Interactive Software Technology

Peter Wegner, Brown University

CRC Handbook of Computer Science and Engineering, May 1996

Contents:

Abstract:

The evolution from mainframe to personal computer and network technology is characterized by a paradigm shift from algorithms to interactive models of computation. The first three sections develop a conceptual framework for interactive computation, showing that models of interaction have observably richer behavior than Turing machines that more completely express the behavior of embedded systems like airline reservation and banking systems. Sections 4 through 9 examine specific software architectures for programming in the large, including object-based design, multiple interface models, and technologies for interoperability, design patterns, coordination, and agent-oriented programming. Sections 10 and 11 present case studies for virtual reality and data information systems, demonstrating the naturalness of interactive models as a basis for computer graphics applications and more generally for empirical computer science.

Interactive Software Technology
Peter Wegner, Draft Chapter for CRC Handbook of CS&E, May 1996

1. Evolution of Computing Paradigms

The problem of driving from Providence to Boston is usually solved by combining *algorithmic* knowledge (a high-level mental map) with *interactive* visual feedback of road signs and road topography. In principle, interactive feedback could be replaced by a noninterctive algorithmic specification so that (assuming no other traffic) a blindfolded person could drive entirely by rules for accelerating, breaking, and turning the steering wheel in an entirely algorithmic mode. However, the complexity of such an algorithmic specification is enormous, and would in any case not handle traffic and other interactively variable factors.

Algorithmic computation is very weak in modeling interactive behavior like driving: it corresponds to blindfolded or autistic behavior in humans. Interactive descriptions are both simpler and more natural than entirely algorithmic specifications even when algorithmic specifications exist, and can express inherently interactive behavior not expressible by any algorithmic specification. Though algorithms are surprisingly versatile, interaction provides an extra dimension that reduces the complextiy of problem solving and expands the power of models of computation so they can express and solve a larger class of problems.

Computing has evolved from machine language programming through procedure-oriented and structured programming to object-based and component-based programming:

1950s: Machine language programming: assemblers, hardware-defined action sequences
1960s: Procedure-oriented programming: compilers, programmer-defined action sequences
1970s: Structured programming: software engineering, algorithm architecture
1980s: Object-based programming: personal computers, sequential interaction architecture
1990s: Component-based software technology: concurrent interaction architecture, coordination

Whereas the transition from machine to procedure-oriented programming simply involves a change in the granularity of actions, the shift from procedure-oriented to object-based systems is more fundamental, involving a shift from algorithmic to interactive computing [We1]. The shift from sequentially interacting objects to concurrent interactions of software components is a further fundamental paradigm shift:
machine language -> procedure-oriented language
quantitative change of scale in the granularity of actions
procedure-oriented -> object-based
qualitative change of expressiveness from algorithms to interactive systems
object based -> component-based
qualitative change of expressiveness from sequential to concurrent (nonserializable) interaction

Models of interaction provide a unifying framework for design and analysis of software engineering, artificial intelligence, and database applications. Objects have richer observable behavior than procedures, while components with multiple concurrent input streams in turn have richer observable behavior than sequential objects.

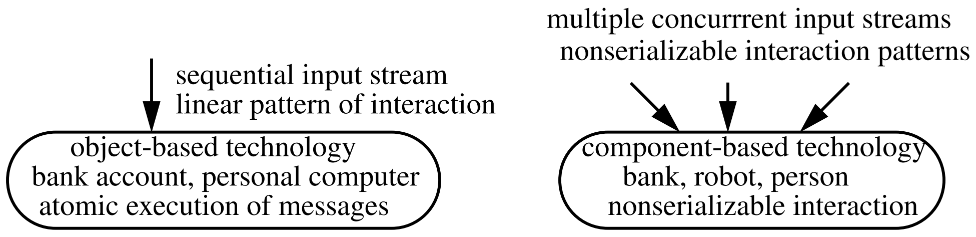

sequential input stream
linear pattern of interaction
object-based technology
bank account, personal computer
atomic execution of messages
multiple concurrrent input streams
nonserializable interaction patterns
component-based technology
bank, robot, person
nonserializable interaction

Figure 1: Interactive Inputs for Objects and Components

Models of interaction provide a unifying conceptual framework for the description and analysis of object and component-based architectures. They are a tool for exploring software design models like OMT, multiple interface models like COM/OLE, models of interoperability like CORBA, design patterns, coordination languages like Linda and Gamma, and AI models of planning and control. A case study of interaction architecture for virtual reality is presented, and a final section considers an interaction architecture for a broad class of interactive systems that includes data acquisition systems like NASA's earth observation system (EOS) and digital library systems.

Objects and algorithms both determine a *contract* between providers and clients of a resource, but objects provide fundamentally richer services to clients that cannot be expressed by algorithms. Algorithms are like sales contracts, guaranteeing an output for every input, while objects are like marriage contracts, describing ongoing contracts for services over time. An object's contract with its clients specifies its behavior for all contingencies of interaction (in sickness and in health) over the lifetime of the object (till death us do part) [We2]. The folk wisdom that marriage contracts cannot be reduced to sales contracts is computationally expressed by interaction not being reducible to algorithms.

Object-based programming has become a dominant technology, but its foundations are shaky: everyone talks about it but no one knows what it is. "Knowing what it is" has proved elusive because of the implicit belief that "what it is" must be defined in terms of algorithms. Irreducibility has the liberating effect of allowing "what it is" to be defined in terms of interactive models rather than algorithms. Component-based software technology is even less mature than object-based technology: it is the technology underlying interoperability, coordination models, pattern theory, and the World-Wide Web. Knowing what it is in turn requires liberation from sequential object-based models.

At a more fundamental level the distinction between algorithms and interaction corresponds to that between closed and open systems and to that between rationalism and empiricism. Interactive models of computation provide a precise characterization of open systems and empirical computer science [We4]. Irreducibility of interactive to algorithmic models implies that empirical computer science is fundamentally richer than models of algorithms and complexity theory, but the cost of greater richness is nonformalizability and incompleteness (in the sense of Godel.

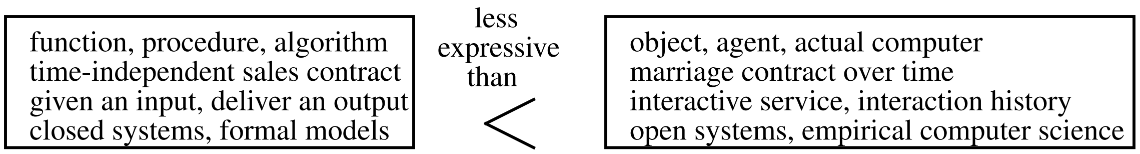
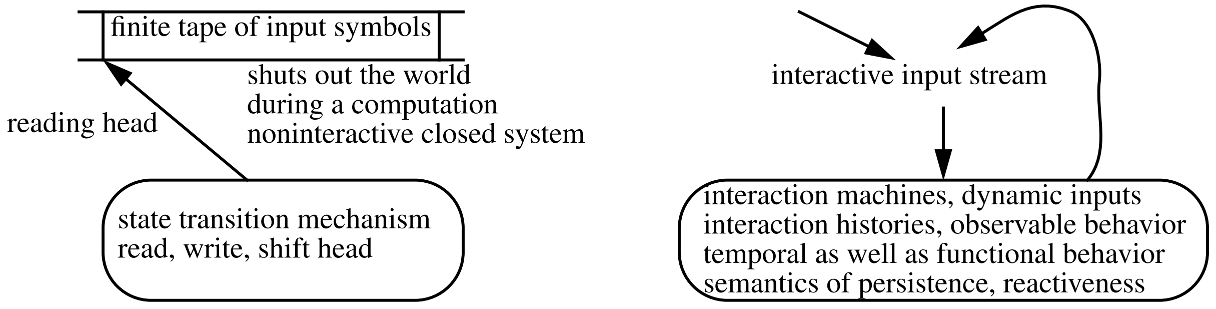

function, procedure, algorithm
time-independent sales contract
given an input, deliver an output
closed systems, formal models
less expressive than
<
object, agent, actual computer
marriage contract over time
interactive service, interaction history
open systems, empirical computer science

Figure 2: Sales Versus Marriage Contracts

2. Models of Interaction

To provide a formal framework for interactive models we extend Turing machines, which model algorithmic computation, to interaction machines, which model objects and software components. Turing machines have a tape that initially contains a finite sequence of input symbols and a state transition mechanism that can read a symbol from the tape, perform a state transition, write a symbol on the tape, and reposition the reading head. Interaction machines extend the Turing machine model by adding input and output actions (read and write statements). Whereas Turing machines require all inputs to appear on the tape prior to the computation and shut out the world during the process of computation, interaction machines allow inputs to be dynamically generated and require inputs to be represented by a potentially infinite stream, since any finite stream can be dynamically extended (see figure 3).


finite tape of input symbols
shuts out the world
during a computation
noninteractive closed system
reading head
state transition mechanism
read, write, shift head
interactive input stream
interaction machines, dynamic inputs
interaction histories, observable behavior
temporal as well as functional behavior
semantics of persistence, reactiveness

Figur 3: Turing and Interaction Machines

Interaction machines are intuitively more expressive than Turing machines because they model the passage of time during the process of problem solving, while algorithms and Turing machines model only time-independent transformations. This informal difference is formally captured by the difference between finite input tapes and unbounded input streams. Interaction machines correspond to Turing machines with infinite tapes, which are known to be more powerful than Turing machines. Whereas the inputs of a Turing machines are countable, the number of potential streams of an interaction machines is nonenumerable.

Interactive adversaries can always extend interactive streams by adding an input. This informal difference between noninteractive and interactive computations is modeled formally by infinite streams as opposed to finite tapes. Though interactive computations are finite in practice, interaction machines and computations with an unbounded number of interaction steps cannot be modeled by the set of all finite sequences (which is enumerable) but must include infinite limit points, because adversaries have the last word and can always extend any finite sequence. The power of adversaries may be used to show how noncomputable functions postulated by the diagonalization argument can be computed by generating sequences that differ from any enumerable sequence in their diagonal position. Environments that generate external input steams not under the control of the interaction machine are modeled mathematically by infinite sets of sequences that include limit points and have a nonenumerable cardinality.

