Tutorial Notes:
Models and Paradigms of Interaction

Peter Wegner, pw@cs.brown.edu

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-21
September 1995

Tutorial Notes: Models and Paradigms of Interaction

Peter Wegner, Brown University, September 1995

1. Interaction Machines (page 3)
 1.1. From algorithms to interaction
 1.2. Interactive identity machines
 1.3. Design space of interaction machines
 1.4. Synchronous interaction machines
 1.5. Asynchronous input and output actions
 1.6. Object machines
 1.7. Concurrent input actions
 1.8. Concurrency, distribution, and interaction
 1.9. Layers of protective abstraction
2. Paradigms of Modeling (page 19)
 2.1. What is a model?
 2.2. Models and their validation
 2.3. Robustness of interactive models
 2.4. Harness constraints on interaction
 2.5. The expressive power of interaction
 2.6. Algorithms and complexity
 2.7. Biological interaction machines
 2.8. Interactive reflection
3. Logic and Semantics (page 32)
 3.1. Irreducibility and incompleteness
 3.2. Interactive declarativeness
 3.3. Interactive types
 3.4. Programs are not predicates
 3.5. Modal, intuitionistic, and linear logic
 3.6. Temporal and real-time logics
 3.7. Logic and constraint programming
4. Distributed Interaction (page 42)
 4.1. Process models
 4.2. Transaction correctness as an interaction constraint
 4.3. Time, asynchrony, and consensus
 4.4. Abstraction and transparency
 4.5. Distributed programming languages
 4.6. Distributed operating systems
5 Software Engineering (page 49)
 5.1. Programming in the large
 5.2. Life-cycle models and the software process
 5.3. Observable behavior of objects
 5.4. Object-based design and use-case analysis
 5.5. Design patterns and frameworks
 5.6. Autonomy, heterogeneity, and interoperability
 5.7. Document engineering and coordination
6. Artificial Intelligence (page 59)
 6.1. Knowledge representation
 6.2. Intelligent agents
 6.3. Planning agents for dynamical systems
 6.4. Formal versus empirical (connectionist) models of intelligence
 6.5. Intelligence without reason
 6.6. Nonmonotonic logic and the closed-world assumption
 6.7. Machines that can think
7. References (page 65)

Abstract*:

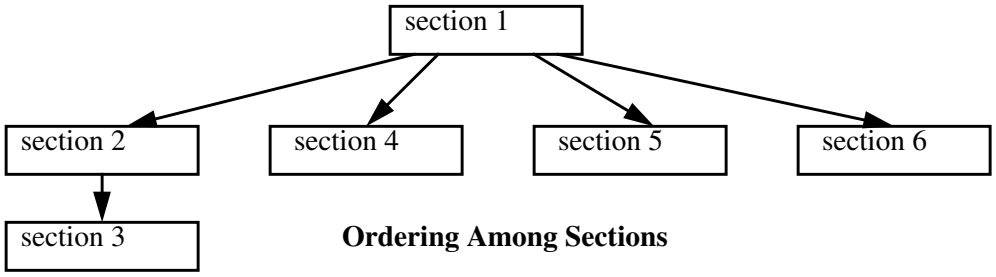
The development of a conceptual framework and formal theoretical foundation for object-based programming has proved so elusive because the observable behavior of objects cannot be modeled by algorithms. The irreducibility of object behavior to that of algorithms has radical consequences for both the theory and the practice of computing.

Interaction machines, defined by extending Turing machines with input actions (read statements), are shown to be more expressive than computable functions, providing a counterexample to the hypothesis of Church and Turing that the intuitive notion of computation corresponds to formal computability by Turing machines. The negative result that interaction cannot be modeled by algorithms leads to positive principles of interactive modeling by interface constraints that support partial descriptions of interactive systems whose complete behavior is inherently unspecifiable. The unspecifiability of complete behavior for interactive systems is a computational analog of Godel incompleteness for the integers.

Incompleteness is a key to expressing richer behavior shared by empirical models of physics and the natural sciences. Interaction machines have the behavioral power of empirical systems, providing a precise characterization of empirical computer science. They also provide a precise framework for object-based software engineering and agent-oriented AI models that is more expressive than algorithmic models.

Fortunately the complete behavior of interaction machines is not needed to harness their behavior for practical purposes. Interface descriptions are the primary mechanism used by software designers and application programmers for partially describing systems for the purpose of designing, controlling, predicting, and understanding them. Interface descriptions are an example of “harness constraints” that constrain interactive systems so their behavior can be harnessed for useful purposes. We examine both system constraints like transaction correctness and interface constraints for software design and applications.

Sections 1, 2, and 3 provide a conceptual framework and theoretical foundation for interactive models, while sections 4, 5, and 6 apply the framework to distributed systems, software engineering, and artificial intelligence. Section 1 explores the design space of interaction machines, section 2 considers the relation between imperative, declarative, and interactive paradigms of computing, section 3 examines the limitations of logic and formal semantics. In section 4 we examine process models, transaction correctness, time, programming languages, and operating systems from the point of view of interaction. In section 5, we examine life-cycle models, object-based design, use-case models, design patterns, interoperability, and coordination. In section 6, we examine knowledge representation, intelligent agents, planning for dynamical systems, nonmonotonic logic, and “can machines think?”. The interaction paradigm provides new ways of approaching each of these application areas, demonstrating the value of interactive models as a basis for the analysis of practical problems in computing. Readers interested in the applications of interaction machines can skip directly from section 1 to any of the application sections:



Ordering Among Sections

* This is an evolving document. Please send comments about omissions, clarifications, and better ways of saying things to: pw@cs.brown.edu.

1. Interaction Machines

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 an algorithmic specification so that (assuming no other traffic) a blindfolded person could drive 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.

Another example of the power of interaction, taken from military applications, is that of smart bombs. Smart bombs are interactive, checking observations of the terrain over which they fly against a stored "mental map" of their route and target. Their smartness is realized by supplementing preprogrammed routing specifications with interaction in exactly the way humans interact while driving. Interaction is clearly an important ingredient of intelligence in many application areas, making the difference between dumb and smart products (see section 6).

Software engineering is concerned primarily with embedded systems that provide interactive services over time and adapt to the needs of their clients. Airline reservation systems or interactive robots are inherently interactive. Interactive systems are a more accurate model of the behavior of actual computers than algorithmic specifications of transformations. Object-based technology has become dominant because it is interactive and therefore more expressive than algorithmic specifications. The irreducibility of object behavior to that of algorithms has radical consequences for both the theory and the practice of computing.

Algorithms determine contracts with their clients that are like a sales contract, delivering a unique output for every input. Objects determine a very different kind of contract with their clients that is like a marriage contract in providing a service for all possible contingencies of interaction (for richer for poorer) over the lifetime of the object (till death do us part). The irreducibility of algorithmic to interactive behavior is the computational analog of the irreducibility of marriage contracts to sales contracts.

The thesis of this work is that objects have observably richer behavior than algorithms or Turing machines. Algorithms have limited computing power because they are closed, noninteractive systems, while objects are open, interactive systems whose observable behavior more accurately models software applications, distributed systems, and actual computers.

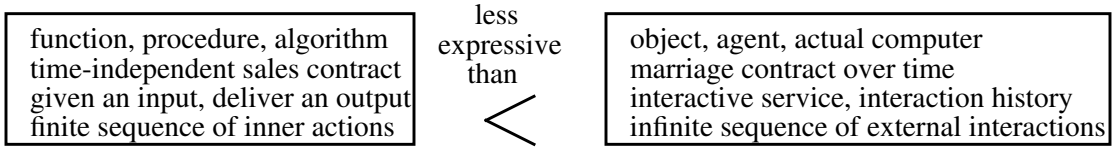


Figure 1: Sales Versus Marriage Contracts

The richer behavior of objects is modeled by *interaction machines,* which extend Turing machines with input actions (read statements). This simple extension of mechanism radically changes both behavior and formal properties, transforming closed Turing machines to open systems that express on-line interaction with external processes as well as the passage of time. Formally, this extension gives interaction machines the power of Turing machines with oracles and causes their "complete" behavior in all contexts of interaction to become mathematically unspecifiable in the sense of Godel incompleteness [We1].

Observability for algorithms is defined by computable functions that transform inputs into outputs. Interaction machines support fundamentally richer observable behavior not expressible by transformations. They support observability of sequences of events in time that may be infinite and not algorithmically describable. The patterns of observable interactions of an interaction machine can be inherently nonsequential, and the duration of both interactions and intervals between interactions may be significant. Whereas Turing machine behavior can be expressed by mathematical models, the observable behavior of

interaction machines corresponds to that of empirical systems in the natural sciences.

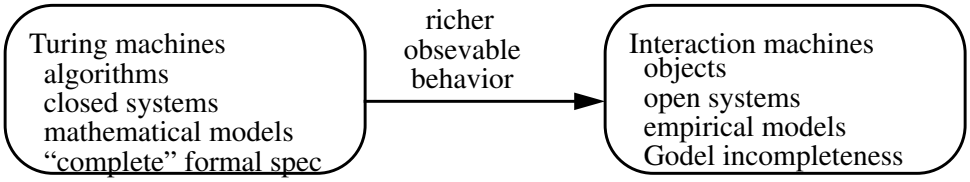


Figure 2: Interaction Has Observably Richer Behavior than Algorithms

The behavior of interaction machines can be effectively harnessed using methods of empirical rather than mathematical modeling. Instead of specifying complete behavior, we must be satisfied by specifying the partial behavior of interfaces, just as physics specifies physical properties of matter without necessarily specifying its chemical or biological properties. Users of interactive systems like airline reservation or personal computer systems are never exposed to the complete behavior of the system (that would be theoretically impossible) but see only partial behavior through specific interfaces. Interfaces encapsulate interactive systems in a layer of protective abstraction that constrains the mode of use to a tractable subset of behavior. Turing machines constrain the mode of interaction to the subset of noninteractive behavior, while interfaces of an airline reservation or personal computer system constrain the mode of use to subsets of interactive behavior.

1.1 From algorithms to interaction

Programming has evolved from machine language in the 1950s to procedure-oriented programming in the 1960s, structured programming in the 1970s, and object-oriented programming in the 1980s. In the 1990s methods for structuring and composing collections of objects are being developed, including frameworks, design patterns, and protocols.

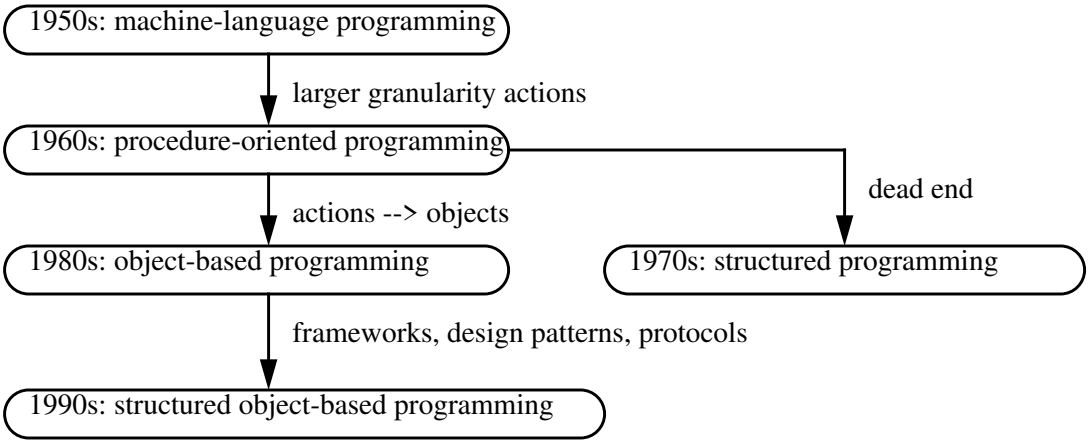


Figure 3: The Evolution of Programming Paradigmns

The transition from machine to procedure-oriented programming involves a quantitative change in the granularity of actions while retaining an algorithmic (action-based) programming model. The transition from procedure-oriented to object-based programming is a more radical qualitative change from programs as algorithms that transform data to programs as systems of persistent, interacting objects.

Algorithms compute by a finite sequence of inner actions. Objects compute by a pattern (interaction history) of externally initiated interactions not under the control of the object. Interaction histories may be infinite and are not in general algorithmically describable even in the special case when they are sequential. For example, sequences of deposit and withdraw operations of a bank account are not in general algorithmically describable, since the time of deposit or withdrawal and even the time taken to perform an

operation may affect behavior. Interaction histories need not be sequential: an airline reservation system must handle concurrent, temporally overlapping, input actions.

The *computable functions* are a very robust class of transformations that express the behavior of Turing machines, the lambda calculus, and algorithms of any programming language. Church and Turing conjectured in the 1930s that this robust notion of computing corresponds to the intuitive notion of what is computable. However, the mathematical intuitions of Church and Turing do not capture interactive behavior. The Church-Turing notion of computing is too restrictive to express object-based programming: it cannot express the behavior in time of objects, software systems, and actual computers. Object-based behavior, composition, and structure are not expressible by (reducible to) procedure-oriented modeling primitives (marriage contracts are not reducible to sales contracts).

