in a network of personal workstations by making use of market-like mechanisms. A *client* processor with a task to be scheduled broadcasts a request for bids to *contractor* processors. Available contractors respond with *bids;* these are evaluated by the client, which then sends the task to the best bidder. The client's request includes characteristics of the task which are pertinent in estimating its processing time. The best bidder is generally the contractor who responds with the earliest estimated completion time. This bidding protocol provides for decentralized decision making and enables clients to use their own criteria in evaluating candidate suppliers.

Compared to the agoric systems approach, Enterprise has several limitations. It assumes full mutual trust between clients and contractors, all working toward a common objective. It is also less flexible in the tradeoffs it can make—the system contains non-adaptable system parameters and uses no price mechanism. Lacking price signals, the system relies on prearranged, non-evolving rules to guide behavior. The inflexibility of such a system is illustrated by the following example.

Imagine two client tasks: a high-priority theorem proving task and a lower-priority fluid flow simulation task, and two server machines: a Vax 780 with an attached array processor and a Vax 750 without one. Both tasks prefer the 780 because it is faster, but the simulation task vastly prefers it because of the array processor; in comparison, the theorem prover is relatively indifferent. In Enterprise, both will try to get the 780, and the 780 will be allocated to the higher priority theorem prover. In an agoric system, however, the simulation task might offer only a trivial amount of money for the 750, resulting in a sufficiently lower market price that the theorem prover finds the bargain worth taking. Alternatively, if the theorem prover is already running on the 780, the simulation task could offer to pay it to migrate to the 750. This is but one example of the flexibility that market prices can bring to a system. Malone acknowledges that it may be useful to provide a price system within his framework.

II.4. Malone's comparison of organizational structure

Malone [76] has also compared various organizational structures for coordinating communities of agents. A strong similarity between Malone's work and ours is the attempt to recognize parallel organizational forms in human societies and computer systems.

Malone sees markets as capable of providing efficient solutions to the problems of decentralized resource allocation in computer systems, as they have done in human organizations. He also maintains that factors existing in human societies which limit the optimality of markets can be excluded from software systems.

Transaction costs—such as expenses involved in trading on the market—limit the use of markets and encourage the use of other forms of human organization, such as hierarchies. These transaction costs increase in uncertain or complex markets. Traders must protect themselves from other opportunistic traders, usually by establishing contracts; negotiating such contracts (and living with their consequences) imposes important transaction costs.

Malone assumes that these costs will be absent from computer systems, arguing that "While non-opportunistic traders may be rare in human markets, there is no reason at all why computer programs cannot be constructed with [non-opportunistic] participants in a market-like organization." This may be so for non-evolving computational entities authored by an individual or team. In an open distributed system, however, the programs will themselves be authored by a diversity of people who will in fact have opportunistic motives with respect to each other; further, EURISKO-like systems [IX,77] may evolve software subject only to the constraint of market success. A system designed under the assumption of non-opportunistic participants can be effectively used only within limited contexts—roughly speaking, within a single firm.

II.5. Harris, Yu, and Harris's market-based scheduling algorithm

Harris, Yu, and Harris have applied simulated markets to difficult factory scheduling problems. Although total optimality can be defined in this case, finding it is known to be NP-hard [78], and their initial results indicate that Pareto optimal schedules are very good by most conventional measures. In their approach, the requirements, constraints, and tradeoffs for scheduling an individual order are represented by a utility function. These utility functions can express many of the "arbitrary" constraints typical of a real factory, such as a requirement that one step follow another within a given time limit. By having these utility functions interact to set prices, a Pareto optimal solution is found relatively quickly by local hill climbing. "In less than a minute [this algorithm] can schedule an order requiring 150 processing steps over 90 resources" [78]. This system, while not allowing for evolution of scheduling strategies, demonstrates the value of a market model for directing resource allocation by computational means.

The representation language for expressing the preferences of an individual order are quite flexible, but less flexible than a general purpose programming language. This loss does confer certain advantages: opportunistic behaviors are impossible, and the algorithm can compose preferences via an efficient dynamic programming technique. Their algorithm thus creates a *computational market simulation*, rather than a *computational market;* it might find a role *within* a market by offering objects a low-overhead scheduling service, guided by external market prices.