Nonenumerability versus countability expresses the essence of the difference between Turing and interaction machines. “Real” numbers and “real” time are dual nonenumerable abstractions of reality. Real numbers were viewed in the 19th century as models of the infinite divisibility of continuous mathematical and physical space. Interaction machines turn this model inside-out, representing time in the “real” world by infinite input streams. The set of all infinite digit streams is in one-to-one correspondence with both the nonenumerable real numbers and the input streams of an interaction machine.

Though nonenumerability provides a mathematical mechanism for proving the greater power of interactive computing it does not completely capture the semantics of interaction. Interactive processes are nonalgorithmic even if they have finite time horizons because their inputs are uncontrollable and nondeterministic. The fact that the next step of an interactive process is externally determined causes even finite interactive processes to be nonalgorithmic: interactive processes have lower complexity because they can simply access external data (driving home from work is a simple interactive process but a horrendously complex algorithmic process). The dynamic acquisition of interactive knowledge is a key to

a) informally better performance of practical interactive tasks

b) formally lower complexity of finite tasks in reducing exponential search to polynomial time

c) formally greater expressiveness in modeling tasks with an unbounded number of interactive steps.

Greater expressiveness is the limiting case of lower complexity for finite tasks: nonenumerability is the limiting case of uniform reduction of exponential to polynomial time complexity.

Interaction machines are incomplete in the sense of Godel: their nonenumerable number of true statements cannot be enumerated by a set of theorems. Incompleteness has strong practical consequences: it implies that projects like the fifth-generation computing project that aim to reduce computation to logic fail because their goals cannot in principle be realized. Even a hundred-fold increase in effort and a ten-year extension would not have allowed the fifth-generation project to succeed. The view that logic programming is theoretically too weak to model interactive systems, presented by the author at the closing session of the fifth-generation computing project in Tokyo in 1992 [We3], has practical technological implications.

Interaction machines provide a precise characterization of empirical computer science while irreducibility and incompleteness indicate that empirical computer science differs fundamentally from algorithms and relate the “real” world to the “real” numbers. Turing machines have been the dominant computational abstraction for the first 50 years of computer science, but their role in the next 50 years may be less central because they are not powerful enough to capture the behavior over time of objects, software systems, and distributed computation.

Church’s thesis that the intuitive notion of computing corresponds to formal computing by Turing machines is seen to be invalid or at least inapplicable, since interaction machines capture persistent, time-dependent behavior of actual computers and software applications more accurately than Turing machines. Interaction machines precisely capture the “empirical computer science” paradigm: interaction expresses observability and models in the natural sciences. The Chomsky hierarchy of machines can be extended beyond Turing machines to synchronous, asynchronous, and nonserializable interaction machines whose expressiveness corresponds respectively to Newtonian, relativistic, and chaos models of physics [We2].

3. Software Engineering, Artificial Intelligence, and Open Systems

Interactive models provide a common conceptual framework for software engineering and artificial intelligence. The evolution in AI from logic and search to agent-oriented models is not merely a tactical change but is a strategic paradigm shift from algorithms to more expressive interactive models that fundamentally increases expressive power. The reasoning/interaction dichotomy is precisely that between good old-fashioned AI (GOFAI) and “modern” agent-oriented AI. This paradigm shift is evident not only in research [AI] but also in textbooks that systematically reformulate AI in terms of intelligent agents [RN].

Programming in the small (PIS) is algorithmic, while programming in the large (PIL) deals not with large programs but with interactive systems. An algorithmic program with a million arithmetic operations is not PIL, while medium-size embedded software systems are. PIL is not simply scaled-up PIS; it has qualitatively different program structures and models of computation. The irreducibility of interaction to algorithms implies inexpressibility of PIL by PIS. Scaling up shifts attention from inner activities within components to interaction among components. PIL was observed to differ from PIS as early as the 1960s, but the difference was viewed as a quantitative change of scale. Expressing the difference as a qualitative change of expressiveness explains the observed inability to scale up from algorithms to software systems.

The irreducibility of interactive systems to algorithms confirms Fred Brooks’ persuasive argument [Br1] that there is no silver bullet for simply specifying complex systems. If silver bullets are interpreted as algorithmic or formal specifications, the nonexistence of silver bullets can actually be proved. Systems that persist in time have an interactive essence that cannot be expressed by silver bullets fashioned from algorithmic or formal models. This negative “impossibility result” has a positive liberating effect on explanatory models. Giving up the goal of complete behavior specification requires a psychological adjustment, but makes partial system specification respectable. Though a complete elephant cannot be specified, its parts and its forms of behavior (its trunk or mode of eating peanuts) are specifiable. Complete specification must be replaced by the more modest goal of partial specification by interfaces, views, and modes of use.

The idea that physical objects are not completely describable or knowable but that they may have describable parts or views is a basic tenet of the scientific method. Plato’s cave metaphor asserts that we are like people in a cave seeing only shadows of the real world (incomplete knowledge) on the walls of our cave but not reality itself. Though Plato’s denial of the validity of complete knowledge is correct, his inference that partial empirical knowledge is worthless and that only mathematical knowledge is “real” is erroneous. Science is based on acceptance that useful models of the real world can be constructed from shadows on the walls of our cave (images on our retina). Interactive objects can be perceived only by partial interface behaviors, just as real-world objects can be perceived only by observed behaviors.

Software architecture is yet another area whose elusive nature can be explained by the irreducibility of interaction to algorithms. Its definition as "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. [Ga]" indicates a clear focus on interactive interrelations among components as opposed to algorithmic control structures. The analysis of specific architectures such as client/server structures, UNIX pipes, blackboards, and distributed network topologies makes use of algorithmic tools in defining protocols, but software architecture is an art rather than a formal discipline and the complete set of interactive system behaviors determined by software architectures cannot be algorithmically specified. The study of component-based architectures does not fit into the framework of algorithm-based models of computation.

A computing system is said to be open if its computations depends on external information and closed otherwise. Turing machines with initial tapes and algorithms with initial inputs are *closed*: their actions do not depend on external interaction. In contrast, interaction machines are *open*: their actions are influenced by the external world. The distinction between closed and open systems is precisely that between algorithmic and interactive computing. Algorithms are closed because they shut out the world during computation, while interactive computing is open in allowing external agents to observe and influence computation.

The complete behavior of servers (or agents) is complex and necessarily open since it must handle all possible clients, while the individual interface demands of clients are often (though not necessarily) simple and closed. Supplied behavior is generally much richer than demanded behavior (see Figure 4).

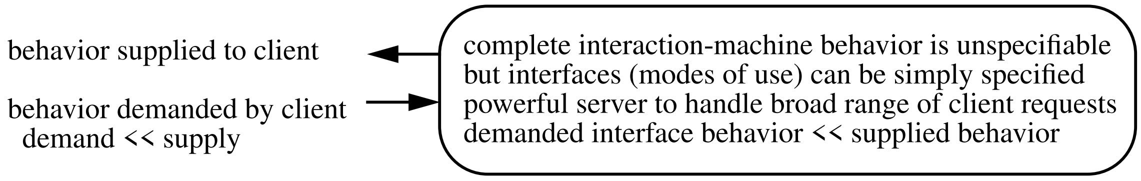

behavior supplied to client
behavior demanded by client
demand << supply
complete interaction-machine behavior is unspecifiable
but interfaces (modes of use) can be simply specified
powerful server to handle broad range of client requests
demanded interface behavior << supplied behavior

Figure 4: Supplied Server Behavior Versus Demanded Client Behavior

Useful categories of interactive systems can be classified by the kinds of constraints we impose upon them to make them tractable. Perhaps the most common constraint is that of precluding interaction during a computation, which causes the system to become closed. For example, Turing machines and algorithms are closed systems because their rules of engagement require all inputs to be supplied on an input tape before the beginning of the computation. Any open system can be closed by constraining its rules of engagement to be independent of external effects. Interaction machines require their rules of engagement to permit inputs during the process of computation and are therefore inherently open.

The state transition mechanism of a Turing machine, considered as an isolated system, is open since inputs occur during execution. Turing machines insulate their state transition mechanism from dynamic interaction by providing a finite tape prior to the start of the computation. Figure 5 illustrates an alternative way of closing open systems by an environment that is itself an object. The two objects O1, O2 are open as isolated systems but become a closed system when composed so that each object talks only to the other. Each object O1, O2 interacts as an open system with an arbitrary collection of clients, while the composite system causes each object to constrain the behavior of the other so it interacts with only a specific client.

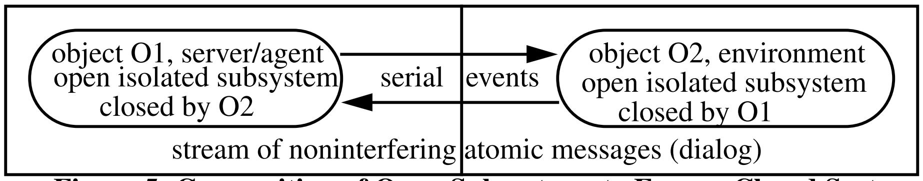


Figure 5: Composition of Open Subsystems to Form a Closed System

The set of all possible behaviors of the object O1, viewed as an isolated system, is nonalgorithmic because its input streams can be infinite sequences not generable by any algorithm (not recursively enumerable). When input sequences of O1 are constrained to be those produced by an object O2 in response to a message stream from O1, the behavior of O1 can be tractably described. The object O2 plays the role of a tape, albeit an intelligent tape, in constraining the interaction machine O1 to closed-system behavior.

Nonalgorithmicity is not preserved under system composition, and conversely algorithmicity is not preserved under system decomposition.

Though the relation between O1 and O2 appears symmetrical from outside the system, it becomes asymmetrical for observers residing in one of the objects or when the two objects play different roles. For example, O1 may be a server and O2 a client, or O1 an agent and O2 an environment that O1 is exploring or a client on whose behalf the agent is acting. Two-component systems where each acts as a constraint on the other arise in control theory: one component is the system being controlled while the other is a controller that may exercise strict control (as in a prison) or loose control (as in a free society). In this context the idea that an isolated system has richer uncontrolled than controlled behavior is very natural.