Interaction is the key notion making object behavior more powerful than that of procedures. Functions compute their output from an initially specified input without any external interaction, while object permit interaction during a computation. Turing machines likewise compute output from an initial input tape and cannot interact with an external environment during a computation, as in Figure 4.

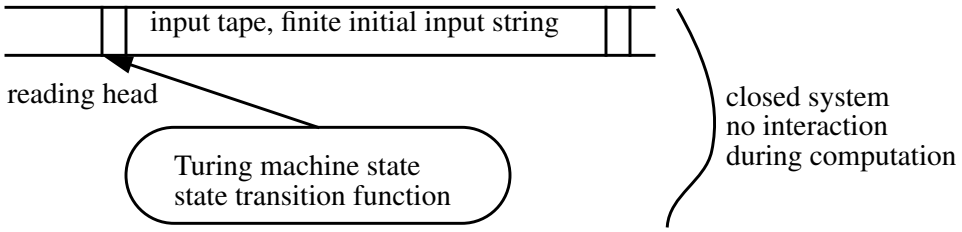


Figure 4: Turing Machine as a Noninteractive Computing Mechanism

Whereas Turing machines are closed systems that compute outputs from their inputs without interaction, objects are open systems that can interact while they compute. Objects are persistent reactive systems that continue to exist between the execution of algorithms and can model the passage of time. Their behavior in time cannot be captured by noninteractive Turing machine transformations.

However, Turing machines can be extended to be interactive by adding *input actions* supporting external inputs during computation. This simple extension transforms Turing machines from closed to open systems, extending their expressive richness to that of objects. We call Turing machines with input actions *interaction machines*. Interaction machines better approximate the behavior of actual computers than Turing machines: they can model the passage of time during the course of a computation while Turing machines cannot. Interaction machines can model the behavior of airline reservation and banking systems that interact with clients in real time, and of embedded, reactive, and hard real-time systems. They are a more accurate model of actual computers than Turing machines because actual computers persist in time and can interact with the external environment during the course of a computation.

Formally, interaction machines are more powerful than Turing machines because they capture the behavior of Turing machines with oracles [Tu]. Moreover, interaction machines are logically incomplete in the sense of Godel's incompleteness result for the integers: their properties (observable behavior) cannot be expressed by any sound and complete first-order logic (see section 2 below). These formal irreducibility results for interaction machines complement the informally demonstrable fact that interaction machines are richer than Turing machines in modeling interaction and the passage of time.

Turing and interaction machines are discrete dynamical systems that evolve over time. Turing machines are governed entirely by inner rules of evolution acting on a predefined input tape, while interaction machines evolve according to a combination of inner rules and environmental stimuli. Turing machines are the discrete analog of dynamical systems (differential equations) with one-point boundary conditions,

while interaction machines are discrete dynamical systems subject to impulses distributed over time.

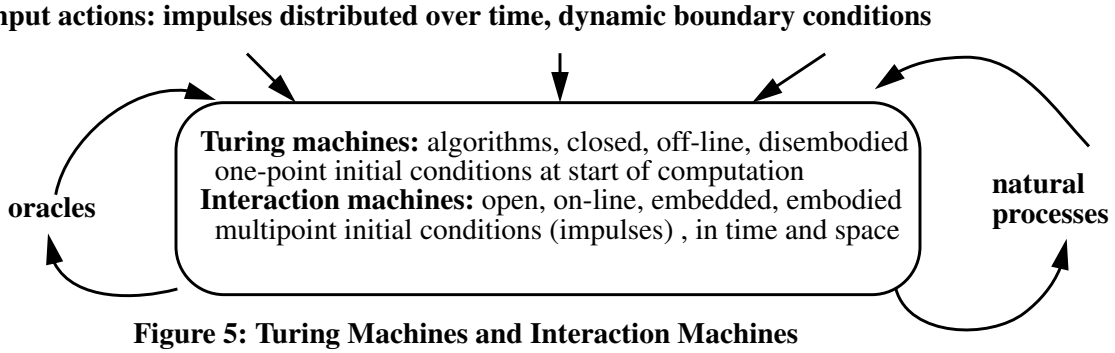


Figure 5: Turing Machines and Interaction Machines

Interaction machines generally have output as well as input actions: input actions are sufficient for behavior richer than Turing machines, while output actions allow two-way interaction with the environment. Input and output actions can be viewed as "logical" sensors and effectors that have a logical effect on data in their environment even when they have no physical effect. Interaction machines can directly realize embedded and reactive systems of software engineering and embodied systems of artificial intelligence. Conversely, embedded, reactive, and embodied systems are interactive and not in general expressible by Turing machines.

Turing and interaction machines may be compared by the metric of observable behavior. Interaction machines have strictly richer observable behavior because they can consult oracles to get the answer to nonalgorithmic computations, and can react to sequences generated by oracles or processes in nature that are not recursively enumerable. They can mimic algorithm behavior either by direct inner computation or by interaction with an external agent.

The observable behavior of interaction machines may be described by *interaction histories* consisting of algorithmic episodes embedded in an interactive context. Simple interaction histories (called traces) model interaction as a sequence of instantaneously executed algorithms, but interaction histories may in general assign meaning to the intervals between the execution of algorithms and to the time of algorithm execution. Even sequential interaction histories can be nonalgorithmic, while histories in general are not expressible as sequences. The behavior of distributed systems without a global notion of time cannot be expressed by linear interaction histories.The representation and modeling of time in interaction machines is discussed in sections 3.6, 4.3. Distributed systems that permit coordinated observation by (interaction with) multiple observers are considered in [BBHK].

The goal of guaranteed correctness for all possible computations is a casualty of greater expressiveness. This goal underlies language design principles like "goto considered harmful" and the more sweeping 'assignment considered harmful" of functional programming, which reduce program flexibility to realize correctness without reducing expressive power. The elimination of interaction in algorithmic models of computation (an application of the principle "interaction considered harmful") is a more extreme consequence of the goal of correctness, limiting the expressive power of models to realize correctness. However, correctness has a cost: reducing program flexibility to realize gotoless or functional programming has not been generally accepted by practicing programmers, while reducing program expressiveness to that of Turing machines is unacceptable to software engineers. Though this trade-off has resulted in a rich and worthwhile theoretical foundation for computing, models that sacrifice correctness for expressiveness must be developed to deal with interactive applications. There is much truth in the design principle "correctness considered harmful", since overemphasis on correctness has hindered the development of interactive models of computation needed for software engineering and distributed computing. Correctness has an important role in computational models but its overemphasis in the first 50 years of computer science has hindered the development of systematic models for object-based interactive programming.

1.2 Interactive identity machines

Both people and computers solve problems by a combination of algorithmic computation and interaction with the environment. Algorithms represent an extreme situation when no interaction is permitted. The other extreme of purely interactive problem solving is expressed by *interactive identity machines* that output their inputs without transforming them. Interactive identity machines have a rich potential behavior because they can harness arbitrarily complex behavior in their environment, but are limited in their actual behavior by the existence of external mechanisms that generate the desired behavior. An interactive-identity machine consists of a read-print loop that reads and prints a sequence of messages:

```
loop "interactive-identity-machine"   -- transmits interactive information without transforming it
  read(message); print(message);      -- input action followed by an output action
end loop
```

Interactive identity machines realize their behavioral power without inner computing because they can harness and replicate the computing power of external agents. They realize the "management paradigm": a good manager reuses and coordinates work without necessarily understanding it, catalyzing behavior richer than that of individual workers.

Interactive identity machines can mimic a Turing machine by printing its output tape, the input followed by the output tape, or the input followed by a step-by-step representation of the computation and the output tape. They can express the observable behavior of any Turing machine for any reasonable notion of behavior. They can also echo the behavior of nonalgorithmic oracles, generating strings of observable output behavior not generable by any Turing machine.

Theorem: Interactive identity machines have observably richer behavior than Turing machines.
Proof: Interactive identity machines can mimic the behavior of any Turing machine and also express the behavior of oracles and processes in nature whose behavior is not expressible by Turing machines.

Interactive identity machines employ the judo principle of using the weight of external opponents (or cooperating agents) to achieve a desired effect. For example, an interaction machine (or person) knowing no chess can win half the games in a simultaneous chess match against two masters by echoing the moves of one opponent on the board of the other. Eliza, which simulates a psychiatrist by "echoing" the patient's input, achieves "echo intelligence" by augmenting an interactive identity machine with simple algorithmic rules. The behavior of "chess by imitation" and Eliza can be realized by interaction machines that replicate inputs with a minimal amount of coordination and transformation. Chess is a finite game whose winning strategies could in principle be realized by an algorithm, though none has yet been developed. The behavior of Eliza, however, is open-ended and could not be realized by a Turing machine even in principle.

Analysis of the chess-playing machine (Figure 4) shows that it makes use of intelligent input actions through one interface to deliver intelligent outputs through another interface. Though the machine M by itself is unintelligent, it is an open system that can augment its intelligence dynamically during execution. Clients of M like player A are unaware of the interactive help that M receives through interfaces unobservable to A. From A's point of view, M is like Van Kempelen's 17th-century chess machine whose magical

mastery of chess was due to a hidden human chess master concealed in an inner compartment.

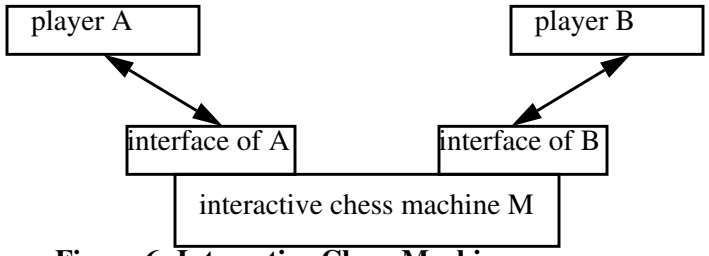


Figure 6: Interactive Chess Machine

Openness is a relative concept: the interactive chess machine is a closed system if both players are a part of the system, but the system consisting of any one player and the chess machine is open. The effect of the environment on an open system can be benevolent or malevolent: in the case of the chess machine the benevolent environment provides exactly the intelligence needed to respond, while in an airline reservation or planning system the environment can be the source of hurricanes or other disturbances that upset the applecart. The ability of hidden, uncontrollable events in the environment to affect system behavior corresponds to "action at a distance" in physical systems.

Simon's well-known example [Si] of the irregular behavior of an ant on a beach in finding its way home to an ant colony is another adaptation of interactive identity (see Figure 6). The behavior of the ant "echoes" the nonalgorithmic complexity of the beach, which acts as an oracle whose nonalgorithmic input is transformed by the ant into nonalgorithmic output.

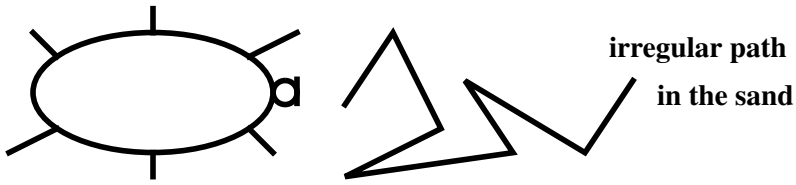


Figure 7: Simon's Ant on a Beach

Interactive identity machines have the behavioral complexity of their environment. The behavior of ants on beaches cannot be described by computable functions because the set of all possible beaches cannot be finitely described. The interactions over time of ants on beaches or of Eliza with potential "patients" are inherently not describable by algorithms because the set of all possible ways in which the environment can interact with the computer is not a recursively enumerable set.

1.3 The Design Space of Interaction Machines

To provide a framework for the analysis of properties of interaction machines, we populate the design space of interaction machines with some examples. Sequential interaction machines that read a sequence of inputs and transform them are one of the simplest kinds of interaction machines:

```
loop "sequential interaction machine"   -- single input stream, batch processing without feedback
  read(inmessage);
  outmessage = transform(inmessage);
  write(outmessage);
end loop;
```

The proof that sequential interaction machines are more powerful than Turing machines because they can replicate the behavior of oracles is more direct and intuitive than for interactive identity machines. But the proof that even pure interaction is more expressive than algorithms, which illustrates that interaction power does not require algorithm power, provides direct insight into the raw power of interaction. Pure

interaction without inner computing power is as expressive as a sequential interaction machine because it can harness and reuse computation power in the environment, but interaction machines that can also compute are less dependent on the environment in performing well-defined algorithmic tasks, just as an intelligent person needs less prompting from an expert.

When the input messages specify programs with their data and the transform function is an interpreter of the form "apply program to data" then a sequential interaction machine behaves like a batch processing operating system.

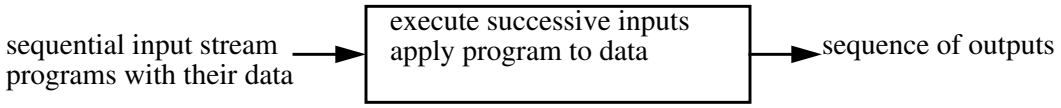


Figure 8: Batch Processing Interaction Machine

Sequential interaction machines are transducers that transform an input stream to an output stream. Algorithms that specify the transformation effect of transduction can be specified noninteractively, while the interaction history of a transducer in time cannot in general be specified algorithmically. Interaction histories determine a broader notion of observability than algorithmic abstraction, consisting of patterns of algorithmic episodes that are not in general algorithmically describable. Since interaction histories use algorithms as building blocks, interactive observability is a second-order semantics that uses observations of behavior of algorithms as a starting point for building more complex behavioral patterns. Transformational versus interactive semantics of transducers brings out very clearly the differences between the algorithmic and interactive paradigms of observability.