II.6. Sutherland's time sharing system

In "A Futures Market in Computer Time" [79], I. E. Sutherland describes a bidding mechanism (implemented in the medium of paper) that results in computer resources being allocated according to the users' priorities. Users compete for computer time by making bids for specific blocks of time, with the bidding currency being tokens which are assigned to users according to their relative priority. A bid can be pre-empted by a higher bid. Since higher priority users have more tokens to bid with, they are able to outbid the lower priority users. Being outbid, a user might then try for a "cheaper" block of time during a less desirable period of the day.

By having the price of a time period vary with demand, more efficient resource allocation is possible. There are, however, restrictions placed on the users—users cannot trade tokens or lower a bid—that limit the flexibility of this system.

II.7. Connectionism and genetic algorithms

Two recent uses of spontaneous order principles in software are connectionism (also known as artificial neural systems or parallel distributed processing models) [80] and genetic algorithms [81]. The first draws its inspiration from models of how neural networks may operate, the second from genetically-based biological evolution. Both systems have shown impressive abilities to learn to recognize patterns in noisy data. Knowledge of these patterns does not have to be designed in *a priori* by some human designer. Rather, these systems are able to sift patterns from the data itself. Though this results in these systems "knowing" the pattern, it is nowhere explicitly represented—they do not know what patterns they know.

These systems and the agoric approach share certain similarities. All are spontaneous order systems engaging in distributed representation and adapting to changing circumstances in part by adjusting (and passing around) numeric weights. Some aspects of genetic algorithms are explicitly based on a market metaphor [82], and Barto proposes connectionist models based on networks of self-interested units [83].

All these systems learn (in part) by increasing numeric weights associated with components that have contributed to overall success. A problem that needs to be addressed by such a learning algorithm is the division of rewards when several components have together contributed to a joint success. Minsky writes:

> It is my impression that many workers in the area of 'self-organizing' systems and 'random neural nets' do not feel the urgency of this problem. Suppose that one million decisions are involved in a complex task (such as winning a chess game). Could we assign to each decision one-millionth of the credit for the completed task?. . .For more complex problems, with decisions in hierarchies. . .and with increments small enough to assure probable convergence, the running times would become fantastic.

Minsky wrote this in 1961 [84]. Despite the current progress of connectionism and genetic algorithms, he still considers this criticism essentially correct [85].

A capable learning system should be able to learn better credit assignment mechanisms. In an agoric system, when several objects are about to work together to produce some result, they can negotiate the division of profits and risk. Among simple objects, and early in the evolution of an agoric system, this negotiation might generally be handled by simple initial strategies that may be no more flexible than the "back propagation" [80] and "bucket-brigade" [81] algorithms employed by some connectionist and genetic-algorithm systems. As the system develops, market competition will reward objects which employ more sophisticated negotiating strategies that better reflect both the value derived from the various contributors, and what their competitors are offering.

Both connectionism and genetic algorithms try to *substitute* spontaneous order principles for design—individual, competing units within such systems are not large programs designed by conventional means. There is much to be gained both from design and evolution; the agoric systems approach has been designed to use the strengths of both.

II.8. Summary

In summary, though the marketplace has often been used as a metaphor, it has generally not been used as a real model—these systems are not true computational markets. Attempts to copy patterns which have emerged in markets entail a loss of flexibility compared with using markets themselves. This criticism is analogous to the connectionist criticism of representationalist cognitive models [80]—that by attempting to model emergent patterns while discarding the foundations which made them possible, representationalist models are overly "brittle", sacrificing flexibility and learning ability.

Acknowledgments

Since 1983, when we started exploring computational markets, many people have contributed their insights on these concepts. We thank the following for helpful suggestions on agoric systems and these papers: Agustin Araya, Yeshayahu Artsy, Jim Bennett, Peter Bishop, Daniel Bobrow, John Seely Brown, Andrew Cameron, Peter Deutsch, Mike Dixon, Tom Finholt, Mike Fischer, Bob Flegal, Felix Frayman, David Friedman, Milton Friedman, Stuart Greene, Roger Gregory, Robert Gruber, Eric Gullicson, Ken Haase, Robin Hanson, Jed Harris, Rich Hasha, Keith Henson, Karl Hess, Carl Hewitt, Chris Hibbert, Tad Hogg, Bernardo Huberman, Gene Hughes, Ted Kaehler, Ken Kahn, Kirk Kelley, Scott Kim, Bill Kornfeld, David Lindbergh, Pattie Maes, Thomas Malone, John McCarthy, Diana Merry, Marvin Minsky, Ted Nelson, Gayle Pergamit, Alan Perlis, Chris Peterson, Harry Pyle, Jim Rauen, Jonathan Rees, Ramana Rao, Phil Salin, Allan Schiffman, Ehud Shapiro, Jeff Shrager, Randy Smith, Terry Stanley, Mark Stefik, Richard Steiger, Debbie Tatar, Eric Tribble, Dave Ungar, Steve Witham, Chee Yu, and Frank Zdybel.