4. Object-Oriented Design: Sequential Interaction

Object models have many "small" objects with sequential interface and interaction protocols, while applications like airline reservation systems have "heavy" components with multiple interfaces and concurrent interaction protocols. In this section we examine the primitive entities and observable dynamic behavior of object models. The gap between static structure and dynamic behavior is greater for objects than for algorithms, with two distinct levels of execution dynamics: the external dynamics of operation execution is entirely separate from that of inner rule-based algorithm execution. The object modeling technique OMT [Ru] has an *object model* for describing static object structure, a *dynamic model* that describes interaction histories, and a *functional model* that describes transformation behavior of operations:

object model: *describes relations among interactive components (nouns)*
static description of objects, operations, and relations among objects by object diagrams
dynamic model: *describes interaction histories compatible with the object model (inter-object dynamics)*
dynamic inter-object interaction histories (event sequences) touching multiple objects
functional model: *describes behavior of specific functions (intra-object dynamics)*
intra-object transformation behavior at the level of algorithms

These three levels provide a robust modeling framework for a variety of models of object-oriented design, as pointed out by Jacobson [Ja]. They reflect the fact that nouns provide a more direct and more expressive model of the real world than verbs and the further fact that nouns are modeled computationally by the patterns of actions (verbs) that they support. Nouns whose behavior is expressed by patterns of observable actions are naturally modeled by an object model expressing relations among nouns, a dynamic model expressing patterns of interaction, and a functional model specifying individual actions. Externally determined patterns of interaction are not constrained by requirements of algorithm specification.

Dynamic models assume that interaction histories are traces (sequences) of time-independent events. They constrain interactions to a disciplined sequential form but exclude time-dependent and nonserializable interaction histories. Interaction histories can be viewed as test cases analogous to instruction execution histories of algorithm computations. Just as no amount of testing can prove correctness of algorithms, interaction histories can only show the existence of desirable behaviors and cannot prove correctness. The incompleteness of interactive systems implies that proving correctness is not merely hard but impossible. We must be satisfied showing the existence of desirable behaviors through test cases (sequential interaction histories in the case of OMT) and cannot hope to prove the nonexistence of incorrect behaviors [We4].

Though there is a superficial resemblance among object, dynamic, and functional models and corresponding levels of modeling for algorithms (see Figure 6), object models are very different from flow diagrams and object interaction histories are very different from algorithm execution histories. Flow diagrams have actions as nodes and control paths as edges, while object diagrams have objects as nodes and access paths as edges. Algorithm execution histories specify time-independent instruction sequences, while system interaction histories specify time-dependent events in real or artificial worlds. Though instruction and operation sequences are both linearly ordered, they have entirely different observable behavior. Instruction sequences define transformations specifiable by computable functions, while operation sequences define processes in time with a richer set of observationally distinct behaviors as described in section 2.

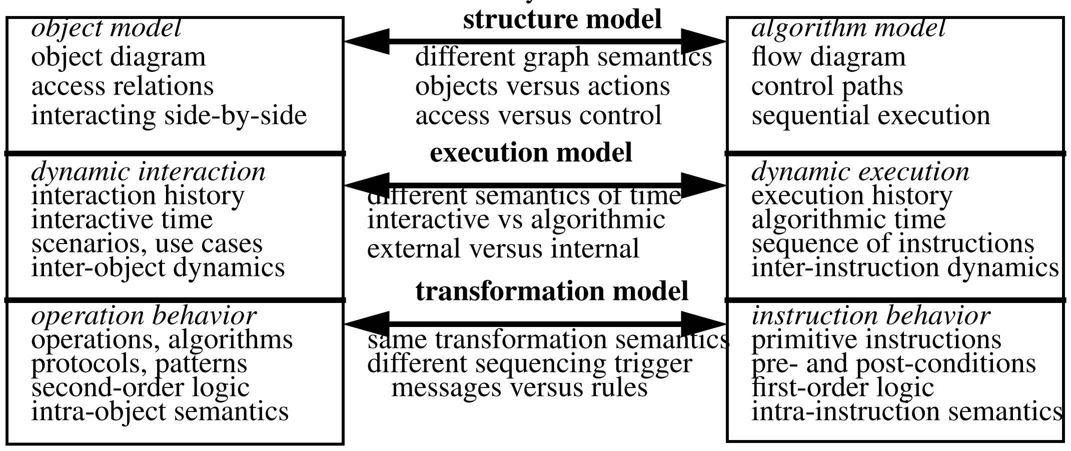


Figure 6: Three-Level Model for Objects and Procedures

Three-level object design models clearly indicate the role of algorithms as low-level transformation specifications of primitive elements of interaction patterns. Object models focus primarily on patterns of interaction: the functionality of primitive elements is a secondary though still important concern. Restricted sequential patterns of dynamic models are often specifiable by algorithms. But unbounded sequential streams generated by external processes need not be recursively enumerable or deterministic.

Because general patterns of interaction are undescribable, the literature on design patterns [GHJV] abandons algorithmic specification in favor of informal verbal description of problem and solution structure to specify reusable patterns of object and component interaction. Design patterns are behaviorally simply interaction patterns, but interaction patterns are too low level to capture user-level regularities (they are the machine language of design patterns). Higher level patterns are therefore used to describe design patterns, such as subproblem descriptions, trade-offs among design features, and processes of design.

5. Multiple-Interface Model: Concurrent Interaction

Objects have fixed sequentially accessible interfaces while components of real-world application systems have multiple interfaces that can be dynamically added and removed and concurrently executed. Such components have richer observable behavior than objects with atomic sequentially executable operations. We first consider multiple independent interfaces that increase the accessible resources and/or the views of a component without introducing concurrency and can be modeled by extending sequential object-based systems, and then explore components with multiple concurrently accessible interdependent interfaces that require fundamentally new models. Microsoft's Component

Microsoft's Component Object Model [COM] is an architecture for multiple independent interfaces. It extends the client-server model so that components can evolve by dynamically adding new interfaces. Components are specified by interface directories rather than sets of operations of a single interface. Every component must possess an interface I-unknown for managing its interface directory, with operations for adding and deleting interfaces and a "queryinterface" operation that checks the interface directory for the existence of interfaces (see Figure 7).

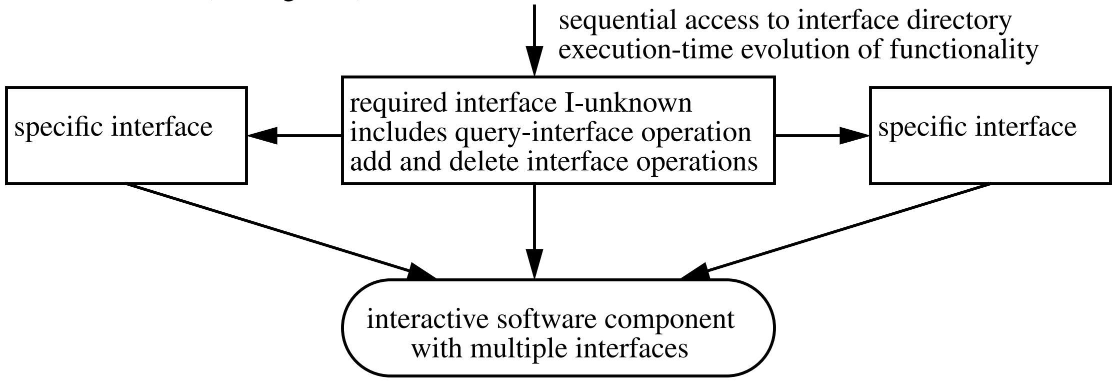

sequential access to interface directory
execution-time evolution of functionality
specific interface
required interface I-unknown
includes query-interface operation
add and delete interface operations
specific interface
interactive software component
with multiple interfaces

Figure 7: Multiple Interface Model for COM/OLE Software Components

COM allows execution-time actions to depend on the existence of specific interfaces and permits the dynamic addition of new interfaces to components of deployed systems. This form of reusability simplifies the maintenance of evolving systems whose components change their interface functionality, but assumes that new interface functionality does not interact with existing functionality through a shared state. Interactive applications like airline reservation and ATM systems whose multiple interfaces interact through a shared state cannot be constructed by adding interfaces to a COM interface directory.

COM's objects support multiple independent interfaces during object evolution but not multiple interdependent interfaces during execution. The time scale for life-cycle evolution has larger granularity than that for real-time execution, just as geological or anthropological time has larger granularity than quantum-theoretic time. Models of persistence must handle time at both the macro and the micro levels: there is a deep analogy between interference phenomena in physical models of quantum theory and chaos and computational interference due to nonserializability. Nonserializable interaction among interfaces requires qualitatively different models from COM's creation and deletion of multiple independent interfaces.

Airline reservation systems are prototypical of applications with multiple views sharing a common database. Multiple concurrent interactions with the database have unpredictable functionality, since the state seen by one interface can be unpredictably changed through another interface. Multiple interfaces are an interactive analog of multiple threads: concurrent interactive access parallels concurrent execution. Airline reservation systems have many interfaces that interact through shared data structures:

Interfaces (modes of use) of airline reservation systems:
travel agents: making reservations on behalf of clients
passengers: making direct reservations
airline desk employees: making inquiries on behalf of clients
flight attendants: aiding passengers during the flight itself
accountants: auditing and checking financial transactions
systems builders: developing and modifying the system

Travel agent interfaces permit a client to reserve a seat under normal conditions, but this response may be subverted by interaction through other interfaces. The response to a given travel agent request depends on other travel agents, direct reservations by travelers, flight cancellations that cause wholesale transfer from one flight to another, and exogenous events like snowstorms or hurricanes. The response to a message depends on nonlocal events in the external (real) world: the computational analog of action at a distance.

Interdependent interfaces that share a state are a fundamental cause of unpredictable behavior in both physical and computational systems. Unpredictability arises even when access to the shared state is sequential because an interface cannot assume that the state remains undisturbed for successive accesses to the state. However, the problem becomes more acute when access is concurrent because the state seen by an interface may change unpredictably during execution of an operation. We examine this situation for joint bank accounts and show that unpredictability due to conflict among concurrent input streams is the discrete analog of chaos in continuous physical systems.