The design space of interaction machines may, as a first approximation be described in terms of the following dimensions:

multiplicity: single stream versus multiple stream
feedback: no feedback (batch processing) versus feedback through a state (object based)
time: time insensitive (relative time) versus time sensitive (absolute time and duration)
distribution: synchronous (lock-step) versus asynchronous (autonomous)
concurrency: sequential versus concurrent input actions

The sequential interaction machine represents the origin of a five-dimensional design space, choosing the simplest alternative in each dimension. The impact of multiple input streams, feedback, time, asynchrony, and concurrency in enhancing the behavior of sequential interaction machines will be considered.

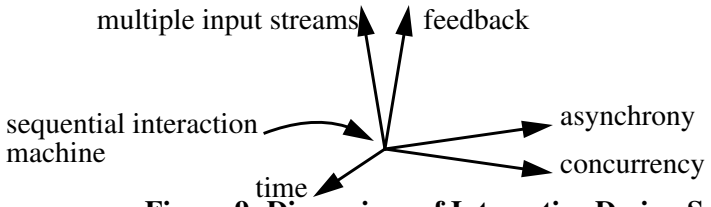


Figure 9: Dimensions of Interactive Design Space

Machines with multiple input streams have been extensively studied in sequential circuit theory. Circuit components like "and" gates are examples of sequential interaction machines with multiple input streams. A sequential multiple input steam machine that reads one input xi from each input stream and outputs one output yj onto each output stream at each computational step simply has read, transform, and output commands with multiple arguments:

loop "multiple input streams"
  read(x1, x2, ... , xk)
  value(y1, y2, ... , yl) = transform(x1, x2, ... , xk)

```
  print(y1, y2, ... , yl)
end loop
```

If the transform function is computable for each computational step, then we can define an extended transform function for finite sequences of M steps that is a computable function for any finite M. However, the interaction machine is the limit point of finite transform functions and is not a computable function.

Feedback can be added to sequential machines by simply including a local state that transmits history from earlier to later computational steps. Sequential machines with feedback have the following transform and newstate functions:

```
  value(y1, y2, ... , yl) = transform(state, x1, x2, ... , xk)
  nextstate = newstate(state, x1, x2, ... , xk)
```

Whereas multiplicity of input streams and feedback can be handled by existing mathematical models, at least in the synchronous case, time is a much more subtle notion. Turing machines cannot model external time though they have a well-developed notion of inner time. measured by the number of executed instructions. Interaction machines overcome this limitation of Turing machines: they can interact with ongoing external processes in real or simulated time. The notion of relative time can be expressed by sequences of ordered inputs called traces. Absolute time and duration in time can be expressed by time-stamped traces and a local time parameter whose current value records the time of occurrence of events. Interaction machines whose actions are entirely determined by relative time are called time insensitive, while machines whose actions depend on absolute time or duration are called time sensitive:

```
object time-sensitive-sequential-interaction-machine
  parameter time;  -- local clock or simulated time
  loop
    read(inmessage);
    outmessage := execute(inmessage, time);
    print(outmessage);
  end loop; end object;
```

Time sensitive interaction machines allow richer information to be passed across an interface than time insensitive machines: the absolute time at which messages are received and sent can be recorded. Time-sensitive machines specify a finer discrimination among inputs and outputs that gives rise to an entirely different technology and research and development community from that for time-insensitive machines (the real-time versus the simulation community).

The notion of external time depends on the nature of the client. If the client is a physical system (say a nuclear reactor) then external time is likely to be real time. However, objects have an event-driven notion of time in terms of their interface events. Interaction machines may have to cope with many notions of time associated with temporal rhythms of different clients.

1.4 Synchronous interaction machines

A synchronous interaction machine that transforms inputs from k input streams into outputs on l output streams has the following form:

```
object synchronous-interaction-machine
  local state: s := initial(s0);
  loop "multiple input and output streams"
    read(x1, x2, ... , xk)
    value(y1, y2, ... , yl) := transform(s, x1, x2, ... , xk))
```

```
  s := newstate(s, x1, x2, ... , xk)
  print(y1, y2, ... yl)
end loop
```

Synchronous machines are the interactive analog of sequential algorithms: they execute sequences of operations of an interactive execution engine, just as algorithms execute sequences of instructions. However, the sequence of instructions of an algorithm is determined by predefined data while the sequence of interactive operations of a synchronous interaction machine is determined by the environment. Inner computation for each interactive step can be concurrent, provided it is finite and deterministic. Multiple input streams are easily handled: they are the interactive analog of multiple arguments of a function.

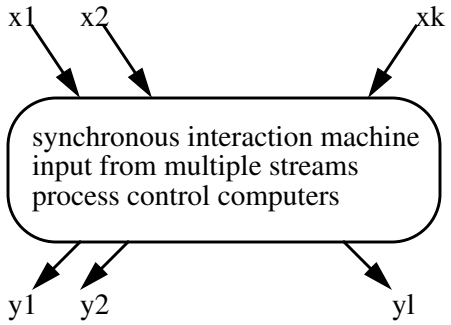


Figure 10: Synchronous (Lock-Step) Interaction with Multiple Streams

Synchronous interaction machines directly model "active" process control computers which proactively read an input from each of their sensors at successive time steps. Theoretical models of process control computers have been studied in [BWM], while concurrent synchronous interaction machine models (Esterel) has been studied in [BG]. Concurrent synchronous machines require no synchronization primitives and show that concurrency in the absence asynchrony that can be effectively modeled. Time sensitivity is likewise simple to model in synchronous computers with a global notion of time and becomes more difficult for asynchronous systems.

Synchronous interaction machines can be adapted (by introducing an idling step) to model traditional computers whose bursts of synchronous activity alternate with periods of inactivity. However, they cannot handle overload situations where multiple competing inputs arrive at the same instant of synchronous time or temporally overlapping nonatomic operations that compete for access to a shared resource. Their behavior is at least as rich as interactive identity machines and therefore richer than Turing machines. But they do not adequately model distributed software systems like airline reservation or banking systems.

1.5 Asynchronous input and output actions

An interaction machine is said to be *synchronous* if it controls the time of execution of its input actions and is *asynchronous* if the time at which messages arrive and are handled by input actions is externally determined and not under control of the machine. Asynchronous machines have an autonomous notion of time independent of that of agents with which they interact, while synchronous machines have a single global notion of time associated with all machines of the system.

This receiver-based notion of synchrony and asynchrony differs from that of synchronous and asynchronous procedure calls. Synchronous procedure calls require the sender to wait for a reply, while asynchronous procedure calls allow the sender to proceed once a message has been sent. Sender-based and receiver-based asynchrony express the relation between agents and the communication medium (ether) from the perspective of different kinds (roles) of observers. Receiver-based asynchrony expresses the temporal autonomy of each receiver, while sender-based asynchrony decouples sender execution from that of dispatched messages. The two forms of asynchrony are inequivalent, since the synchrony or asynchrony of input actions is independent of the synchrony or asynchrony of output actions. Synchronous message send-

ing is compatible with asynchronous receivers and vice versa.

Asynchronous interaction machines have input actions that can be specified by guarded commands expressible as "select" statements:

select(guard1:action1, guard2:action2, ... , guardN:actionN)
if no guard is enabled (true) then wait (block)
if a guard becomes enabled then execute the action of that guard
if several guards are enabled, then choose nondeterministically among the enabled actions

The dual notion of sender-based asynchrony can be specified by a "sendselect" statement whose output guards are enabled by eligibility of receivers. Sendselect statements are an extension of "fork" commands for multiple potential receivers:

sendselect(receiver1:message1,... , receiverk:messagek) --send to selected receivers, wait if none is enabled, send deterministically if a unique receiver is enabled, choose nondeterministically otherwise

The select and sendselect commands embody two kinds of nondeterminism referred to as input and output nondeterminism. Though input nondeterminism of input actions waiting for their inputs and output nondeterminism of senders in determining a receiver or a next action are two aspects of the same underlying phenomenon of asynchronous communication, they are modeled in entirely different ways. Object-oriented languages like Smalltalk and C++ have receiver-based input nondeterminism as their input mechanism but use synchronous procedure calling to a deterministic receiver as their output mechanism. Their receiver-based nondeterminism is sufficient to make the system nondeterministic and asynchronous even though the synchronous output mechanism would, in the absence of receiver-based asynchrony, be deterministic.

Sender-based output nondeterminism corresponds to that of classical nondeterministic automata with more than one next action and can be formalized algorithmically. Receiver-based input nondeterminism is the mechanism used by input actions of objects. It cannot be formalized algorithmically because it causes mechanisms to be interactive open systems and, presumably for this reason, has not been modeled by nondeterministic automata.

Sendselect commands can model broadcasting: the sender is viewed as a transmitter and the set of potential receivers is specified by a "wavelength" pattern. The case when all eligible receivers get the message is called "multicasting" while that when only a single receiver gets the message is called unicasting:

multicast(wavelength: message) -- send to all currently eligible receivers without blocking
unicast(wavelength: message) -- send message to a unique eligible receiver, block if none is eligible

Unicasting is the basic communication primitive of Milner's Pi calculus [Mil], which sends to a named channel, blocks if no receiver having a channel with the given name is enabled, and is nondeterministic when more than one receiver with the given channel name is enabled. It models mobile processes by allowing processes by allowing processes that come into the range of a transmitting sender to become dynamically eligible to receive messages.

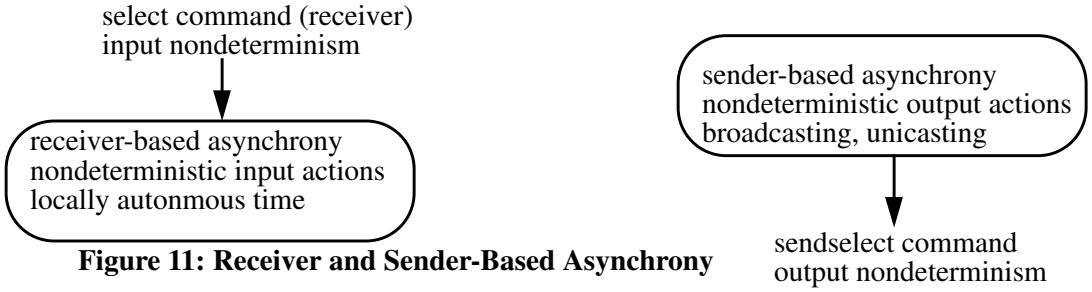


Figure 11: Receiver and Sender-Based Asynchrony

Nondeterministic output actions of unicasting mobile-process models such as the Pi calculus are further discussed in section 4.1.

1.6 Object machines

Objects are the most widely used class of interaction machine and the design space of object machines therefore deserves special study. Objects with a set of interface operations op1, op2, ... , opN have input actions governed by a special form of select statement with implicit input guards triggered by the arrival of messages:

select(op1, op2, ... , opN)

Operations of an object may be parameterized and are enabled by the arrival of a message whose operation (textual pattern) matches that of the operation. When objects have unique operation names and the select statement is accessed sequentially, the potential input nondeterminism cannot occur, since select statements never deal with more than one enabled input action at a time. Object machines have an interface that selects successive input operations in their order of arrival and a body with a hidden local state that provides feedback among successive operations:

object *object-machine*
interface
loop select (op1, op2, ... , opN)
end loop
body local state s; operation implementations that share and modify the state

Example: Bank account objects with deposit and withdraw operations are an example of time-sensitive object machines, since interest may be accrued on deposits. Interaction histories of deposit and withdraw operations need a time stamp to provide sufficient information for computing interest. Bank statements specify the observable behavior of bank accounts by time-stamped traces:

object *bank-account*
interface loop select
(procedure deposit (argument: Money);
procedure withdraw (argument: Money);**)**
end loop;
body local state balance**;** code of deposit and withdraw operations that share the balance

The time sensitivity of bank accounts can be conceptually modeled by an internal active process that accumulates interest even when the bank account is not interacting with external clients. Objects that execute internal processes while not interacting with external clients are called active objects. Systems of active objects are concurrent, since their internal processes execute concurrently with each other even when external activity observable by clients is sequential

Objects are modeled by relatively complex interaction machines that support feedback, time-sensitivity, and asynchrony. The most common kind of interaction machine occupies a relatively complex region of the interactive design space, supporting input nondeterminism to allow flexible scheduling of client needs and time sensitivity to handle interaction with the real world. Sharing of the state by interface operations is a form of inner interaction that complements external interaction by messages. The structural complexity needed to support these features is considerable.