For providing the ideas which inspired this work, we thank Carl Hewitt, Marvin Minsky, Ted Nelson, Doug Lenat, Robert Axelrod, Richard Dawkins, and most especially Friedrich Hayek.

For having arranged to make this research possible, we thank Jonathan Schmidt, Vic Poor, Charlie Smith, Mark Stefik, the Datapoint Technology Center, the Foresight Institute, the MIT Artificial Intelligence Laboratory, the Stanford Artificial Intelligence Laboratory, and the Xerox Palo Alto Research Center.

Mark S. Miller dedicates his contributions to these papers to his uncle

Henry I. Boreen

who started him on the road to intelligence.

References

Papers referenced with roman numerals can be found in the present volume:

Huberman, Bernardo (ed.), *The Ecology of Computation* (Elsevier Science Publishers/North-Holland, 1988).

[I] Miller, Mark S., and Drexler, K. Eric, "Comparative Ecology: A Computational Perspective", this volume.

[II] Hewitt, Carl, "Offices are Open Systems", this volume.

[III] Drexler, K. Eric, and Miller, Mark S., "Incentive Engineering for Computational Resource Management", this volume.

[IV] Kahn, Kenneth, and Miller, Mark S., "Language Design and Open Systems", this volume.

[V] Liskov, Barbara, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", this volume.

[VI] Rashid, Richard, "From RIG to Accent to Mach: The Evolution of a Network Operating System", this volume.

[VII] Stefik, Mark, "The Next Knowledge Medium", this volume.

[VIII] Malone, Thomas W., Fikes, R. E., and Howard, M. T., "Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments", this volume.

[IX] Lenat, Douglas B., and Brown, John Seely, "Why AM and EURISKO Appear to Work", this volume.

[1] Popper, Karl R., *Objective Knowledge: An Evolutionary Approach* (Oxford University Press, London, 1972).

[2] Hayek, Friedrich A., *The Counter-Revolution of Science: Studies on the Abuse of Reason* (Liberty Press, Indianapolis, 1979).

[3] Clinger, Will, *Foundations of Actor Semantics* (MIT, Cambridge, MA, May 1981) MIT AI-TR-633.

[4] Agha, Gul, *Actors: A Model of Concurrent Computation in Distributed Systems* (MIT Press, Cambridge, MA, 1986).

[5] Alchian, Armen A., and Allen, William R., *University Economics* (Wadsworth, Belmont, CA, 1968, Second Edition).

[6] Hayek, Friedrich A., *The Constitution of Liberty* (University of Chicago Press, Chicago, 1960) p.156.

[7] Smith, Adam, *An Inquiry into the Nature and Causes of The Wealth of Nations* (University of Chicago Press, Chicago, 1976) p.531.

[8] Hayek, Friedrich A., "Competition as a Discovery Procedure", in: *New Studies in Philosophy, Politics, Economics and the History of Ideas* (University of Chicago Press, Chicago, 1978) p.179–190.

[9] Hayek, Friedrich A., "Economics and Knowledge", from: *Economica, New Series*

(1937), Vol. IV, pp.33–54; reprinted in: Hayek, Friedrich A., (ed.), *Individualism and Economic Order* (University of Chicago Press, Chicago, 1948).

[10] Hayek, Friedrich A., *New Studies in Philosophy, Politics, Economics and the History of Ideas* (University of Chicago Press, Chicago, 1978) p.71.

[11] Drexler, K. Eric, "Molecular Engineering: An Approach to the Development of General Capabilities for Molecular Manipulation", in: *Proceedings of the National Academy of Science USA* (Sept. 1981) Vol. 78, No.9, pp.5275–5278.

[12] Drexler, K. Eric, "Rod Logic and Thermal Noise in the Molecular Nanocomputer", in: *Proceedings of the Third International Symposium on Molecular Electronic Devices* (Elsevier Science Publishers / North Holland, 1987).