Suppose that a joint bank account contains \$1.5 million and that two clients simultaneously try to withdraw \$1 million at different ATMs. Assume that the withdrawal process requires adjusting an investment portfolio and is therefore not instantaneous (see Figure 8).

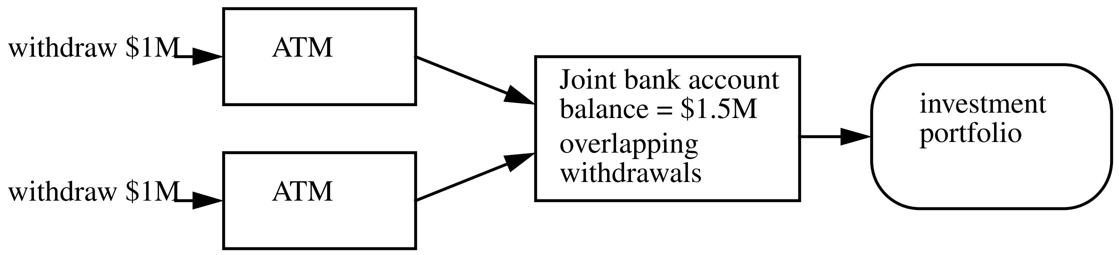


Figure 8: ATM System as an Asynchronous Interaction Machine

A transaction system could in principle handle this situation by satisfying only one client and aborting the transaction of the other. But concurrent interactive systems cannot be presumed to be transactionally well behaved: we must model and manage breakdown of transactional behavior, just as psychologists must model and manage nervous breakdown in people, and physical systems must cope with chaos. Transactions are a computational mechanism for handling system overload caused by multiple simultaneous demands on a resource. The effect of concurrent potentially conflicting operations op1, op2 of an object can, in the absence of transaction atomicity, be arbitrary and chaotic. Behavior becomes *nonserializable* in that it does not correspond to any sequential execution of the operations op1 and op2. Nonserializability of concurrently executed operations of an object's interface specializes nonserializability of database transactions by considering only atomicity of interface operations, but is essentially the same problem.

Though nonserializable behavior is considered undesirable in many contexts, it is more expressive (observably richer) than serializable behavior and can be harnessed for useful purposes. Aborting a transaction or replacing a time consuming optimal algorithm by an approximate just-in-time algorithm is a technique for managing nonserializable behavior. Aborted transactions and just-in-time algorithms replace ideally desired functionality by less desirable functionality that can be serializably realized.

Transaction management systems that guarantee atomicity are safe but often too conservative. Permitting some degree of controllable nonserializability, just as permitting some unsound behavior in tasks like error checking, can be worthwhile. For example, optimistic concurrency control systems lower their guard on the assumption that no conflicts occur and pay for this by requiring drastic and time consuming actions when this assumption is violated. High-achieving people, like efficient computers, operate close to the margins of nonserializability and pay the price of greater stress and a higher incidence of nervous breakdowns than person opting for comfort at the expense of achievement.

Noatomicity of operations causes unobservable and uncontrollable temporal sensitivity in accessing a shared resource (the object's state) that is analogous to chaos in physics: sensitivity among competing clients to shared computing resources corresponds to sensitivity among competing forces on a shared physical object. The pattern of competing access to shared resources in the bank account example arises in the well-known demonstration of physical chaotic behavior of Figure 9, which illustrates a two-dimensional pendulum (steel ball) in the presence of two magnets. The magnets M1, M2 correspond to the operations op1, op2, while the steel ball corresponds to the state. The magnets exert two streams of impulses on the steel ball just as operations exert streams of impulses on the object's state. Chaos arises when the steel ball passes through regions where the force fields are almost equal in which the motion of the steel ball is extremely sensitive to minute perturbations of the force field. This example illustrates the deep correspondence between chaos and nonserializability and more generally the fact that interaction machines can model phenomena that arise in physical systems.

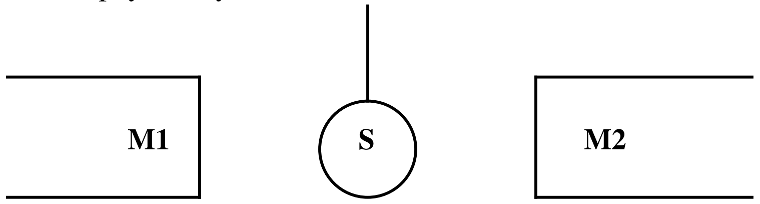


Figure 9: Chaotic Two-Dimensional Motion of Steel Pendulum in a Magnetic Field

Chaotic behavior in physics is modeled by nonlinear differential equations. The differential equations for a pendulum have a linear first-order behavior but nonlinear second-order terms when the first-order terms cancel each other. Thus pendulum behavior is linear when in the force field of one of the magnets but nonlinear in regions where the force fields cancel each other out. Interaction machines are a discrete analog of dynamical systems in physics. Their chaotic behavior is a discrete analog of turbulence in differential equations: both are a consequence of extreme sensitivity to initial conditions.

6. Interoperability: Mediated Interaction

Interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform. It is a scalable form of reusability, extending the reuse of server resources to clients whose accessing mechanisms may be plug-incompatible with sockets of the server. Plug compatibility arises most literally with electrical appliances that require both static compatibility of shape and dynamic compatibility of voltage and frequency. If there is no direct match, interoperability of electrical appliances can be achieved by adapters and transformers.

interoperability = scalable reusability = plug and socket compatibility

As with electrical appliances, incompatibility of software plugs and sockets can be mediated by adapters. Software adapters are part of the interaction architecture rather than the functionality of a software system. Interoperability architectures like the Common Object Request Broker Architecture (CORBA) are effectively elaborate adapters that provide both static and dynamic compatibility among heterogeneous components. To provide a framework for describing particular architectures like CORBA, we briefly discuss methods of interface specification and interface bridging.

Interfaces are traditionally specified by types. Static compatibility in strongly typed languages can be determined by type checking. Type compatibility guarantees that the plug of the client fits the socket of the server but not that the service provided is that desired by the caller. Minor type differences between data representations, like that between integers and floating-point numbers, can be handled by coercion. Stronger type differences can be handled by polymorphism, which allows an interface to accept inputs of a variety of types. Heterogeneous components have even stronger differences that cannot be handled by polymorphism. Some of the most powerful systems of interoperation, like UNIX pipes and http World-Wide-Web protocols, are typeless. Interface definition languages with static type-compatibility protocols promote safety and efficiency at a considerable cost in design and implementation complexity. Much of the research in this area is designed to realize the safety of typed systems with acceptable complexity.

Clients and servers from different software platforms or programming languages can talk to each other through *mediators (adapters)* that convert data formats. Mediation in information systems as an organizing principle for interoperation of heterogeneous components is reviewed in [Wi]. Though mediators can handle a wide range of differences in data formats and recognized differences of representation, like that between Cartesian and polar coordinates, the general problem of reusing procedure functionality is unsolvable, since functional equivalence is undecidable.

The two major mechanisms for interoperation are interface standardization and interface bridging:

interface standardization: map client and server interfaces to a common representation
interface bridging: two-way map between client and server

Interface standardization is more scalable because *m* clients and *n* servers require only *m+n* maps to a standard interface, compared with *m**n* maps for interface bridging. However, interface bridging is more flexible, since it can be tailored to the needs of specific clients and servers. Interface standardization makes explicit the common properties of interfaces, thereby reducing the mapping task, and it separates communication models of clients from those of servers. But predefined standard interfaces preclude supporting new language features not considered at the time of standardization (for example, transactions). Standardized interface systems are closed while interface bridging systems are open. Architectures for standardized interfaces may be considered a special case of interface-bridging architectures in which the bridge from clients to servers is replaced by two half-bridges from clients to the standard interface and from the standard interface to the servers. We examine CORBA, whose architecture is based on interface standardization, and then consider briefly architectures based on interface bridging, like megaprogramming languages.

CORBA [CO] realizes interface standardization by an object request broker (ORB) that serves as an adapter/transformer. The ORB handles communication among application objects and provides object services (system objects), and common facilities (library objects). Standard interfaces are specified in an interface definition language (IDL) and stored in an interface repository. Interfaces of application objects specified in a variety of object-based languages are mapped to IDL by a language mapping that automatically maps language-dependent interfaces into language-independent IDL interfaces.

The ORB accepts requests from client objects through an IDL stub or a dynamic invocation for the services of server objects, transmits them for execution to the server, and returns the result to the client. It also accepts calls to system objects from both client and server objects and object-specific calls through an object adapter, as shown in Figure 10.

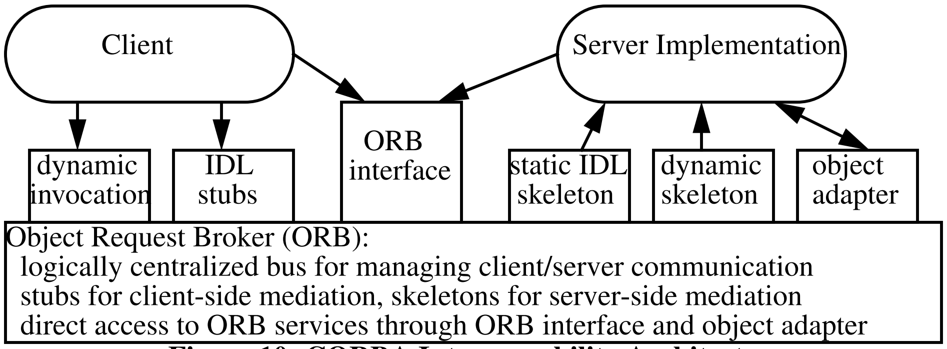

Client
Server Implementation
dynamic invocation
IDL stubs
ORB interface
static IDL skeleton
dynamic skeleton
object adapter
Object Request Broker (ORB):
logically centralized bus for managing client/server communication
stubs for client-side mediation, skeletons for server-side mediation
direct access to ORB services through ORB interface and object adapter

Figure 10: CORBA Interoperability Architecture

CORBA provides mediation services (half-bridges) from clients to the ORB and from the ORB to servers. Clients may invoke a service through a static stub or through a dynamic invocation created from the IDL specification at run time. The ORB validates client requests against the IDL interface and dispatches them to the server where arguments are unpacked (unmarshalled), methods are executed, and results are returned. Server-side software includes object adapters that bind object interfaces and manage object references, and a server skeleton that uses the output of object adapters to map operators to the methods that implement them.