An object may be viewed as a composite structure created by the composition of its operations and the shared state:

compose(s, op1, op2, ... , opN)

However, sharing of the state by the object's operations greatly complicates the composition operation, since the only way of sharing the hidden state is by hidden nonlocal variables that access the shared state from the operation implementation.

operation(parameters) body with hidden nonlocal variables

The composition of operations with their shared state is not "compositional" in the sense that the composite behavior cannot be simply expressed in terms of the behavior of components. Operations of the composite object cannot be modeled as algorithms whose output behavior is uniquely determined by its input parameters. Nonlocal variables may cause the behavior of operations to be changed in an uncontrollable way, so that object behavior cannot be expressed by treating the operations as algorithms. The behavior of an object can be expressed by axiomatic relations among operations like "deposit(x) followed by withdraw(x) has no effect", but individual operations cannot be expressed as algorithms because their nonlocal variables are at the mercy of unpredictable interactive influences.

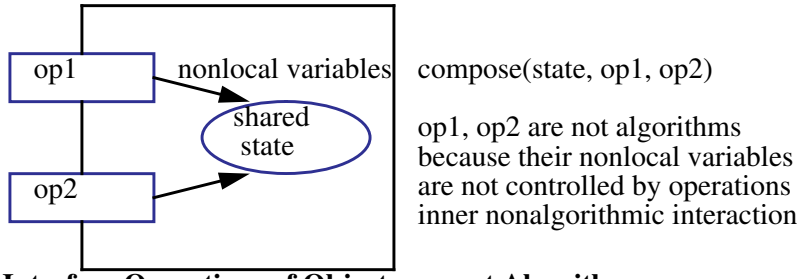


Figure 13: Interface Operations of Objects are not Algorithms

The effect of the operation op1 depends on a hidden shared state that can change its value unpredictably between successive sequential executions of op1. In concurrent systems the shared state can be changed by op2 during the execution of op1 and the behavior of operations can be chaotic (see section 1.7 below).

The nonalgorithmicity of an object's operations indicates that object-based programs cannot be analyzed algorithmically even for small programs. The object-oriented programming paradigm is therefore unsuitable for expressing algorithmic behavior: it focuses on the interaction of algorithms rather than on their noninteractive behavior, suggesting that courses on algorithms should not be taught using object-oriented implementation techniques, since the sharing of data structures by collections of operations permits the operations to interact in nonalgorithmic ways. Object-oriented implementation violates the "divide and conquer" premiss on which algorithm analysis is based. Algorithm analysis requires that algorithms are treated as isolated (closed) systems not subject to any interactive influences during their execution.

The insights of algorithm analysis will continue to be very valuable, and courses on algorithms will continue to occupy an important place in computer science curricula. But the role of algorithmic models (Turing machines) in computing will become less central because of the increasing importance of interactive applications. Algorithms should be viewed as idealized abstractions rather than as complete specifications of the behavior of computing systems. Since algorithmic description breaks down even for single object, multiparadigm models of logic and object-based programming are seen to be very brittle. The object-based interactive paradigm appears to be incompatible with logic-based, functional, or Turing machine paradigms.

Object machines for sequential languages perform synchronization at the object interface and complete each operation before the next one commences. The abstraction interface for purposes of information hiding and the synchronization interface controlling the synchronization of arriving messages are generally the same for sequential object-oriented systems. This identity of abstraction and synchronization interfaces cannot, however, be taken for granted, since it breaks down for multi-object components with internal synchronization where the abstraction boundary is coarser than the synchronization boundary. Objects that violate abstraction by allowing clients with special privileges to peek inside have a synchronization bound-

ary coarser than the abstraction boundary.

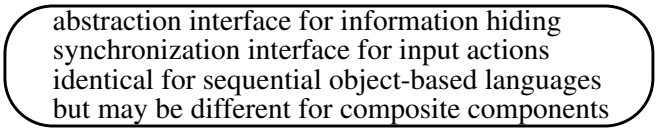


Figure 12: Abstraction and Synchronization Boundaries of Software Components

Monitor machines relax the requirement of synchronization at the object interface, performing synchronization by wait and signal operations that suspend and reactivate threads during execution at the time of access to shared resources rather than at the time of entry to the agent. Thus a deposit of a bank account implemented as a monitor would be synchronized at the time of access to the shared balance rather than on entry to the bank account. Monitors demonstrate that asynchrony does not inherently require blocking input actions, since blocking needed for synchronization can be accomplished by wait and signal blocking primitives within the monitor independently of input actions. Their synchronization boundary is that of shared data structures, which is finer than that of objects which manage the shared data structures. Monitor-style control simulates concurrent processing of arriving threads of control by dynamic interleaving within the monitor based on availability of critical resources. This style of simulated concurrency is sometimes referred to as quasi-concurrent processing and is closely related to coroutine models of execution.

1.7 Concurrent input actions

Concurrent interaction machines can process inputs concurrently: machines that are internally concurrent but process their inputs sequentially are not included, since their observed interactions are sequential. For example, systems of active objects that process inputs sequentially are not considered concurrent interaction machines because observable interface behavior is sequential in spite of the local concurrent computation within each active object.

Concurrent interaction can be realized either by multiple input streams or by a single input stream with overlapped execution of stream elements. Actor machines [Ag] are an inherently asynchronous concurrent interaction machines that allow concurrent interaction between successive messages of a single input stream. An actor has a mail address, an associated mailbox that holds a queue of incoming messages in arrival order, and a behavior that may at each computational step read the next mailbox message and execute it. Actors support concurrent overlapping and interference among sequentially handled messages:

object *actor-machine*
 loop "single input stream with temporally overlapping execution"
  read next message in input queue
  perform action concurrently with already executing messages
  actions may create new actors, send messages to existing actors, or become a next state
 end loop

Execution of a message can initiate a sequence of actions that overlap with the execution of earlier and later messages. Create, send, and become actions reflect the interactions that an actor may initiate at each computational step.

Concurrent interaction machines that handle multiple asynchronous input streams is illustrated by automatic teller machine (ATM) services of banking systems:

Example (overlapping withdrawals from joint bank account): A banking system with multiple ATMs that allows concurrent withdrawals from joint bank accounts is a concurrent interaction machine. Consider the effect of two autonomous requests to withdraw $1 million from a joint account containing $1.5 million. If withdrawal involves juggling an investment portfolio and the operations overlap in time, the unsuccess-

ful operation can have disastrous side-effects. Such problems can be avoided by transaction protocols, but the set of all possible behaviors of ATM systems includes the case when transaction protocols are absent or imperfect. In the real world, large financial transactions may have unpredictable ripple effects.

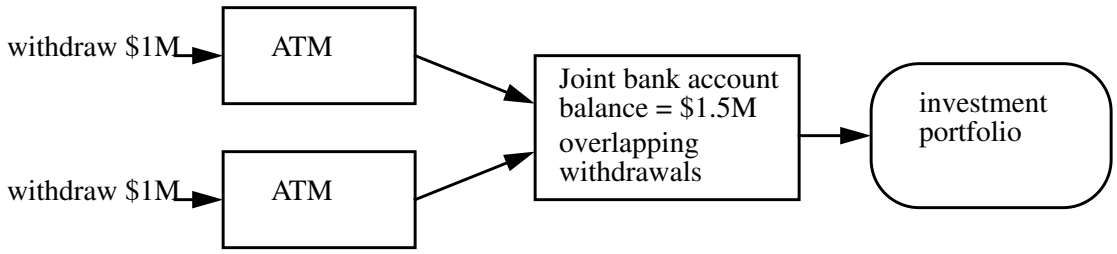


Figure 14: ATM System as an Asynchronous Interaction Machine

The effect of concurrent nonatomic operations can be abstractly expressed by considering an object with two procedures *P1* and *P2* sharing an internal state, as in Figure 6. Assume that *P1* receives a message to perform a nonatomic operation and that *P2* concurrently changes the state being used by *P1*. The behavior of such an object cannot be described by a trace of its interface operations and is *nonserializable*. Interleaved execution of nonatomic procedures (for example in a monitor) causes nonserializable behavior due to interleaving even without actual concurrency. Asynchronous nonatomicity rather than physical concurrency is the cause of nonserializable behavior.

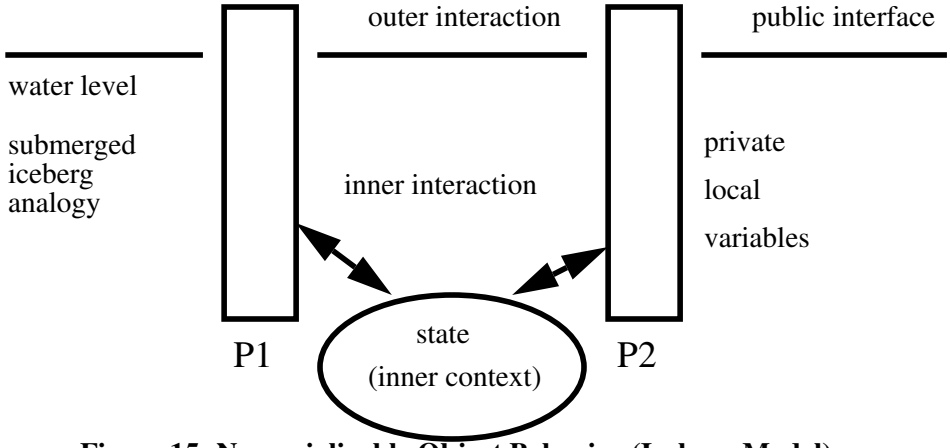


Figure 15: Nonserializable Object Behavior (Iceberg Model)

Definition: A computation of an interaction machine (interaction automaton) is said to be *nonserializable* if its effect cannot be described by a sequence of the object's interface operations (input actions).

Asynchronous interaction machines impose an additional layer of abstraction that hides the implementation of nonatomic interface procedures sharing inner resources. The nonserializable behavior in Figure 5 could in principle be "explained" by serializable behavior of interleaved lower-level primitive instructions. But models at a given level of abstraction should not involve explanation at a lower level, just as the rules of cognitive explanation exclude explanation in terms of atomic physics. The level of abstraction constrains the level of explanation to observable phenomena at a given level of abstraction. Explanations in terms of inherently unobservable inner distinctions are "metaphysical".

Theorem: The behavior of asynchronous interaction machines cannot be expressed by synchronous interaction machines.

Proof: Asynchronous interaction machines are more expressive than synchronous machines either by showing that a) they can model richer external behavior or b) their stimulus-response behavior cannot be modeled by synchronous machines.

a) Asynchronous input actions can model external behavior of temporally autonomous agents that cannot be modeled by synchronous machines. b) Nondeterministic and nonserializable behavior of asynchronous interaction machines cannot be expressed by synchronous interaction machines.

Nonserializability is due to unobservable and uncontrollable sensitivity to the order of access to a shared resource (the object's state). This erratic behavior of objects is analogous to chaos in physics, which is also due to sensitivity of access to shared resources. The pattern of access to shared resources in Figure 15 arises in the well-known demonstration of chaotic behavior of Figure 16, which illustrates a two-dimensional pendulum (steel ball) in the presence of two magnets. The magnets M1, M2 correspond to the procedures P1, P2, the steel ball corresponds to the state. The magnets exert two streams of impulses on the steel ball just as the procedures 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.

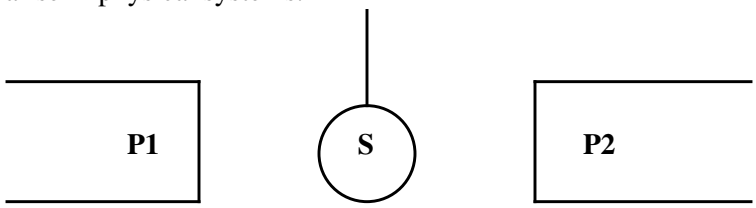


Figure 16: Chaotic Two-Dimensional Motion of Steel Pendulum in a Magnetic Field

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 in the force field of one of the magnets but nonlinear in regions where the force fields cancel each other. Though interaction machines are discrete systems not modeled by differential equations, their chaotic behavior for nondeterministic access to shared resources corresponds to a form of nonlinearity. Interface operations of an object do not control the inner operations executed on the shared object state, and interference between inner operations of two temporally overlapping interface operations causes nonserializable, chaotic behavior.

Though asynchronous interaction machines are simply definable by adding asynchronous input actions to Turing machines or asynchronous binding to the lambda calculus, they can give rise to very complex effects, reflecting the complexity of heterogeneous distributed applications. Asynchronous machines illustrate even more strongly than synchronous machines that mechanisms are more powerful than formalisms in defining behavior:

object *asynchronous-interaction-machine* (asynchronous inputs and multiple external notions of time);
  multiple loosely-coupled interfaces that share data and may cause unanticipated "action at a distance"
  no global notion of time or of observers and observability
  concurrent, potentially overlapping arriving messages
  concurrent asynchronous inner computation
  undisciplined sharing of local memory by nonatomic interface operations
end object

1.8 Concurrency, distribution, and interaction

Concurrency and distribution are logically distinct notions: concurrency involves simultaneity in time while distribution involves separation in space. A shared memory multiprocessor is concurrent but not distributed, while a sequential object-based system is distributed but not concurrent. Interaction, which

involves distribution of inputs over time, is distinct from both concurrency and distribution in space. A distributed concurrent algorithm for shortest paths on a graph of the traveling salesman problem is concurrent and distributed but not interactive. The three independent dimensions of concurrency, distribution, and interaction may be characterized as follows:

concurrent computations are simultaneous or overlapping in time
distributed computations are separated in space and execute autonomously (asynchronously) in time
interactive computations have dynamic inputs that are separated (distributed) in time

Noninteractive concurrency and distribution can be handled within the Turing machine paradigm: it is interaction that is more expressive by permitting models of real time and open systems. Textbooks on concurrent algorithms examine the reduction of complexity achievable by concurrency but pay little attention to interaction. Distributed systems can be analyzed algorithmically if no interaction is permitted once the computation is initiated. The recognition that interaction rather than concurrency or distribution is the basis for greater expressiveness is a fundamental insight.