[13] Drexler, K. Eric, *Engines of Creation* (Anchor Press / Doubleday, Garden City, New York, 1986).

[14] Coase, R. H., "The Nature of the Firm", in: *Economica, New Series* (1937), Vol. IV, pp.386–405; reprinted in: Stigler, G. J., and Boulding, K. E., (eds.), *Readings in Price Theory* (Richard D. Irwin, Inc., Chicago, 1952).

[15] Williamson, Oliver, *Markets and Hierarchies: Analysis and Anti-Trust Implications* (Free Press, New York, 1975).

[16] Malone, Thomas W.; Yates, JoAnne; and Benjamin, Robert I., "Electronic Markets and Electronic Hierarchies", in: *Communications of the ACM* (June 1987) Vol.30, No. 6, pp.484–497.

[17] Smith, Vernon L., "Experimental Methods in the Political Economy of Exchange", in: *Science* (10 October 1986) Vol.234, pp.167–173.

[18] Star, Spencer, "TRADER: A Knowledge-Based System for Trading in Markets", in: *Economics and Artificial Intelligence First International Conference* (Aix-En-Provence, France, September 1986).

[19] Hoare, C.A.R., *Communicating Sequential Processes* (Prentice-Hall, New York, 1985).

[20] INMOS Limited, *Occam Programming Manual* (Prentice-Hall International, London, 1984).

[21] Shapiro, Ehud, (ed.), *Concurrent Prolog: Collected Papers* (MIT Press, Cambridge, MA, 1987) in press.

[22] Artsy, Yeshayahu, and Livny, Miron, *An Approach to the Design of Fully Open Computing Systems* (University of Wisconsin / Madison, March 1987) Computer Sciences Technical Report #689.

[23] Nelson, Theodor, *Literary Machines* (published by author, version 87.1, 1987, available from Project Xanadu, 8480 Fredricksburg #8, San Antonio, TX 78229. Available as hypertext on disk from Owl International, 14218 NE 21st St., Bellevue, WA 98007. 1981).

[24] Granovetter, Mark, "The Strength of Weak Ties", in: *American Journal of Sociology* (1977) Vol. 78, pp.1360–1380.

[25] Miller, Mark S., Bobrow, Daniel G., Tribble, Eric Dean, and Levy, Jacob, "Logical Secrets", in: Shapiro, Ehud, (ed.), *Concurrent Prolog: Collected Papers* (MIT Press, Cambridge, MA, 1987) in press.

[26] Hardin, Garrett, "The Tragedy of the Commons", in: *Science* (13 December 1968) Vol. 162, pp.1243–1248.

[27] Kurose, James F., Schwartz, Mischa, and Yemini, Yechiam, "A Microeconomic Approach to Decentralized Optimization of Channel Access Policies in Multiaccess Networks", in: *Proceedings of the Fifth International Conference on Distributed Computing Systems* (IEEE, Denver CO, May 1985) pp.70–77.

[28] Goldberg, Adele, and Robson, Dave, *Smalltalk-80: The Language and its Implementation* (Addison-Wesley, Reading MA, 1983).

[29] Rivest, R., Shamir, A., and Adelman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", in: *Communications of the ACM* (Feb. 1978) Vol. 21, No. 2, pp.120–126.

[30] Tanenbaum, Andrew S., and van Renesse, Robbert, "Distributed Operating Systems", in: *ACM Computing Surveys* (ACM, New York, December 1985) Vol. 17, No. 4, pp.419–470.

[31] Hayek, Friedrich A., *Denationalisation of Money* (The Institute of Economic Affairs, Westminster, London, 1978, Second Edition).

[32] Denning, Peter J., "The Working Set Model for Program Behavior", in: *Communications of the ACM* (May 1968) Vol 2, No. 5, pp.323–333.

[33] Artsy, Y., Chang, H-Y, and Finkel, R.,

Processes Migrate in Charlotte (University of Wisconsin / Madison, August 1986) Computer Sciences Technical Report #655.

[34] Artsy, Y., and Finkel, R., "Simplicity, Efficiency, and Functionality in Designing a Process Migration Facility", in: *Proceedings of the Second Israel Conference on Computer Systems and Software Engineering* (IEEE, Tel-Aviv, Israel, May 1987) 3.1.2, pp.1–12.