Microsoft's Component Object Model (COM/OLE) [COM] realizes interoperability through a binary (machine-language) standard that specifies multiple interfaces by a pointer to a function table and objects by a principal interface I-unknown for accessing a directory of interfaces through a "queryinterface" function. COM/OLE objects have multiple interfaces that share a common data structure. The multiple-interface model for objects is more scalable than the hierarchical inheritance model of object-oriented programming, providing greater flexibility for both interoperation and extension of object functionality. Class-based inheritance is less scalable than multiple interfaces as an organizing principle for interfaces, both because complex objects are naturally modeled by multiple interfaces and because hierarchies are too restrictive a structuring principle. Use-case models [Ja], as well as COM/OLE, elevate collections of interfaces that share a state into a primary structuring principle of software design.

COM/CORBA interoperability, which aims at compatibility between Microsoft and CORBA-compliant components, is being pursued by a broad industry consortium under the auspices of the object-management group (OMG). Proposals for COM/CORBA interoperability aim to realize interoperation by interface bridging. Clients send a request via a surrogate object to a state that marshals the arguments and sends them across a bridge to a server skeleton that ummarshals the arguments and manages the implementation of operations by methods of the target object. The combination of interface standardization within CORBA and interface bridging to COM provides a balance between closed efficiency and open extensibility expressed by the metaphor of closed islands connected by bridges.

The interface-bridging approach to interoperation is illustrated by megaprogramming languages [WWC]. Megamodules capture the functionality of services provided by large organizations like banks, airline reservation systems, and city transportation systems. They are internally homogeneous, independently maintained software systems that embody the "software community" paradigm, encapsulating not only data but also knowledge, programming traditions, goals and values, and their own terminology, concepts, and interpretation paradigm (called an ontology). Megamodules control their own evolution and can add interface functionality as well as inner functionality. For example, a megamodule for a city transportation system can add new trucking companies, bus companies, and airlines to its interface. Such interface functionality can be modeled for example by COM's add-interface function.

The autonomy of megamodules in computing systems has many dimensions. Multidatabase technology distinguishes between *design autonomy,* which localizes control over system changes, *execution autonomy* which localizes control over system execution, and *communication autonomy,* which localizes access to data. Asynchrony allows components to execute autonomously in time. Heterogeneous interfaces are still another form of autonomy (representation autonomy) that requires translation (transduction) mechanisms to be interposed between senders and receivers:

design autonomy: local control over system changes
execution autonomy: local control over system execution
communication autonomy: localizes access to data
representation autonomy: local control over interface representation

Autonomy is natural in a world where programmers develop heterogeneous modules (megamodules) primarily for local use, ignoring the fact that the application might at some future time interact with remote clients. For example, the San Diego transportation authority might develop a model of transportation without considering that it might at some later time require an interface to airline schedules or transportation systems in Chicago. When an interface is provided, it is unlikely to be compatible with clients using different software platforms and object models. Megaprogramming languages regulate interaction among autonomous modules (megamodules) and where necessary translate messages from the output language of a sender to the input language of a receiver. Megamodules are created and maintained by a software community with a uniform terminology (ontology) that provides a conceptual framework and language for problem solving in an application domain that may differ from the ontology of software communities with whom the megamodule needs to communicate. The problem of communicating among megamodules is similar to that of communicating among countries that speak different languages.

The basic unit of interoperation in the client-server paradigm is the procedure. But procedure-level interoperation is not a sufficient condition, though it is a necessary one, for interoperation of software components [NT]. Software components may require larger-granularity units of interoperation, since the correspondence between client and server operations may not be one to one. Moreover, interoperation may require preservation of temporal as well as functional properties (order constraints on operations, coordination of inputs from multiple input streams). Such protocol constraints cannot be captured by functional correspondences of individual operations.

7. Design Patterns: Specification of Interaction

Patterns are a general tool for describing regularities in any domain of discourse. In the context of interactive modeling, patterns are useful in the description of observable behavior of software components, static structure of software systems, and processes of product development and design. Design patterns are reusable regularities of products or processes of design. The product of design is a component, called a framework, whose observable behavior can in principle be specified as a pattern of interaction. But patterns of interaction are too low level for designers concerned with regularities that help them create a design from a high-level specification. Design patterns specify high-level regularities at the level of the problem rather than machine-language interaction patterns. Developing a high-level language for patterns of design is harder than developing high-level algorithmic languages. Design patterns are specified in [GHJV] by the problem, the solution, and design trade-offs associated with the problem:

pattern = (name, problem, solution structure, consequences and trade-offs)

This format for specifying design patterns is independent of both domain and granularity and can be specialized when applied to object-oriented design. Object-oriented patterns are described by their *scope*, which specifies whether the pattern applies to *classes* or *objects*, and by their primary *purpose*, which may be to create a class or object (*creational*), to compose a structure out of components (*structural*), or to specify the interaction of a group of components (*behavioral*).

The model-view-controller (MVC) paradigm has been widely used to illustrate the role of patterns in design. It specifies design by three interdependent perspectives: a *model* that represents the application object, multiple *views* that capture syntactic interfaces, and a *controller* that defines the mode of execution. The model and controller correspond to the object and dynamic models of OMT, while MVC views are omitted in OMT but correspond to use cases in Jacobson's object design model. Though the MVC paradigm is a design pattern, its granularity is too great for inclusion by [GHJV] in their design catalog. MVC is described in terms of three component patterns:

observer: a pattern to describe one-to-many sharing of a resource (object) by multiple interfaces
composite: a pattern to describe composite nested structures
strategy: a pattern for run-time selection among multiple implementations of the same algorithm

Observer describes the many-to-one relation between a model and its views in an abstract reusable way, composite allows nested many-to-one structures, and strategy handles multiple algorithms for realizing controllers associated with views. These three patterns represent independent primitive components that may individually be used in many other contexts that work nicely together in realizing a general form of the MVC paradigm. Each pattern would be very difficult to define by composition of lower-level algorithmic or object-based primitives. Patterns introduce reusable primitive units of behavior that are not easily reducible to or expressible in terms of programming language primitives.

Patterns should be specified so it is easy to determine when they are applicable and should have well-defined dimensions of variability. For example, sort procedures may be viewed as patterns with well-defined applicability and parameters for varying the data and size of the set of elements to be sorted. Design patterns have more complex criteria for applicability that include a description of the class of problems the pattern is designed to solve and a description of the results and trade-offs of applying the pattern. The problem specification and results for sorting can be formally specified, while the problem class and effects of interactive patterns cannot generally be formalized and require a sometimes complex qualitative description. Though there are loose analogies in scaling up from algorithmic to interactive patterns, the details of pattern specification are entirely different.

Patterns facilitate codifying of design experience that has in other design domains like architecture been perceived to be elusive and unformalizable. A systematic method of describing, cataloging, and using design patterns provides clues to the process of design for both software systems and other artifacts. Frameworks provide an alternative approach that was quite successful in specifying reusable designs at a fairly high level of granularity and was a primary catalyst in developing the pattern concept. In [GHJV], frameworks are viewed as executable implementations of particular application program interfaces (APIs) that may contain implementations of several patterns. They are less flexible in both their granularity and their level of abstraction than implementation-independent design patterns.

Object models use empirical observation of objects of the application domain to model objects of the design [Ru]. However, designs generally include a richer set of objects than those in the modeled domain and require communication and composition protocols among objects to be modeled. Object-oriented design provides little help with structuring collections of objects into cohesive and reusable subsystems. Patterns for object design are closely tied to objects, classes, and object composition mechanisms. Object interfaces are specified by the set of signatures of an object's operations and determine an object's behavior and its type. Classes specify both the interface and the implementation of objects of a class so that a given interface type can be implemented by several classes. Abstract classes that defer implementation of all their operations to subclasses could in principle be used to realize implementation-independent interface inheritance, but would require the client to do the work of implementing all operations. Implementation-independent access to the functionality of an interface is better realized by object composition than by class composition. The design patterns described in [GHJV] include patterns for object composition.

8. Coordination: Management of Interaction

Coordination is concerned with the management of interactions among components of a composite system. The entities being coordinated may be user-level services like airline reservations, system-level services like those of an operating system, or algorithm-level services like instructions of a program. Control structures of algorithms are a highly-constrained (degenerate) form of coordination. Coordination of interactions among objects and components of a software system is subject to looser constraints and therefore gives rise to richer architectures. Models of coordination are concerned primarily with coordination among components: the term coordination has a specialized meaning in this context.

Coordination was popularized by the language Linda [CG], which distinguishes between *coordination primitives* of specifying interactions among processes and *computation primitives* for executing algorithms within processes. The study of coordination independently of computation is becoming an important area of research that has spawned several conferences and workshops over the past few years [CNY, CH]. Recent work on coordination models and their applications may be found in [AHM].

Linda processes interact through a shared data space called the "tuple space". Processes cannot interact directly: they post output to the tuple space which may then by used as input by other processes (see Figure 11). Because the tuple space is like a blackboard on which public notices may be posted, this model is referred to as the blackboard model Linda has four coordination primitives, three of which simply output and input data into the tuple space. Linda may also output live (executing) processes into the tuple space using the "eval command:

out(t): output the passive data tuple t to the tuple space

in(t): input a tuple with the value t from the tuple space, removing it from the tuple space. If no tuple with value t exists, the executing process blocks until a tuple with this value becomes available.

rd(t): copy a tuple with the value t from the tuple space leaving t in the tuple space for others to use. If no tuple with value t exists, the executing process blocks until a tuple with this value becomes available.

eval(t): output a live tuple t and execute it; when execution is completed the tuple becomes passive and can be accessed by in or rd commands of processes that use the result.