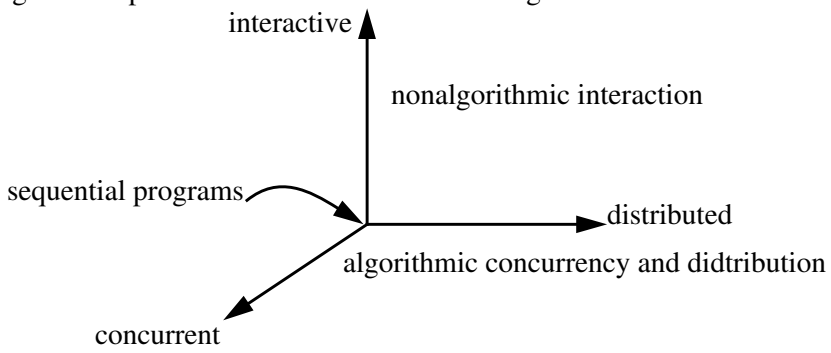


Figure 17: Design Space of Concurrent, Distributed, Interactive Computation

Milner defines concurrent composition "P|Q" of two processes P and Q as *"P and Q acting side-by-side, interacting in whatever way we have designed them to interact*". This definition recognizes that interaction rather than the concurrency causes nondeterministic behavior. The composition P|Q of autonomously executing processes P and Q is interactive: P and Q are necessarily interactive processes that are not necessarily concurrent.

Sequential programs which share resources fail to preserve their equivalence (compositionality) under concurrent composition. As shown by Milner, the procedures "P1: x:=1; x:=x+1;" and "P2: x:=2;" are equivalent as sequential programs but concurrent composition with "P3: X:=3;", yields nonequivalent composite processes P1|P3 and P2|P3with different possible values generated by interleaved execution:

P1|P3 can yield the values 2, 3, or 4, while P2|P3 can yield only 2 or 3

This simple example clearly illustrates that nonequivalent behavior is due to interaction, which causes a breakdown of functionality because of sensitivity to the order of access to shared information. Compositionality breaks down in this example for precisely the same reasons as in the joint bank-account example.

The analysis of interaction independently of concurrency is fundamental to this work, suggesting new ways of analyzing process models and as well as other models of concurrent programming. For example, the analysis of transaction correctness in section 4.2 suggests that serializability should be viewed as a condition for interaction control rather than concurrency control.

1.9 Layers of protective abstraction

Turing machines have been the dominant computational abstraction during the first 50 years of computer science, serving as a basis for modeling algorithms, computable functions, and computational complexity. However, their role in the next 50 years may be less central because they are not powerful enough to capture the interactive behavior over time of objects, software systems, and distributed computation. They wrap physical computers in a confining layer of protective abstraction that realizes mathematical

tractability by sacrificing the ability to model dynamic interaction, external time, and persistence.

Interaction machines wrap computers in a less confining layer of abstraction by extending Turing machines with input actions (read statements) that model relations over time between systems and their observers. Once interaction is admitted as legitimately observable, computing mechanisms can be classified by their ability to model external behavior. Asynchronous interaction machines (with asynchronous interaction protocols) have richer behavior than synchronous machines because they can model richer external environments, including the interaction of temporally autonomous distributed components with no global notion of time. Thus there are at least two distinct layers of interactive behavior richer than Turing machines.

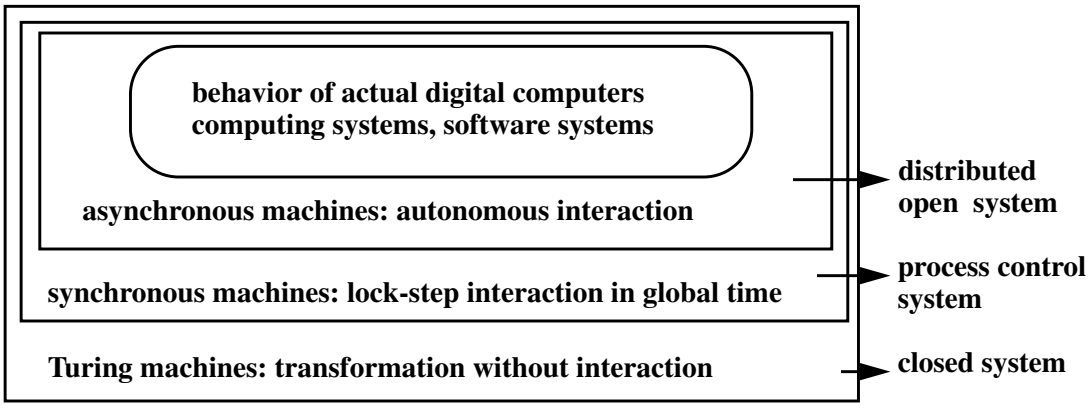


Figure 18: Layers of Computational Abstraction

Chomsky classified the expressive power of machines (automata) weaker than the computable functions. He identified four classes of computing mechanisms (finite, pushdown, linear bounded, and Turing automata) and defined their behavior in terms of four progressively more powerful formal grammars (regular, context-free, context sensitive, unrestricted). Synchronous and asynchronous machines can be simply defined by extending computing mechanisms of the Chomsky hierarchy with input actions, but the classes of observable behaviors of interaction machines have no corresponding formal behavior specifications.

Observable Behavior		**Computing Mechanisms**
regular sets	weaker than comp functions	finite automata
context-free languages		pushdown automata
context-sensitive languages		linear-bounded automata
computable functions		Turing machines
synchronous machine behavior	stronger than comp functions	synchronous interaction machines
asynchronous machine behavior		asynchronous interaction machines

Figure 19: Extended Chomsky Hierarchy of Behaviors and Associated Mechanisms

The idea that machines have formally specified behaviors, which is a cornerstone of the theory of computing, is a casualty of the extension to interaction machines. Figure 19 suggests that machines are a more powerful way of specifying behavior than formal specifications, since they can be extended to describe interactive behavior while formal specifications cannot. First-order logic cannot be extended to meet the greater expressiveness of interaction machines. Open specifications by interaction machine are more expressive than closed specifications by first-order logic. Turing machines merely capture the most powerful formally specifiable closed behavior but not, as Church and Turing conjectured, the most expressive intuitive notion of computability. Interaction machines extend the Chomsky hierarchy of machines beyond Turing machines to observably richer intuitive forms of computing whose behavior cannot be expressed by a corresponding extension of a specification formalism. Moreover, there is a hierarchy of machines more powerful than Turing machines classified by their ability to handle forms of external interaction [We2], including at least synchronous and asynchronous interaction machines.

2. Paradigms of Modeling

Modeling is an interdisciplinary notion introduced by the Greeks and used in physics and engineering long before the birth of computer science. Models are abstractions of an application domain (domain of discourse) that ignore properties judged to be irrelevant to simplify the presentation of properties judged to be relevant. A general framework for modeling that characterizes algorithmic and interactive models of computation as well as models in other disciplines is illustrated by Figure 20.

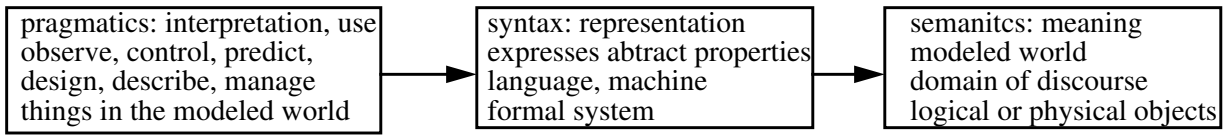


Figure 20: Interdisciplinary Framework for Modeling

A model captures properties of a semantic world by a syntactic representation that may be used to understand, predict, or control its properties. Algorithmic models can be completely described and validated by logic while interactive models can be only partially described. Interactive models are shown to be robust in that many alternative syntactic representations appear to have the same expressive power, just as computable functions can be expressed by Turing machines, the lambda calculus, and many other representations.

The computing literature has focused primarily on syntactic and semantic aspects of modeling. Interactive models shift the focus to pragmatic questions of observation and use, being concerned with the presentation of a given modeled world from the viewpoint of different clients or observers. The pragmatic perspective is goal-oriented, allowing us to examine the purposes for which a model was developed as a part of the study of the model itself. For example, in section 5, life-cycle models are distinguished from system-structure models by the pragmatic differences between their clients (human software developers and computers that execute developed models).

Since interaction machines are not formally describable it is important to show that their behavior is can nevertheless be described and used for practical purposes. The role of harness constraints in taming the power of interaction machines and describing their behavior is examined. Harness constraints include both interface constraints and constraints on behavior that cannot be entirely described by interfaces. Interface descriptions are the pragmatic mechanism used by software designers and application programmers for partially specifying systems for the purpose of controlling, predicting, and understanding their behavior. Harnesses are an interactive modeling mechanism that abstractly captures the pragmatic description of properties of interactive systems by their interfaces.

Section 2.5 examines the formal arguments for proving that interaction machines are more expressive than Turing machines. Here we rely primarily on Turing's results about oracles and the adaptation of Godel's incompleteness result for interaction machines. The formal arguments are presented late because our primary concern is that of establishing an understanding of the intuition underlying interaction machines. Formal arguments by themselves could be ignored, but the weight of intuitive evidence from software engineering, artificial intelligence, and distributed systems make the formal arguments more persuasive.

The notion of algorithmic complexity and reducibility is reviewed to understand its role as an idealized model of noninteractive behavior in the more expressive interactive world. Biological models of DNA-based computation are shown to fit the interactive model of computing and to support forms of communication similar to process models of the Pi calculus. Interactive reflection is seen to be more powerful than algorithmic reflection: such *lifting* of concepts from the domain of algorithms to the more general setting of interaction changes their meaning, as demonstrated in section 3 when generalizing the notions of declarativenness and types from algorithmic to interactive contexts.

2.1 What is a model?

Models express selective properties of a domain of discourse for the purpose of understanding, predicting, or controlling its behavior. They abstract from behavior judged to be inessential to focus on behavior judged to be important. The natural sciences and engineering generally model some aspect of the real world: models of the solar system express the motions of the planets, while models of airplanes aim to predict aerodynamic properties by differential equations, physical simulation, or both.

Computational models express properties of modeled domains by the behavior of programs. If M is a computational model of a modeled world W, then we can learn about W by executing programs of M. Computer science has different styles of modeling computational behavior, just as the physical sciences have different styles of modeling physical behavior in physics, chemistry, and biology. Three principal styles (paradigms) of computational modeling may be distinguished:

declarative modeling paradigm: aims to specify what is computed independently of how it is computed
imperative modeling paradigm: specifies computation operationally by an abstract computer model
interactive modeling paradigm: specifies interaction of an agent over time with its environment

Declarative models are mathematical, imperative models have an algorithmic flavor, while interactive models are empirical. The evolution of models of computation from declarative to imperative and then interactive models has occurred in many subfields of computer science, for example in programming languages, software engineering, and artificial intelligence:

declarative (mathematical) models -> imperative (algorithmic) models -> interactive (empirical) models

Declarative models include set theory, first-order logic, the lambda calculus, and denotational semantic models of programming languages. Imperative models include Turing machines, algorithms, and operational semantic models of programming languages. Interactive models include object-oriented and process models, reactive systems, embedded software engineering models, and agent-oriented artificial intelligence models. Though the differences between declarative and imperative models are considerable, they are dwarfed by the greater gap between algorithmic and interactive models. Declarative and imperative models were shown by Turing to have the same expressive power, while interactive models have greater expressive power. The following notion of "model" is broad enough to include declarative, imperative, and interactive models:

Definition: A model *M = (R, W, I)* has a *syntactic component R*, a *semantic component W*, and a *pragmatic component I* that maps representations into their behavioral (computational) semantics. A *model* M is a *representation* R of a *modeled world* W and an *interpreter* I that captures properties of elements w in W by interpretations I(r) of representations r of R.