[35] Cheriton, D.R., "The V Kernel: A Software Base for Distributed Systems", in: *IEEE Software* (April 1984) 1, 2, pp.19–42.

[36] Leach, P.J., Levine, P.H., Douros, B.P., Hamilton, J. A., Nelson, D.L., and Stumph, B.L., "The Architecture of an Integrated Local Network", in: *IEEE Journal on Selected Areas in Communication* (IEEE, November 1983) SAC-1, 5, 842–857.

[37] Bishop, Peter B., *Computers with a Large Address Space and Garbage Collection* (MIT, Cambridge, MA, May 1977) MIT/LCS/TR-178.

[38] Birrell, Andrew D.; Levin, Roy; Needham, Roger M.; and Schroeder, Michael D., "Grapevine: an Exercise in Distributed Computing", in: *Communications of the ACM* (April 1982) Vol. 25, No. 4.

[39] Terry, Douglas Brian, *Distributed Name Servers: Naming and Caching in Large Distributed Environments* (Xerox PARC, February 1985) CSL-85-1.

[40] Barak, A., and Shiloh, A., "A Distributed Load-Balancing Policy for a Multicomputer" in: *Software Practice and Experience* (September 1985) 15, pp.901–913.

[41] Shapiro, Ehud, "Systolic Programming: A Paradigm for Parallel Processing", in: *Proceedings of the International Conference on Fifth Generation Computer Systems* (1984) pp.458–471.

[42] Kahn, Kenneth, *A Partial Evaluator of Lisp Written in a Prolog Written in Lisp Intended to be Applied to the Prolog and Itself which in turn is Intended to be Given to Itself Together with the Prolog to Produce a Prolog Compiler* (University of Uppsala, Sweden, 1983) UPMAIL Tech. Report No. 17.

[43] Theriault, D., *Issues in the Design and Implementation of Act 2* (MIT AI Lab, Cambridge, MA., 1983) AI-TR-728.

[44] Winograd, Terry, and Flores, Fernando, *Understanding Computers and Cognition* (Ablex, Norwood, NJ, 1986).

[45] Witham, Steve, *personal communication* (1987).

[46] Safra, S., and Shapiro, Ehud, "Meta-Interpreters For Real", in: *Proceedings, IFIP-86* (1986) pp.271–278.

[47] Stamos, James W., *A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance* (Xerox PARC, Palo Alto, CA, May 1982) SCG-82-2.

[48] Stanley, Terry, *personal communication* (1987).

[49] Hamming, R. W., "One Man's View of Computer Science", in: Ashenhurst, Robert L., and Graham, Susan, (eds.), *ACM Turing Award Lectures: The First Twenty Years 1966–1985* (Addison-Wesley, Reading, MA, 1987) pp.216.

[50] Cox, Brad J., *Object Oriented Programming: An Evolutionary Approach* (Addison-Wesley, Reading, MA, 1986) pp.26–28.

[51] Jacobson, Gary, and Hillkirk, John, *Xerox: American Samurai* (Macmillan, New York, 1986).

[52] Chaum, David, "Design Concepts for Tamper Responding Systems", in: *Advances in Cryptology: Proceedings of Crypto '83* (Plenum Press, NY, 1984) pp.387–392.

[53] Levy, Henry M., *Capability-Based Computer Systems* (Digital Press, Bedford, MA, 1984).

[54] Gehringer, Edward F., *Capability Architectures and Small Objects* (UMI Research Press, Ann Arbor, MI, 1982).

[55] Organick, Elliott I., *A Programmer's View of the Intel 432 System* (McGraw-Hill, New York, 1983).

[56] Rees, Jonathan A., and Adams, Norman I., IV, "T: a Dialect of Lisp or, Lambda: The Ultimate Software Tool", in: *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming* (August 1982).

[57] Conway, M.E., "How Do Committees Invent?", in: *Datamation* (April 1968) 14, 4, pp.28–31.

[58] Brooks, Frederick P., Jr., *The Mythical Man Month* (Addison-Wesley Publishing Company, Reading, MA, 1975) pp.111.

[59] Hayek, Friedrich A., "Cosmos and Taxis" in: *Law, Legislation and Liberty, Vol. 1: Rules and Order*, (University of Chicago Press, Chicago, 1973) pp.35–54.

[60] Minsky, Marvin, *The Society of Mind*

(Simon and Schuster, New York, 1986).