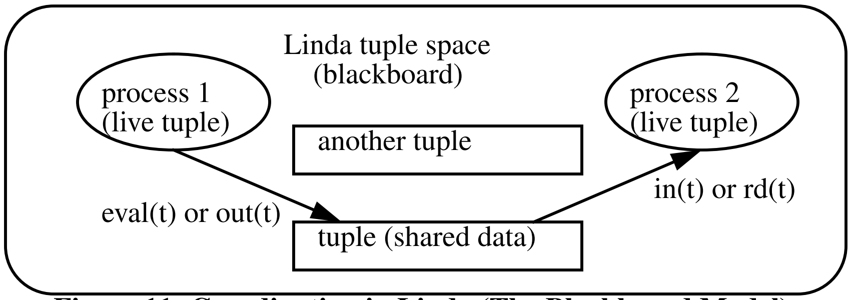


Figure 11: Coordination in Linda (The Blackboard Model)

Linda's coordination primitives constitute a very simple coordination language that can be combined with a variety of different computation languages like C or Fortran. Inner program structure and outer communication structure can be expressed by distinct and independently specifiable paradigms of coordination. Linda captures a particular (blackboard) coordination architecture that generalizes the producer-consumer paradigm of resource management to multiple processes sharing a pool of common resources. The formal properties and semantics of Linda's coordination primitives are examined in [CNY].

The blackboard paradigm is a particular object-based architecture: objects and processes effectively have an internal blackboard whose protocols of inner sharing are like those of blackboards though they are not a part of the public system semantics. We can think of objects as mechanisms for coordinating collections of tightly-coupled operations by sharing a local state. Sequential object-based languages have a restricted coordination mechanism strictly weaker than that of concurrent languages. Whether or not access to internal blackboards is controlled by a transaction manager is not observable or controllable by the system. Concurrent coordination semantics that guarantees atomicity of communication can be achieved only by sacrificing efficiency. In general atomicity cannot be guaranteed so that the behavior of concurrently accessible processes with nonserializable inner coordination architectures must be assumed chaotic in the absence of explicit guarantees to the contrary.

The coordination language Gamma [BL] uses multisets (sets with multiple instances of elements) as the basic data structure. Multisets make minimal assumptions about relations among elements compared with arrays, lists or other common data structures, allowing data structures to be defined without artificial sequentiality. For example, the Gamma reduction rule:

max: x,y -> y <= x < y

replaces ordered pairs (x,y) by y for any x<y until only the singleton maximum element remains.

This Gamma specification can be implemented either sequentially or concurrently. A sequential implementation is accidentally rather than necessarily sequential, since the implementation mapping loses information about control and does not reflect the specification semantics. In contrast, implementation mappings for procedure-oriented implementations preserve control structure of the source program.

Chemical reactions are an appropriate computation metaphor for abstract computation on multisets: reactions among molecules in a chemical solution are determined solely by chemical affinity independently of any externally imposed order or control structures. The very loose (nonexistent) coordination structure of multisets allows the chemical reaction metaphor to model both the coordination of loosely-coupled software components and tightly-coupled instructions of an algorithm. Coordination of actions within algorithms can be expressed by adding specialized control structures to multisets.

The elements of multisets may be of any kind whatsoever, including multisets. Multisets with reduction and interaction rules are a model for components, and multisets with multisets as elements can model components. "Second-order Gamma" models multisets with multisets as components and can in principle express the coordination of components that are modeled by multisets within a larger multiset.

Coordination of software components can be viewed as "second-order" computation, while execution of instructions is "first-order" computation. Coordination of components deals with the management of behavior of components composed of first-order actions, just as second-order logic is concerned with the behavior of functions and predicates of a first-order language. Second-order logics capture both the mathematics and the intuitive semantics of models of coordination and interaction. The set of all second-order entities (functions, predicates) over an enumerable set like the integers is nonenumerable, just as the inputs of an interaction machine. Second-order logics pay the price of not being sound and complete: this follows from the fact that the set of functions and predicates which they manage is nonenumerable.

Because coordination behaviors have more degrees of freedom than algorithm behaviors, coordination models are more varied than algorithm models. Petri nets are a "pure" model of coordination whose transitions model the process of consuming and creating resources independently of the purposes for which the resources are used. A Petri net is a graph structure with two kinds of nodes:

places that hold tokens representing resources

transitions that fire by consuming a token from each input place and sending a token to output places

Tokens are represented by multisets and coordination is realized by rewriting rules for multisets that can easily be modeled in the language Gamma. Petri nets model resource consumption of tasks that require multiple resources to execute and expressing distributed coordination that depends on the availability of resources at distributed locations (places). They abstract away from specific data structures by permitting only a single data structure (tokens) and abstract away from algorithms by modeling resource consumption and creation independently of the task for which resources are used.

Petri net tokens are nonreusable resources better modeled by linear logic than by traditional logics. Linear logic treats rules of inference as resources that are consumed when they are applied. It provides a framework for coordination of nonreusable resources that admits both sequential and parallel coordination, illustrated by *interaction abstract machines* in [ACP].

Coordination of heterogeneous distributed components can be realized by megaprogramming languages [WWC] that coordinate execution among megamodules by "megaprograms" whose statements specify sequences of coordination actions interspersed with transformation of message data from the format of sending megamodules to that of receivers. Megaprograms for simple coordination tasks can be very simple. However, coordination of distributed concurrently executing tasks with atomicity and real-time constraints can be arbitrarily complex. Megaprograms are examples of "middleware" interspersed between components to realize coordination.

component (resource, megamodule) <--> mediator (middleware, megaprogram) <--> component

Coordination elevates middleware to a first-class status, corresponding to the role of management in large organizations. Middleware mediates among software components by transforming data and coordinating actions: middleware for mediating and coordinating software components is examined from several viewpoints in [Surv]. Garlan defines software architecture in terms of the middleware for coordination and interaction among software components, Nierstrasz and Meijler explore the technology of software component composition, Wiederhold proposes mediators as a uniform mechanism for component coordination, Manola and Heiler examine issues of interoperability, and Sutherland examines middleware for business objects. These diverse papers exhibit a remarkable similarity in their overall goals in exploring middleware mechanisms for component coordination, while demonstrating the complexity of the problem and the diversity of approaches currently being considered.

Explicit coordination becomes more important as systems become large, just as explicit management structures are more important for large than for small companies. However, experience shows that simple typeless coordination systems like UNIX pipes or http are often more effective than more elaborate strongly typed interface definition languages. Static type-compatibility requirements that promote safety and efficiency appear to have unacceptably high implementation cost in today's coordination technology.

Coordination among heterogeneous components by OMG's CORBA and Microsoft's COM solves the problem of reusability of resources specified in one environment by components in another environment. Most work on interoperability assumes a client-server, object-based model of communication [NT]. Coordination among heterogeneous distributed components may utilize protocols developed for client-server compatibility in more general contexts of coordination.

Coordination is concerned both with rules for scheduling and firing actions and with communicating and transforming data among components. Executing actions and communicating data, which require very different models (Petri nets and CORBA,) need to be integrated:

coordination -> firing rules + data exchange

Designers of coordination languages must address the following issues [Proc]:

1. What are the entities being coordinated?
procedures, objects, processes, components of a specific type, subsystems
2. What are the media (architectures) for coordination?
blackboard model, client-server, Petri net, Pi calculus, CORBA, COM, UNIX pipes, middleware
3. What are the protocols and rules of coordination?
multiset rewriting rules, message send and receive, object request brokers, megaprogramming, HTML

In [Proc] it was further suggested that coordination languages be classified according to the dimensions of scalability, encapsulation, decentralization, dynamicity, open-endedness, generativeness, and semantic richness. These dimensions were used to classify and justify design decisions of existing languages. Though these dimensions are useful, the design space for coordination is not well understood and more work is needed to characterize and explore it.

9. Agents: Dynamical Systems, Control Theory, and Planning

Agents are proactive components that act on behalf of someone or something else to realize specified goals. Dynamical systems are an interactive abstraction whose continuous form has been widely studied in applied mathematics and whose discrete form models agents. Planning involves choosing a strategy for acting (interacting) that enables an agent to realize goals: plans can be viewed as strategies for controlling agents and therefore as an application of control theory. In exploring trade-offs between interactive and algorithmic models for controlling interactive behavior of agents and dynamical systems, we find that algorithmic, off-line planners have great complexity, while on-line, interactive planners sacrifice algorithmicity to realize more effective plans with lower complexity and greater flexibility.

Artificial intelligence focused primarily on logic and algorithms in the 1960s and 1970s but is becoming increasingly interactive. The conflict between logic-based and agent-oriented AI is documented in [Gr], which traces the demise of early research on neural nets and perceptrons, the increasing dominance of logic-based models spearheaded by research groups at MIT, CMU, and Stanford, and the reemergence of distributed AI in the 1980s. During 1985-1995 AI has increasingly focused on interactive models. Textbooks like [RN], which systematically reworks all subareas of AI in terms of models of intelligent agents, suggest that the agent-oriented perspective is a foundation rather than merely an advanced topic of AI. The 1995 special issue of the journal *Artificial Intelligence* on interaction and agency [AI] has over 700 pages of articles exploring the emerging interaction paradigm.

Agents function interactively in an environment that may contain other agents, performing tasks on behalf of other agents or autonomously on their own behalf. They perceive their environment through sensors and act upon it through effectors. Human agents have eyes and ears for sensors and hands, legs, and mouth for effectors, while robots have cameras and infrared range finders for sensors and motor-driven arms and wheels for effectors. Whereas software components are typically passively responding servers, agents realize goals by proactive interaction with the environment. Agents may be classified by the degree to which they passively react to or actively control interaction with their environment. Active agents view their environment as a passive domain to be observed or explained, while passive server agents have an active environment that directs and controls their actions.

Agents balance built-in cleverness with interactive adaptability in realizing computational goals. The degree to which agents use inner versus interactive cleverness is another dimension of classification: agents range from algorithms that rely entirely on inner cleverness to interactive identity machines that rely entirely on interaction. Agents with inner cleverness are more self-reliant, but interactive utilization of external cleverness, though dependent on external help, is potentially more powerful.