Model M = (R, W, I)	Syntax Representation R	Semantics Modeled world W	Pragmatics Interpretation I
Declarative models:			
first-order logic	well-formed formulae	model theory domain	rules of inference
lambda calculus	lambda expressions	computable functions	reduction rules
Imperative models:			
Turing machines	states, transitions	computable functions	transition rules
procedure paradigm	bnf specification	pre and post conditions	interpreter
Interactive models:			
distributed systems	system specification	interacting computers	interaction protocols
software systems	design specification	software applications	use cases, traces
AI agent models	representation language	intelligent agents	behavior spec
dynamical systems	differential equations	real world systems	closed or open soln

Figure 21: Declarative, Imperative, and Interactive Models

Imperative models support history-sensitive variables that allow sharing of resources and memory to be more directly modeled than in purely imperative models. Much of the research on programming language semantics and software models has focussed on expressing imperative behavior by declarative models. But

the imperative modeled world is no richer than that of declarative models: W is an abstract world of functions and predicates over a domain of values. From the point of view of expressiveness, declarative and imperative models are equivalent, being expressed by the modeled world of mathematical model theory. Since expressiveness is the single most important criterion for classifying models, models may be classified into two major categories:

mathematical (algorithmic) models: modeled world of computable functions

empirical (interactive) models: modeled world of eternal environment (empirical phenomena)

The robust equivalence of mathematical models for alternative mathematical formalisms provided an important foundation for theoretical computer science, but was referred to as the "Turing tarpit" by researchers who felt frustrated by the apparent inability of formalisms to express richer behavior. Interaction machines provide an escape from the Turing tarpit into the richer but equally robust world of empirical models by simply extending Turing machines with input actions. The tarpit is seen in retrospect to have been subjectively created by imposing the rule that seeking an outside helping hand from agents in the environment was against the rules. The robustness of empirical models has been historically validated by the evolution of the natural sciences but the intuitive notion of computation was thought to be nonempirical because of the influence of the Church-Turing view. Interaction machines capture properties like observability, incompleteness, action at a distance, and chaos that arise in physical models and conclusively show that models of computation can be included among empirical models. They provide a precise definition of empirical computer science as the study of interactive computational models.

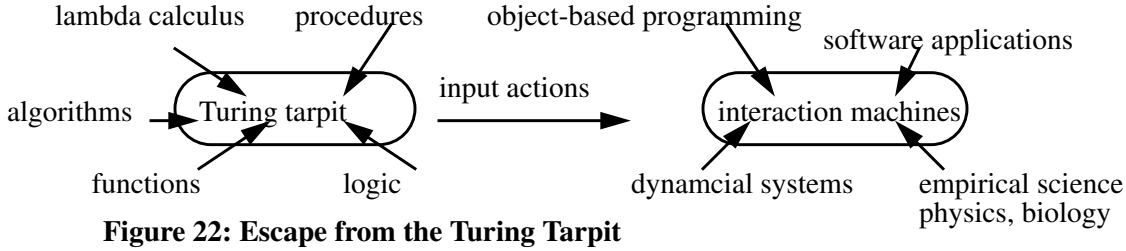


Figure 22: Escape from the Turing Tarpit

The role of incompleteness in increasing the expressive power of models can be illustrated by considering the relation between representations R and modeled worlds W. Soundness of a model means that the representation R expresses correct behavior in W, while completeness means that R expresses all possible behavior in W. Though soundness and completeness is formally desirable for guaranteeing that all behavior can be modeled, it requires W to be isomorphic to R and therefore no more expressive than R. Incompleteness permits W to include behavior not expressible by R and therefore to be richer than R. Interactive models can represent modeled worlds richer than those expressed by their representations R by allowing R to be an incomplete representation of W. Incompleteness is a necessary feature of systems that talk about a world richer than that expressible by a syntactic formalism and is a key to empirical models of worlds whose semantic properties transcend their syntactic representations. Interaction machines allow computational models to transcend the modeled world of purely mathematical models and express richer incomplete modeled worlds of empirical phenomena.

The distinction between mathematical and empirical models can be most directly expressed by characterizing the difference between their domains. Mathematical domains can in principle be denotationally described by a set of fixed denotations, while empirical domains have no fixed denotational description, since they evolve dynamically by knowledge acquisition through observation (interaction). Whereas there is just a single mathematical domain common to all mathematical models, each interaction machine has its own evolving domain of discourse. Mathematical derivations and algorithmic computations are denotation-preserving transformations of a closed world of syntactic representations, while interactive computations violate the closed-world assumption and introduce new knowledge and new denotations during computation.

2.2 Models and their validation

Models may be classified *syntactically* by the kinds of representations that they use, *semantically* by the nature of the real or constructed world they are trying to model, and *pragmatically* by the relation between the model and its interpreters and users. Syntax is specified in different ways by formal systems, automata, programming languages, object models, knowledge representation formalisms, etc. Semantics is similar for declarative and imperative models but richer for interactive models: declarative and imperative semantics is about unchanging mathematical worlds of sets and functions, while interactive semantics is about a changing real world whose properties are progressively revealed by observation (interaction).

Pragmatics differs for declarative and imperative models without affecting expressive power: interpreters for functional and logic programming languages have a characteristic structure that differs from that of interpreters for procedure-oriented languages. The pragmatics of interactive programs differs from that of declarative and imperative programs in both expressive power and validation techniques. Validation is a pragmatic notion that cannot be expressed by pre and post conditions in interactive models. The meaning of declarative and imperative models is statically specified, while interactive models have an autonomous meaning determined by the environment that must be discovered and can never be completely known.

Logic distinguishes between *proof theory* that deals with the syntactic properties of proofs and programs and *model theory* that deals with associated modeled worlds. Formal models are validated by criteria of soundness and completeness that establish relations between proof theory and model theory. Soundness implies that proofs (computations) capture properties of the modeled world while soundness and completeness together imply an isomorphism between the syntactic notation and the modeled world. Soundness is a local property definable by induction on syntactic derivations (computations), while completeness is an elusive global property defined by a mapping from modeled entities onto syntactic terms. Completeness is a desirable formal feature, but limits the power of the model theory to model properties that are not statically expressed by the proof theory. Soundness without completeness guarantees that proofs (computations) express valid behavior but liberates the model theory from its dependence on proof theory, allowing meaning to be dynamically determined by interaction. Incompleteness is a two-edged property: it has negative connotations because it expresses limitations of a formalism in capturing desired behavior, but is positive in allowing modeled worlds to be open and only partially describable.

Logics are syntactically described by a set of well-formed formulae WFF and by axioms and rules of inference that determine a subset Pr of provable formulae. The set Tr of formulae true in all interpretations is also a subset of well formed formulae. Soundness and completeness may be expressed as relations between the subsets of provable and true formulae. Soundness implies that the set of provable formulae is a subset of the set of true formulae, completeness implies that the true formulae are a subset of the provable formulae, and soundness and completeness together implies that the true and provable formulae are coextensive.

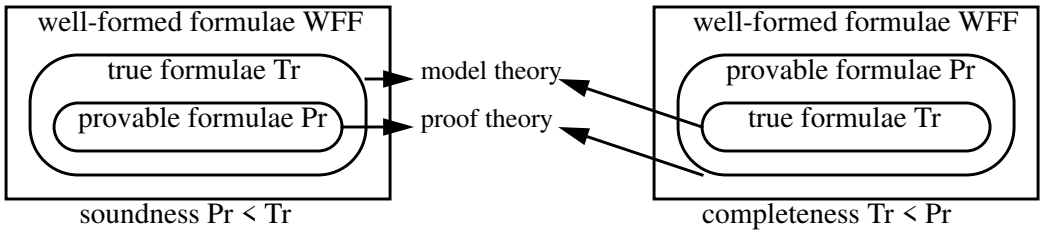


Figure 23: Soundness and Completeness as Relations among Sets of Formulae

The notion of observability in physics corresponds (is dual) to that of interaction in computing: observation describes the external view of an observer while interaction describes the internal view of the object being observed. Empirical models in physics are validated by their externally observable behavior because internal structure is inaccessible, while empirical models in computer science are validated by their interactive behavior because the goal is to express observable behavior in terms of the internal structure.

Interaction machines have a richer syntactic behavior (proof theory) than Turing machines and a corre-

spondingly richer set of worlds that can be modeled, including the real world. The concept of soundness is transferrable from the world of logic to that of interaction machines, translating into correctness of interface behavior. Completeness, which is more difficult to handle than soundness even for formal models, becomes undefinable for interaction machines because empirical modeled worlds whose behavior is specified by interaction machines cannot be formally specified. Tools of logic and mathematics provide a "complete" description of functions and algorithms but do not scale up to provide a similarly complete analysis of software systems.

Empirical models can guarantee only the existence of correct behavior (type 1 correctness) but not the nonexistence of incorrect behavior (type 2 correctness). The view of programs as contractual predicates that guarantee output (postconditions) for given inputs (preconditions) must be replaced by a weaker notion of programs as incompletely specifiable reactive (interactive) processes. The development of a framework for understanding and validating empirical models of computing that parallels the maturity of empirical models in the physical sciences is a primary challenge of computer science.

2.3 Robustness of interactive models

The robustness of Turing machines in expressing the behavior of algorithmic, functional, and logic languages has provided a basis for the development of a theory of computation as an extension of mathematics. Interaction machines are an equally robust model of interactive computation. Each concept on the left-hand side of figure 24 has greater expressive power than the corresponding concept on the right-hand side. Moreover, it is conjectured that left-hand-side concepts have equivalent expressive power captured by a *universal interaction machine*, just as right-hand-side concepts have equivalent expressive power of a Turing machine *(universal algorithm machine).*

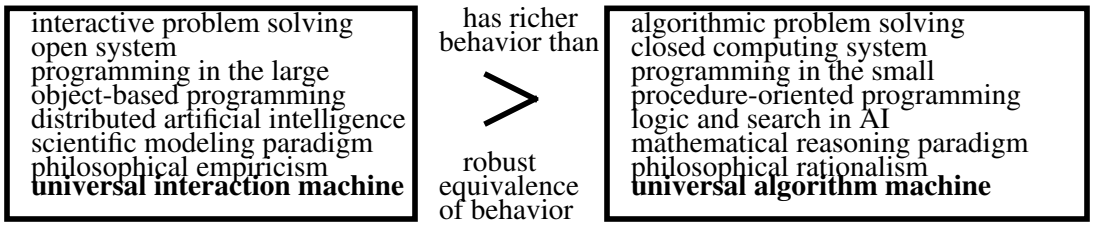


Figure 24: Parallel Robustness of Interaction and Turing Machines

Interaction machines provide a common modeling framework for left-hand-side concepts just as Turing machines provide a common framework for right-hand-side concepts. The equivalence of the first five right-hand-side entries is the basis for the robustness of Turing machines. The corresponding left-hand-side entries are less familiar, and there has not until recently been a crisp notion of expressive power for interactive problem solving, open systems, programming in the large, object-oriented programming, or distributed artificial intelligence. Interaction machines provide a crisp model for these concepts and allow the equivalence of behavior and of interactive problem-solving power to be expressed as computability by a suitably-defined universal interaction machine.

The somewhat fuzzy notion of programming in the large can be precisely defined as interactive programming. Large entirely algorithmic programs of one million instructions do not qualify, while modest interactive systems with a few thousand instructions do. The identification of programming in the large with interactive programming implies that programming in the large is inherently nonalgorithmic, supporting the intuition of Fred Brooks that programming in the large is inherently complex. Interaction machines provide a precise way of characterizing fuzzy concepts like programming in the large and empirical computer science and elevate the study of models for objects and software systems to a first-class status independent of the study of algorithms.

The role of interaction machines in expressing the behavior of scientific models and empiricism is less direct than their role in expressing open systems, object-based programming, and distributed artificial

intelligence. The identification of interactive problem solving with the scientific (empirical) modeling paradigm follows from the correspondence between interaction and observability (interaction from the point of view of an agent or server corresponds to observability by an external client).

Interaction machines express processes of observation and capture the intuitive notion of empiricism. Moreover, the logical incompleteness of interaction machines corresponds to descriptive incompleteness of empirical models. Modeling by partial description of interface behaviors is normal in the physical sciences. The incompleteness of physical models is forcefully described by Plato in his parable of the cave, which asserts that humans are like dwellers in a cave that can observe only the shadows of reality on the walls of their cave but not the actual objects in the outside world.

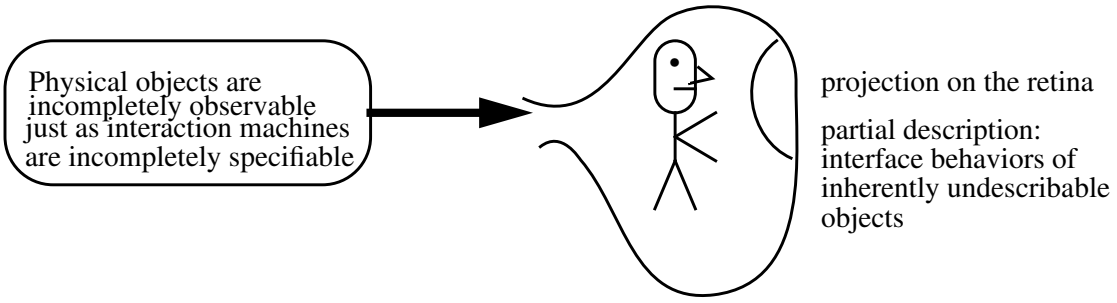


Figure 25: Plato's Cave as a Paradigm for Describing Incomplete Behavior

Plato's pessimistic picture of empirical observation caused him to deny the validity of physical models and was responsible for the eclipse of empiricism for 2000 years. Modern empirical science is based on the realization that partial descriptions (shadows) are sufficient for controlling, predicting, and understanding the objects that shadows represent. The representation of physical phenomena by differential equations allows us to control, predict, and even understand the phenomena represented without requiring a more complete description of the phenomena. Similarly, computing systems can be designed, controlled, and understood by interfaces that specify their desired behavior without completely accounting for or describing their inner structure or all possible behavior.