[61] Kornfeld, William A., and Hewitt, Carl, "The Scientific Community Metaphor", in: *IEEE Transactions on Systems, Man, and Cybernetics* (IEEE, 1981) SMC-11, pp.24–33.

[63] March, J. G., "Footnotes to Organizational Change", in: *Administrative Science Quarterly* (1981) 26, pp.563–577.

[64] Barstow, David R., Shrobe, Howard E., and Sandewall, Erik, (eds.), *Interactive Programming Environments* (McGraw-Hill, New York, 1984).

[65] Xerox, *Courier: The Remote Procedure Call Protocol* (Xerox Corp, Stamford CT, 1982) p.5.

[66] Rao, Ramana Balusu, *Toward Interoperability and Extensibility in Window Environments via Object-Oriented Programming* (MIT Press, 1987) submitted as Masters Thesis.

[67] Shrager, Jeff, and Klahr, David, "Instructionless Learning about a Complex Device: The Paradigm and Observations", in: *Int. J. Man-Machine Studies* (1986) 25, pp.153–189.

[68] Dawkins, Richard, *The Selfish Gene* (Oxford University Press, New York, 1976).

[69] Axelrod, Robert, *The Evolution of Cooperation* (Basic Books, New York, 1984).

[70] McDermott, Drew, "A Critique of Pure Reason", to appear in: Levesque, Hector, (ed.), *Computational Intelligence* (National Research Council of Canada, August or September, 1987).

[71] Dijkstra, E. W., "Co-operating Sequential Processes", in: Genuys, F., (ed.), *Programming Languages* (Academic Press, New York, 1968) pp.43–112.

[72] Demers, Alan, Greene, Dan, Hauser, Carl, Irish, Wes, Larson, John, Shenker, Scott, Sturgis, Howard, Swinehart, Dan, and Terry, Doug, "Epidemic Algorithms for Replicated Database Maintenance", in: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (ACM, Vancouver, British Columbia, Canada, August 10–12, 1987) pp.1–12.

[73] Hanson, Robin, *Toward Hypertext Publishing: Issues and Choices in Database Design* (draft available from Foresight Institute, Palo Alto, CA, 1987).

[74] Stefik, Mark, Foster, Gregg, Bobrow, Daniel G., Lahn, Kenneth, Lanning, Stan, and Suchman, Lucy, "Beyond the Chalkboard: Computer Support for Colaboration and Problem Solving in Meetings", in: *Communications of the ACM* (January 1987) Vol. 30, No. 1, pp.32–47.

[75] Hewitt, Carl, "Robert's Rules of Order" (in press).

[76] Malone, Thomas W., "Organizing Information Processing Systems: Parallels Between Human Organizations and Computer Systems", in: Zacharay, W., Robertson, S., and Black, J., (eds.), *Cognition, Computation, and Cooperation* (Ablex Publishing Corp., Norwood, NJ, 1986).

[77] Lenat, Douglas B., "The Role of Heuristics in Learning by Discovery: Three Case Studies", in: Michalski, Ryszard S., Carbonell, Jaime G., and Mitchell, Tom M. (eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing Company, Palo Alto, CA, 1983) pp.243–306.

[78] Harris, Jed, Yu, Chee, Harris, Britton, *Market Based Scheduling* (1987) in preparation.

[79] Sutherland, I.E., "A Futures Market in Computer Time", in: *Communications of the ACM* (June 1968) Volume 11, Number 6.

[80] McClelland, James L., Rumelhart, David E., and PDP Research Group, *Parallel Distributed Processing* (MIT Press, Cambridge, MA, 1986) Volumes 1 and 2.

[81] Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R., *Induction: Processes of Inference, Learning, and Discovery* (MIT Press, Cambridge, MA, 1986).

[82] Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R., *Induction: Processes of Inference, Learning, and Discovery* (MIT Press, Cambridge, MA, 1986) pp.72–75, 79.

[83] Barto, Andrew G., "Game Theoretic Cooperativity in Networks of Self-Interested Units", in: Denker, John S. (ed.), *Neural Networks for Computing* (American Institute of Physics, New York, 1986) pp.41–46.

[84] Minsky, Marvin, "Steps Toward Artificial Intelligence", in: Feigenbaum, Edward A., and Feldman, Julian, (eds.), *Computers and Thought* (Robert E. Krieger, Malabar, FL, 1981) pp.406–450.

[85] Minsky, Marvin, *personal communication* (1987).