Dynamical systems are agents with a state and a dynamical law that governs how the state changes over time. The intensive study of continuous dynamical systems like Newtonian models of the solar system has over the last 300 years provided many insights applicable to the study of discrete agents in AI and computer science. Continuous differential equations with initial conditions are a discrete analog of algorithms with initial inputs: both determine closed dynamical systems. Physics and computing have primarily considered closed dynamical systems whose initial conditions are given prior to the beginning of the computation. Open dynamical systems that receive unpredictable interactive stimuli do not have a large body of mathematics from which analogous discrete models may be developed.

Dynamical systems with specified patterns of interaction over time can be reduced to closed noninteractive systems by modeling the interaction pattern as an agent that acts as an environment. A controller for a chemical plant or nuclear reactor that regulates a dynamical system to control its behavior may be viewed as a two-agent system: the system being controlled and the system that controls it may be viewed as coupled interacting agents that together form a closed system. Unpredictability (uncontrollability) of the input is an essential ingredient of openness.

There is a close analogy between initial inputs of algorithms and boundary conditions of differential equations. One-point boundary conditions correspond to arguments of a function or the tape of a Turing machine, while distributed boundary conditions correspond to closed systems like multitape Turing machines whose inputs are supplied prior to the beginning of the computation.

The classical Newtonian model, which assumes that the universe runs like a clock from initial conditions, is a continuous analog of an "algorithmic" dynamical system. Once the initial condition is given, Newtonian systems become nonempirical because no further observation is needed to predict their behavior. However, closed deterministic models of the universe have interactive open models as subsystems, both because isolated subsystems cannot know about or control interaction with the rest of the universe and because even predictable interactions may simply be too complex to be describable. Thus interactive subsystems are necessary in describing even closed, deterministic, nonempirical worlds.

Control theory studies two-agent systems where one agent, called the controller controls the behavior of agents (processes) being controlled. Dean and Wellman [DW] apply control theory to planning, where the controller/planner formulates plans to control the behavior of agents to realize specified goals. The controller/planner and agent being controlled are independently designed, loosely coordinated open systems. Planners must maintain a model of agents whose behavior they are controlling, tracking behavior either by monitoring through interactive sensors or prediction by algorithms. Planners that interactively monitor behavior are closed-loop, on-line systems while predictive planners are open-loop, off-line, closed systems. Closed-loop systems are in our sense open while open-loop systems are closed. Figure 12 indicates the relation between a controller/planner and the agent (embedded system) that it controls.

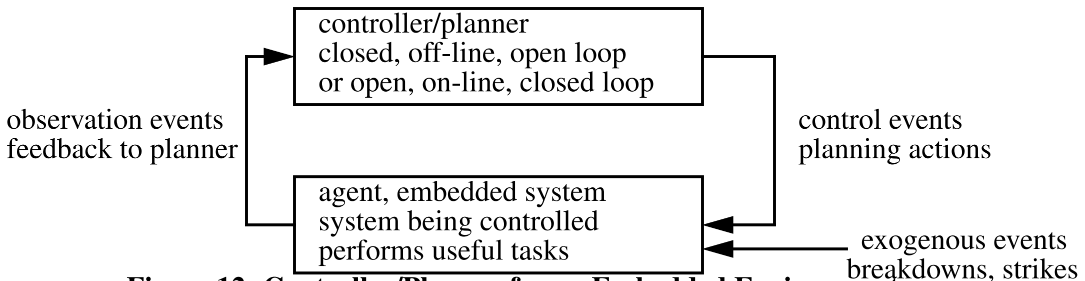


Figure 12: Controller/Planner for an Embedded Environment

Control theory is concerned with both controllability and observability of processes, where a process (dynamical system) is said to be controllable with respect to a set of goal states if it can reach and remain in a goal state and is said to be observable if its state can be identified by observing its outputs. Controllability and observability are dual concepts: constraints on inputs needed to guarantee reaching a goal state are dual to constraints on outputs needed to guarantee observability of the inner state. Conversely, uncontrollability of interactive inputs to a process from the environment is dual to uncontrollability of inner actions of a process by the environment.

Duality of controllability and observability in dynamical systems reflects the inherent duality between agents being controlled and the agents that control them. In a closed-loop environment, each determines an environment for the other: the agent provides an environment for the controller and the controller provides an environment for the agent. Control is realized by messages from the controller to the agent while observation is realized by messages from the agent to the controller.

Agents controlled by plans can be classified by whether they have a closed algorithmic model or an open interactive model. Open interactive planners are more adaptable: they can react to unpredictable exogenous events like breakdowns, strikes, and earthquakes. Open interactive planners are more expressive and efficient than closed algorithmic models, but closed models are preferred in many contexts because one can reason about their properties. For example, exogenous events are often approximated by probability distributions to transform inherently open systems with unpredictable events into closed systems with predictable statistical properties. Agent behavior is often approximated by probability distributions to permit planners to construct off-line plans that will on the average perform well.

Characterization of components of a system by whether they are open or closed tells us something fundamental about their qualitative behavior and expressive power. For example, the design of off-line planners involves single-minded algorithmic considerations, while on-line planners trade off reasoning power for greater efficiency, flexibility, and expressive power. The domain of agents, control theory, and planning clearly illustrates the trade-offs between algorithmic tractability and interactive openness that occur in many other AI and software engineering domains.

10. Virtual Reality: A Case Study in Interactive Architecture

Paradigms and models of computation can be classified by the relative importance and role of algorithms and interaction in the overall problem-solving task. Autistic systems with no connection to the outside world are completely noninteractive, mathematical and algorithmic models permit just a single interaction, while interactive models support patterns of interaction over time:

autistic system: no interaction
predicate (pure logic programming): stimulus -> binary response
function: stimulus -> range of response
Turing machine: initial input tape to final output tape
object: input stream to output stream
graphical user interface: input stream to stream of two-dimensional images
robots, software components: multiple inputs to multiple outputs
virtual reality: multisensory real-time inputs to multisensory real-time outputs

Predicates, functions, and Turing machines differ in their interactive and algorithmic mechanisms but have the same expressive power. Objects, GUIs, robots, and VR have greater expressive power than Turing machines and progressively richer interaction mechanisms. GUI, robot, and virtual reality paradigms differ in the degree to which the computing agent, its environment, or both are active or passive agents:

GUI paradigm: the computing agent is passive and the client in the environment is active
robot paradigm: the computing agent (robot) is active and the environment is generally passive
virtual reality paradigm: the virtual reality agent and the human client are both active

Virtual reality systems must coordinate a variety of modes of interaction to create the illusion of immersion in a fictitious or remote environment. The illusion is created by coordinated, real-time, multimodal stimulation by a variety of sensory mechanisms:

visual sensors for visible and electromagnetic stimuli
auditory sensors of sound waves
olfactory sensors of chemicals in atmosphere
gustatory sensing of chemicals that stimulate the tongue
haptic sensors of touch including:
  tactile sensors of temperature, texture, pressure
  kinesthetic sensors of force by muscles, joints, tendons
  spatial sensors of limbo/torso positions and angles
  motion sensors of linear and angular acceleration (inner ear)

To provide a feel for architectures to handle multimodal interactive effects, we briefly examine the structure of virtual reality computers (VRCs). A VRC has two clients (a human user and a model of reality) and two primary components:

user interface manager: transmits multisensory view of model to the user
  handles changes in perspective due to motion or attitude changes of the user
model manager: manages real-time model updating and environment interaction
  handles changes of model due to change of the virtual environment

Changes in perspective are largely model-independent and can be handled independently of the model itself. The model may be stored locally within the VRC, nonlocally within a remote computer, or in a distributed form sharable by multiple agents and/or multiple remote virtual reality sites. The complexity of the model management may range from simple storage of a static local model to time-critical updating of shared distributed information. For slowly changing models, supplying incremental change information may be the most efficient way of updating the model.

A VRC can be conveniently described by loosely coupled user-interface and model management components. Multiagent systems must provide a common distributed model of the environment (for example a surgical operating room) for a group of users. The task of maintaining such a model is a hard distributed networking problem with stringent real-time constraints that must handle multiple interfaces sharing a common state. However, it is entirely separate from the problem of creating the illusion of immersive interaction for individuals through a virtual reality interface. These two tasks are concerned with different aspects of interaction and together provide a rich environment for case studies in interactive behavior.

Figure 13 shows two users U1, U2, associated virtual reality computers VRC1, VRC2, and a remote telepresence computer TPC interacting with users through a distributed network. Virtual reality computers have a VR user interface that handles realistic multisensory immersion and a network interface that handles communication with other users and with the remote environment. The inner architecture of network and model management coordinates interaction among the VRCs and TPC through interfaces that share a common state, while the outer architecture handles high-bandwidth interactions with users to create the effect of immersion and with the remote environment to handle its representation and updating.

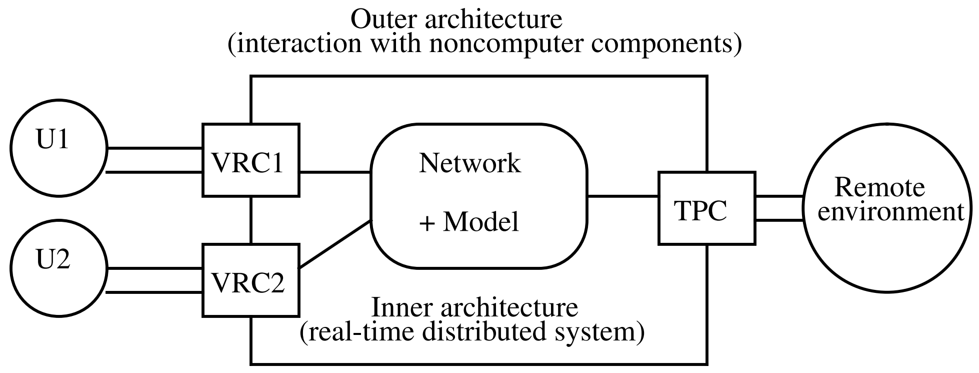


Figure 13: Interactive Architecture for Multiagent Virtual Reality

11. Data Information Systems: A Case Study in Empirical Computer Science

The scientific method operates by data acquisition, discovery of interesting regularities in the data, and the development of models for understanding, predicting, and changing the environment (see Figure 14). Large-scale information systems that follow this paradigm include NASA's earth observation system distributed information system (EOSDIS) and digital library systems that combine data acquisition and user services. Architectures of distributed (or database) information systems (DISs) includes a data repository for the mass storage of raw data. The analysis and recognition of data regularities is handled by a feature editor that catalogs the raw data and accumulates feature descriptions as well as summary data in a metadata repository. The process of discovery of significant features of observed data is called data mining.