Turing machine models of computers correspond to Platonic ideals in focusing on mathematical tractability at the expense of modeling accuracy. To realize logical completeness, they sacrifice the ability to model external interaction and real time. The extension from Turing to interaction machines, and of procedure-oriented to object-based programming, is the computational analog of the liberation of the natural sciences from the Platonic worldview and the development of empirical science.

The correspondence of closed systems and philosophical rationalism follows from Descartes' characterization of rationalism by "Cogito ergo sum", which asserts that noninteractive thinking is the basis for existence and knowledge of the world. Interaction corresponds precisely to allowing internal computations (thinking processes) of agents (human or otherwise) to be influenced by observations of an external environment. The correspondence between rationalism and empiricism and algorithmic and interactive computation is thus quite direct. The demonstration that interaction machines have richer behavior than Turing machines implies that empirical models are richer than rationalist models. Fuzzy questions about the relation between rationalism and empiricism can be crisply formulated and settled by expressing them in terms of computational models.

The equivalent expressive power of imperative (Turing machine) and declarative (first-order logic) models lends legitimacy to computer science as a robust body of phenomena with many equivalent models, including not only Turing machines but also the predicate and lambda calculus. Church's thesis expresses the prevailing view of 20th century formalists that the intuitive notion of computing is coextensive with the formal notion of functions computable by Turing machines. However, Church's thesis is a rationalist illusion, since Turing machines are closed systems that shut out the world during the process of computation and can very simply be extended to a richer intuitive notion of open, interactive computation that more accurately expresses the interactive behavior of actual computers.

Declarative and imperative systems compute results from axioms, arguments or initial inputs by rules of inference, reduction rules, or instructions. The interactive paradigm extends the declarative and imperative paradigms by allowing initial conditions distributed over time. This extension is analogous to extending differential equation techniques from one-point to distributed boundary conditions. In a distributed interactive system we further extend initial conditions to distribution over both space and time. Distribution of initial conditions over space is familiar from multihead Turing machines and does not by itself increase expressive power. However, distribution over time increases expressive power:

declarative paradigm: initial conditions (axioms + theorem) + rules of inference -> (yes + proof) or no
imperative paradigm: initial value (precondition) and program yields result (postcondition)
interactive paradigm: initial conditions distributed over space and time, imperative or declarative rules

This analysis suggests classifying paradigms by the distribution of external interactions over space and time. Differences among inner rules of computation have no effect on expressive power, while extension from one-point to distributed interaction over time increases expressive power. Interaction machines derive their power from their ability to harness the interactive power of the environment. Their expressive power is comparable to that of managers who need not have inner ability to solve a problem since they can harness the problem-solving power of their employees.

2.4 Harness constraints on interaction

Raw interactive power can be tamed by imposing constraints on the forms of permitted interaction. For example, a personal computer constrains interaction by permitting only serial execution of commands in displayed windows. Software engineering and artificial intelligence also make extensive use of interface constraints on the set of permitted interactions of a system. Interfaces are the most common mechanism for constraining interaction, but interaction can also be constrained by inner constraints imposed during system design to make system behavior more tractable. For example, serializability in transactions can be viewed as a constraint on interaction among concurrently executing algorithmic operations. Turing machines go even further and impose system constraints that altogether eliminate interface interaction during execution. Constraining the "open-loop" behavior of interactive environments to a subset of tractable "closed-loop" behaviors arises in many application contexts:

The term *harness* captures both the notion of constraining the freedom of an interaction mechanism as in "test harness" and the notion of harnessing an agent for a specific purpose. Turing machines and transactions are system-level harnesses that impose tractable constraints on intractable interaction machines, while interfaces in software engineering and artificial intelligence applications are goal-oriented, domain-specific harnesses.

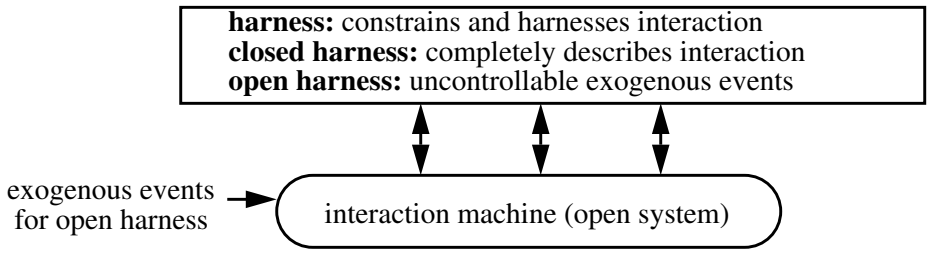


Figure 26: Harnesses that Constrain Interactive Behavior

Closed harnesses like Turing machine tapes that exclude exogenous events can tame intractable interaction mechanisms so that the harnessed system has tractable behavior. But open harnesses like modes of use of an airline reservation system or planning systems remain intractable because exogenous events remain unconstrained. Keeping the system open is often inherently desirable in modeling real-world interaction: unpredictable failures need to be observed and modeled rather than ignored. Open harnesses are therefore an important analysis tool in spite of the fact that they preclude algorithmic analysis. Open harnesses may be simulated by closed harnesses by assuming a probability distribution for the occurrence of exogenous

events (like hurricanes and strikes). The simplest way of closing systems with open harnesses is to make the simplifying assumption that exogenous events do not occur. Thus travel agent interfaces to airline reservation systems assume the absence of overload of other unusual system conditions in specifying the interface behavior.

Examples of harness constraints:
Turing machines: the input tape constrains the state transition function to off-line interaction (see below)
transactions: serializability imposes sequentiality and atomicity constraints on execution (section 4.2)
software engineering: use cases constrain software systems to a specific mode of use (section 5.3)
artificial intelligence: planners constrain agents (dynamical systems) to "planned" behavior (section 6.3)

Turing machines realize their tractability by harness constraints that preclude dynamic interaction. They may be described by an interactive, open-loop computation engine (the state transition mechanism) and a tape that serves as a harness constraint. The "rule of engagement" of the state transition mechanism with its harness (tape) constrain Turing machines to perform algorithmic, closed-loop computation. Finite, pushdown, and linear-bounded automata have the same state transition mechanism as Turing machines, but have more constrained "rules of engagement": finite automata permit only reading from a finite input tape, pushdown automata allow last-in-first-out access to an unbounded scratch tape, while linear-bounded automata bound the size of their tape by the size of the input. Though unbounded read/write tapes of Turing machines have more liberal rules of engagement than other classes of automata, they are constrained by disallowing inputs during the course of the computation. Removing this constraint transforms Turing machines into interaction machines, allowing them to express behavior more powerful than the computable functions. The classification of automata by their rules of interaction with clients makes it clear that the constraints of interaction of Turing machines can be further relaxed to realize completely unconstrained interaction.

Automata constrain interactive state transition mechanisms to an algorithmic mode of use. Different classes of automata can be classified by constraints imposed by their harnesses. Harnesses that constrain the mode of use of interactive system are a tool in software engineering and artificial intelligence. Harnesses in software engineering systems, called use cases (section 5.2), are an important tool of software design. Planners can be viewed as harnesses that constraints on the mode of use of interactive AI applications (section 6.3)

Harnesses are user-defined layers of protective abstraction in the sense of Figure 12, whereas Turing machines impose a system-defined layer of abstraction on all users. In the Turing machines prematurely restrict users to noninteractive computations, while interaction machines provide users with a richer interactive space of potential computations on which harness (interface) constraints may be imposed. User interfaces of PC applications and of banking and airline reservation systems are generally much narrower than harnesses of general-purpose Turing machines, but transcend the Turing machine harness in being interactive. It turns out that the greater freedom of interaction machines is fundamentally necessary as a starting point for expressing interactive distributed and software engineering applications.

The "negative result" that interactive behavior is not expressible by Turing machines determines a "positive challenge" to develop practical models of interactive computation. Proving irreducibility is an important first step in understanding interactive computation, but our main task is to characterize practical empirical models for distributed systems, software engineering, and artificial intelligence. Harness constraints on interaction are a key to expressing useful interface behaviors of inherently intractable interaction machines.

2.5 The expressive power of interaction

The observably richer behavior of interaction machines can be formally demonstrated in two ways by appeal to known mathematical results. Interaction machines have the expressive power of Turing machines with oracles which were shown by Turing [Tu1] to have richer behavior than Turing machines. Godel's

incompleteness result, which shows that Peano arithmetic cannot be expressed by first-order logic, can be adapted to show that interaction machines likewise cannot be expressed by first-order logic. Either result is by itself sufficient to show the greater power of interaction machines, but incompleteness is more powerful, implying the impossibility of extending formal modeling techniques of Turing machines to interaction machines:

Theorem: *Interaction machines have more expressive observable behavior than Turing machines.*
Two alternative proofs *(by appeal to known mathematical results):*
1. Interaction machines are as expressive as Turing machines with oracles, which were shown by Turing [Tu1] to be more expressive than Turing machines. Oracles provide greater computing power by allowing the machine to access oracles that provide noncomputable answers to questions. Interaction machines that passively interact with input sequences generated by oracles or processes in nature can have richer behavior in a more direct sense since their input may not have a recursively enumerable specification.
2. Interaction machines are not expressible by sound and complete first-order logics (adaptation of Godel incompleteness). The key idea is that the set of theorems of first-order logic is recursively enumerable and that the set of true statements of a sound and complete first-order model must also be recursively enumerable. The set of interaction histories of an interaction machine is not recursively enumerable, since it may include input sequences of an oracle or a natural process as well as interaction histories that cannot be sequentially represented (section 4.2). The set of all interaction machine inputs has an even stronger property: its cardinality is that of the power set of infinite sequences which is not enumerable. Since the set of theorems of any first-order logic is enumerable, incompleteness of interaction machines follows from simple arguments of nonenumerability. Second-order logic can be shown incomplete by similar arguments: the set of all predicates has the cardinality of the power set of the integers and is therefore not enumerable.

Interaction machines model persistence, real time, and distributed systems more accurately than Turing machines. The characterize the fuzzy notion "open system" by a precise model of computation. They provide a counterexample to Church's thesis: interaction machines are clearly included in any reasonable intuitive notion of computing, yet they are richer than Turing machines. They provide a canonical form for empirical models of computation that parallels Turing machines as a canonical model for imperative computation, capturing the behavior of distributed systems and of software systems like banking and airline reservation systems. Interaction machines turn out to be as robust a model for open systems as Turing machines are for closed systems: a wide variety of models of interactive computation have equivalent expressive power, including process models and IO automata (section 3.1), embedded software systems, intelligent agents etc. The pi calculus [Mi] claims to be a universal notation for expressing interaction, just as Turing machines and the lambda calculus are universal notations for expressing computable functions. Algorithms can be viewed as the circuit theory of interactive models of computing, modeling episodes of noninteractive automatic computing in an interactive open environment. The dichotomy between formal (Turing) and empirical (interaction) machines occurs in many subdisciplines of computing.

programming languages: procedure-oriented versus object-oriented paradigms artificial intelligence: formalist (logic and search) versus connectionist models software engineering: programming in the small versus programming in the large systems: algorithmic versus reactive (persistent) systems

Figure 27: Dichotomy between Algorithmic and Interactive Models

The extension of algorithmic to interactive models of computing is proceeding along parallel tracks in imperative, functional, logic, and distributed programming. Adding input actions to Turing machines is paralleled in functional languages by adding input types as a first-class primitive. Concurrent logic and constraint programming, implemented by adding input synchronization and don't care nondeterminism to pure logic programs, extends traditional to interactive logic programming. Lynch's input-output automata extend traditional automata with input actions to model distributed systems. Though the concepts and

models of interactive computing are not as mature as those of algorithmic computing, a body of models and analysis techniques is beginning to emerge that includes Milner's pi calculus, Lynch's input-output automata, functional languages with input types, object modeling techniques of software engineering, and models for agents and robots in artificial intelligence.

2.6 Algorithms and complexity

Algorithms determine transformations from prespecified inputs (preconditions) into outputs (postconditions). They have an internal clock that determines the time complexity of computations, but does not express the passage of external time. Turing machines necessarily preclude modeling external time, both because they cannot observe the passage of external time during their execution and because complexity depends only on the number of steps and not on the elapsed time of algorithm execution. Turing machines and algorithms are *off-line* systems that specify the complete course of a computation before it starts: they are *closed* systems because they do not permit external interaction during the course of the computation. Interaction machines extend Turing machines by dynamically executable "read statements", transforming closed Turing machines into open systems which can interact on-line with the environment and express behavior that cannot be modeled by temporal or real-time logics (section 2.6). Turing and interaction machines precisely characterize algorithmic versus interactive computation:

Turing machines = algorithms = off-line = closed = mathematical model -> inner time
interaction machines = reactive systems = on-line = open = empirical model -> external (real) time

Computational complexity deals with quantitative utilization of inner algorithmic resources: it does not address questions of interaction. Algorithmic problems are defined as parameterized classes of instances: an algorithm solves a problem if it computes the solution for every instance. The relation between problems and their algorithms is complex: equivalence of algorithms is generally undecidable and not even partially decidable. Lower-bound complexity, which requires navigation within algorithm equivalence classes, is difficult to establish. Complexity classes refine the notion of computability, partitioning the space of computable functions by the resource requirements of their algorithms. Tractable problems in the class P solvable in polynomial time are distinguished from intractable problems requiring exponential time. The question whether problems in the class NP of nondeterministic polynomial time are computable in polynomial time (whether P=NP) is a central problem of complexity theory.