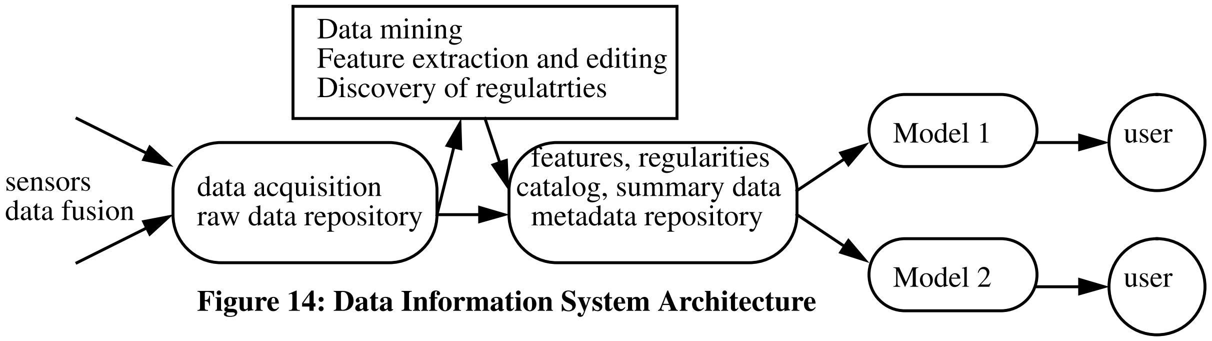


Figure 14: Data Information System Architecture

The metadata is interpreted by multiple models that focus on views of the data for a variety of end-users. Geographers and ecologists focus on different models of raw data of the earth observation system, while anthropologists and historians focus on different aspects of a library corpus. Though end-user views of data are largely independent, cooperative analysis by coordinated loosely-coupled specialists can be critical in realizing higher-level goals. Since specialist interfaces are determined by preestablished traditions of their user communities, an interoperation layer is needed to coordinate communication.

Data information systems must handle three high-level tasks (data acquisition, cataloguing and feature extraction, and multiple user models), each of which has its own research community. Data acquisition requires familiarity with sensor hardware and has historically been handled by hardware engineers. Cataloguing can benefit from techniques of information retrieval, while feature extraction (data mining) has been studied using tools of spectral analysis.

User interface modeling is the most software-intensive and also the most neglected of the three DIS tasks. Coordinated access to a data repository by multiple users through a variety of different models or views is a major unsolved problem in both database technology and software engineering. Problems of interoperation have been recognized as a major unsolved problem for which techniques of CORBA, COM, and megaprogramming provide only partial solutions. Standardization on interface definition languages and data exchange formats is needed to make the interoperation problem more tractable. A high-level view of the back end of a DIS with multiple specialist views mediated by interoperation is given in Figure 15.

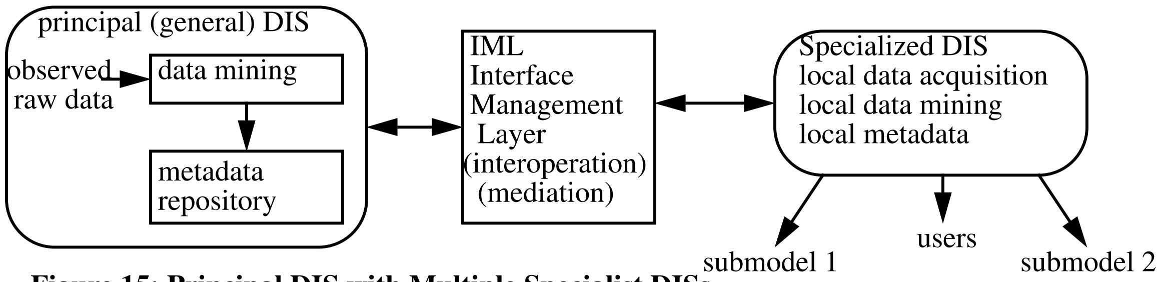


Figure 15: Principal DIS with Multiple Specialist DISs

The support of multiple specialist views of a general-purpose database requires both deep system expertise and domain-specific knowledge in each of the areas being supported. The interoperation layer must solve the very complicated problem of interoperation among components with different domain-specific interface domains. To manage multiple users, the IML must handle transaction management among interoperating system components. The complete design of systems for handling shared access to a database by specialist users is a complex open problem clearly beyond the scope of this paper. The ubiquity of this "scientific method" paradigm is illustrated by the fact that this architecture is common to both NASA's earth observation system and digital library systems.

NASA's earth observation system [Sh+] calls its specialist DISs regional data centers (RDCs). RDCs have local storage, may support local data acquisition and feature editing, and may have multiple users, so that they have the same structure as the principal DIS, though on a smaller scale. The principal DIS provides support for a broad user community, while RDCs provide support for specialized user communities that in the case of earth observation may correspond to geographical regions or resource management centers for water or timber resources. In the case of digital libraries specialist DISs may correspond to universities, or to disciplinary research centers for history, physics, or oceanography. End-users access the system through a user workstation that may have the full power of a specialist DIS or be a simpler interface with just a query language but no facilities for extraction of new features or acquiring new data.

Currently each domain-specific software application reinvents its own architecture for managing communication between general-purpose information systems and multiple specialist users. Interactive software technology can develop systematic architectures (high-level design patterns) that can be reused by applications as varied as earth observation systems and digital libraries to support communities of specialist users in a variety of application domains.

12. Conclusion

Interaction machines provide a unifying model for software engineering, artificial intelligence, and database technology. The distinction between object-based and component-based technology provides a framework for modeling embedded systems with multiple interfaces that share a common state. We have examined a variety of interactive technologies like OMT, COM/OLE, CORBA, design patterns, planning and control, and case studies of virtual reality, earth observation systems, and digital libraries, demonstrating that interactive models do indeed provide new perspectives and insights. However, we merely scratch the surface of a very large topic, providing the outlines of a new approach to modeling based on a notion of interactive computation richer than the algorithmic notion of computation of Church and Turing.

The paradigm shift from algorithms in the 1960s and 1970s to interaction in the 1990s is refocusing attention from inner processes of execution to interactive components. Though interactive processes are inherently less amenable to formal analysis than algorithms, they will play an increasingly important role in the software technology of the 21st century.

13. References

[ACP] J. Andreoli, P. Ciancarini, and R. Pareschi, Interaction Abstract Machines, in *Research Directions in Concurrent Object-Oriented Programming*, Eds Agha, Wegner, Yonezawa, MIT Press, 1993.

[AHM] J. Andreoli, C. Hankin, and D. Le Metayer, Coordination Programming: Mechanisms, Models, and Semantics, 1996.

[AWY] Gul Agha, Peter Wegner, and Akinori Yonezawa, *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[AI] *Computational Research on Interaction and Agency*, Special Double Issue on Artificial Intelligence, January and February 1995, Guest Editors: P. Agre and S. Rosenschein.

[Bl] Manuel Blum, Result Checking, MIT Distinguished Lecture, 1994.

[BL] J. Banatre and D. Le Metayer, Gamma and the Chemical Reaction Model, Ten Years After, in [AHM].

[Br1] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1995.

[CG] N. Carriero and D. Gelernter, Coordination Languages and Their Significance, CACM, Feb 1992.

[CH] Paolo Ciancarini, and Chris Hankin Eds, *Coordination Languages and Models*, LNCS #1061, 1996.

[CNY] Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, *Proc. ECOOP '94 Workshop on Coordination Languages*, LNCS #924, Springer Verlag, 1995, especially articles by Gelernter and Ciancarini.

[COM] Kraig Brockschimidt, *Inside OLE 2*, 2nd edition, Microsoft Press, 1995.

[CORBA]: Architecture and Specification, Revision 2.0, Object Management Group, July 1995.

[DW] Thomas Dean and Michael Wellman, *Planning and Control*, Morgan Kaufman 1991.

[Ga] David Garlan, Research Directions in Software Architecture, *Computing Surveys*, June 1995.

[GHJV] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of*

Reusable Object-Oriented Software, Addison Wesley 1994.
[Gr] Stephen Graubard Ed., *The Artificial Intelligence Debate*, MIT Press 1988.
[Ja] Ivar Jacobson, *Object-Oriented Software Engineering,* Addison-Wesley/ACM Press, 1991.
[Mil] Robin Milner, Elements of Interaction, *CACM*, January, 1993.
[NT] Oscar Nierstrasz and Dennis Tsichritzis, Eds, *Object-Oriented Software Composition*, Prentice Hall, 1996. Especially Chapter 3 by Dimitri Konstantas.
[Proc] *Proceedings of First Annual Workshop on Coordination*, Imperial College Dept of Computer Science, December 1994.
[RN] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach,* Addison-Wesley, 1994.
[Ru] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1990.
[Sh+] Nicholas Short et al, Mission to Planet Earth: AI Views the World, *IEEE Expert*, December 1995.
[Surv] Research Directions in Software Engineering, *Computing Surveys*, June 1995.
[We1] Peter Wegner, Interactive Foundations of Object-Based Programming, *IEEE Computer*, Oct. 1995.
[We2] Peter Wegner, Foundations of Interactive Computing, Report CS-96-01, May 1996.
[We3] Peter Wegner, Tradeoffs Between Reasoning and Modeling, in *Research Directions in Concurrent Object-Oriented Programming*, Ed Agha, Wegner, Yonezawa, MIT Press, 1993.
[We4] Peter Wegner, Interaction as a Basis for Empirical Computer Science, *Computing Surveys*, March 1995.
[Wi] Gio Wiederhold, Mediation in Information Systems, *Computing Surveys*, June 1995.
[WWC] Gio Wiederhold, Peter Wegner, and Stefano Ceri, Towards Megaprogramming, *CACM* Nov 1992.
[YS] Daniel Yellin and Robert Strom, Protocol Specifications and Component Adapters, IBM Report, 1996.