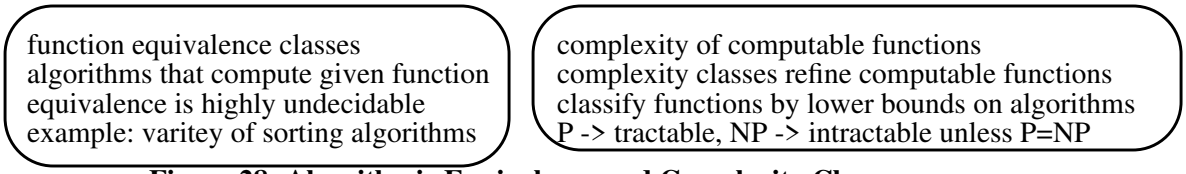


Figure 28: Algorithmic Equivalence and Complexity Classes

Complexity theory is very rich but deals entirely with properties of algorithms. There is no corresponding notion of complexity for interaction machines, though a much weaker notion of "software complexity" is examined in section 5.4.

If problem A is reducible to problem B it is no harder than B, and conversely, if A is known to be hard then reducing it to B shows that B is at least as hard: B serves as an upper bound on the complexity of A, while A serves as a lower bound on the complexity of B. A problem is said to be NP-complete if any problem in the class NP can be reduced to it and it is therefore as hard as any problem in NP. The class of NP-complete problems includes satisfiability (for Boolean expressions), the Hamiltonian path problem, and the traveling salesman problem. An NP-complete problem "represents" the class NP in the sense that its solution in polynomial time implies P=NP. The strong evidence that P is not equal to NP for a wide range of models of computing includes the fact that no NP-complete problem has yet been solved in polynomial time. Complexity metrics are remarkably robust with respect to alternative models of computation [Bo], but if unlimited (exponential) parallelism is permitted, then P=NP.

An X-complete problem Pr represents all problems with complexity X in the sense that if Pr is shown to have weaker complexity Y then all problems with complexity X have complexity Y. This notion of *relative*

completeness is used in other contexts. For example, an AI-complete problem is one to which combinatorially intractable problems of AI may be reduced and whose tractable solution in polynomial time therefore implies the solution of all currently difficult problems in AI (which is actually the class of NP-complete AI problems). The halting problem may be described as RE-complete, since its reduction to a recursive function (that halts for all its inputs) would imply that all recursively enumerable (RE) problems are recursive. The distinction between relative completeness and logical completeness of a formalism in capturing meaning (section 1.2) is as follows: relative completeness of a problems means that it completely represents a complexity class, while logical completeness of a logic means that its semantic domain of meaning is completely expressed by its syntactic formalism.

Computational complexity is one of many kinds of complexity that arise the study of computing. Information (Kolmogoroff) complexity is the length of the shortest program for describing (generating) a given body (sequence) of information. Thermodynamic complexity (entropy) is the degree of disorder in a system. In section 4 we consider a notion of "software complexity" for interactive systems that is not as precise as that of computational complexity but aims to capture the intuitive notion of complexity for purposes of life-cycle cost and scalability. Computational complexity is carefully defined to be mathematically tractable but fails to capture interactive software complexity, just as Turing machines fail to capture interactive computation.

The equivalence in expressive power of Turing machines, the lambda calculus, recursively enumerable functions, and first-order logic is not an inherent intuitive property of computation, but a carefully engineered property of the Turing model. Turing machines capture a limited formal notion of computing that cannot handle interactive software engineering and artificial intelligence applications. The rich and deep results of algorithm and complexity theory owe their aesthetic coherence to the limitations of the Turing machine model, but, as is usual for formal models, the cost of coherence is a loss of expressiveness.

2.7 Biological interaction machines

The double-helix model of cell biology is an interactive model of discrete computation realized by the binding of complementary DNA sequences [Ad]. It turns out that there is a correspondence between binding mechanisms of DNA and communication mechanisms of process models like the Pi calculus, demonstrating a relation between the structure of interaction in computational and physical models.

DNA sequences (character strings from the four-character alphabet ACGT) are data representations of an interactive computing engine that computes by chemical reaction. The affinity of the complementary (dual) pairs A-T and C-G is the basis for dual sequences like CTCG and GAGC that have a binding affinity like ions of a chemical abstract machine (see section 3.1). CTCG can be joined to a second string AGCT by GCTC, whose first two characters complement (and therefore bind to) the last two characters of CTCG and whose last two characters bind to the first two characters of AGCT. The "joint" GCTC can be viewed as an edge in a graph joining the two nodes CTCG and AGCT, as a channel that joins the port CG of CTCG to the port AG of AGCT, or simply as a process whose two beginning and two end characters may be interpreted as port names that are duals of two-character port names CG of CTCG and AG of AGCT.

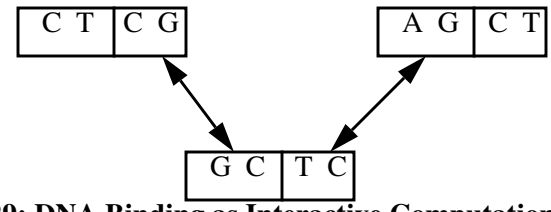


Figure 29: DNA Binding as Interactive Computation

DNA binding protocols express a form of interaction by broadcasting called "unicasting" that broadcasts its DNA binding pattern so it is accessible to everyone but can be captured by (bound to) only a single dual binding pattern. Unicasting corresponds precisely to the form of communication of process models like CSP and CCS (section 3.1), with port names playing the role of DNA sequences. The seren-

dipitous correspondence of DNA and process communication suggests that unicasting is a fundamental communication mechanism of both nature and computing. It combines potential massive parallelism through broadcasting with opportunistic (mobile) choice of a partner and commitment to a particular partner once the choice has been made. Moreover, chemical binding of DNA sequences is like social binding to a reproductive partner, where species play the role of DNA sequences, sexual attraction is the mechanism for affinity between dual DNA strings, and unicasting is the communication mechanism. Dating and mating is a social binding mechanism that corresponds to both chemical binding of DNA and computational binding.of mobile process: unicasting is the common interdisciplinary interaction mechanism.

Millions of copies of any desired DNA sequence can be easily and cheaply manufactured, and the cost of massively parallel computation among selected DNA strings is therefore negligible. If sequences representing nodes and edges of a graph are manufactured and thrown together in a solution, then bindings corresponding to paths in the graph will be created by chemical reactions. By testing the solution for the existence of paths with certain properties a variety of path problems in graphs can be solved by chemical reactions of DNA computers. In particular the problem of finding a Hamiltonian path, which is NP-complete and therefore intractable on traditional computers, can be tractably solved.

To find Hamiltonian paths in a given graph:
encode input and output bindings of nodes and edges as DNA sequences (Fig 29)
manufacture DNA strands for the given encoded specifications
throw in a solution, almost instantaneous linear-time reaction
test for the existence of composite strings with subsequences for each node

DNA computers are massively parallel and effortlessly perform billions of parallel computations, making up in their level of parallelism for their slowness of sequential computation. Because of their massive parallelism they can solve NP-complete problems of moderate size as fast as polynomial-time problems: their time-complexity metric is entirely different from that of sequential computers. The massive parallelism of DNA computers may in time be harnessed for the solution of intractable problems. But more importantly the mode of operation of DNA computers is closer the human brain and may in time explain how humans solve sequentially intractable problems of pattern or scene recognition.

Though computing engines in the brain that cause thinking are unlikely to be actual DNA computers, they are quite likely to use the same binding principles as the chemical DNA model of digital communication. Massive parallelism changes the complexity metric so that the gap between P and NP for moderately-sized problems becomes smaller, though it does not affect arguments concerning the actual truth or falsity of P=NP [Ad]. The DNA computation model suggests why humans have a relative advantage over computers for certain kinds of combinatorial problems in spite of their slow sequential computing speeds.

2.8 Interactive reflection

Computational reflection is the ability of computational systems to represent, control, modify, and understand their own behavior. Reflective systems facilitate introspective computing tasks such as debugging, monitoring, compilation, and execution. They support system evolution and self-reorganization. The reflective nucleus of a system is a “homunculus” that contains an embryonic specification of the system which may be used to perform reflective tasks.

Formal systems have great reflective power but all derivable consequences are already implied by axioms and rules of inference. The same is true for algorithms and all computations in closed systems. Interactive systems have greater reflective power, since their evolution is determined by acquired as well as inherited characteristics. Biological reproduction is perhaps the most astonishing example of reflective creation of organisms of great reflective complexity by combination (composition) of male and female components. We do not understand the secret that underlies the enormous reflective power of DNA in determining the evolution of people from fertilized eggs, but there is no doubt that its power depends on interactive rather than purely transformational reflective system evolution. Humans are the product of

acquired as well as inherited characteristics (figure 30).

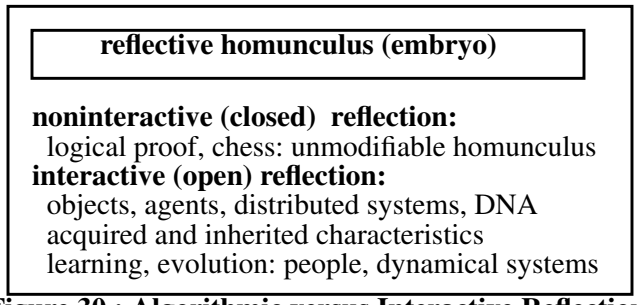


Figure 30 : Algorithmic versus Interactive Reflection

Computing systems and automata are reflectively specified by state transition rules much simpler than their set of programs. Programming languages are reflectively specified by their interpreters. Metacircular interpreters specified in the language being interpreted allow languages to reflect on themselves. A self-referential reflective description of a computing system is causally connected to the system so it can dynamically control as well as statically describe the system. Self-reflective systems were first defined for LISP by a simple interpreter written in LISP for transforming a list representation of any program with its data into a result. Universal Turing machines are self-reflective systems that transform a representation of any Turing machine with its data into a representation of the value. More recently, self-reflective systems for object-oriented languages like CLOS and ABCL/1 have been defined by "metaobject protocols" [KRB] that specify reflective control structure for message handling by protocols. Self-reflective models are useful in capturing the essence of complex systems by a complex set of rules, and allow modifications in the semantics of the system to be simply specified at the reflective level.

Metacircular interpreters capture a notion of noninteractive reflection distinct from interactive reflective interpreters that can be interactively modified. In an interactive reflective system the specification is not only an executable metadescription of a problem solving procedure, but is causally (interactively) connected to the environment so it can both cause changes in the environment and be affected and change in response to environmental stimuli. Reflection is a powerful mechanism from a simple set of rules even for noninteractive.systems. We do not understand the much richer reflective power of biological reflective models, but it appears to be governed by principles of interactive reflection.

3. Logic and Semantics

Logical reasoning is a form of computation. Well-formed formulae are programs, rules of inference are computation rules, the theorem to be proved is the initial data, proofs are computations, and truth or falsity of the theorem is the value.

well-formed formulae = programs
rules of inference = computation rules
theorem to be proved = initial data
proofs = computations
truth value of theorem = value

The rules of reasoning determine a "proof theory" concerned with properties of the syntactic representation, while the behavioral properties of its modeled world determine a "model theory". From the viewpoint of model theory, well-formed formulae are assertions that become either true of false when function and predicate symbols are interpreted as functions and predicates of the semantic modeled world (domain of discourse). Though logic is a form of computation, the converse view that computation is a form of logic is not correct, since computing systems are not in general expressible in terms of logic.

every proof is a computation but not every computation is a proof

A logic is sound if all provable statements are true and complete if all true statements are provable. More precisely, a logic is sound if all its theorems are true for all possible interpretations of its nonlogical

symbols and is complete if all assertions true in all interpretations are provable. Soundness implies that the set of provable statements is a subset of the set of true statements, completeness implies that the set of true statements is a subset of the set of provable statements, while soundness and completeness together implies that the set of true and provable statements are coextensive.

The first-order predicate calculus is a sound and complete logic whose semantic model is a world of functions and predicates in which the semantics of computable functions and algorithms may be expressed. Whereas the semantics of algorithms and Turing machines can be expressed in the first-order predicate calculus, the semantics of interaction machines cannot be syntactically captured by any sound and complete first-order logic.

Soundness and completeness relate the syntactic representation R of a logical model to its semantic modeled world W. Soundness ensures that a representation correctly models behavior of its modeled world that is representable, while completeness ensures that all possible behavior is modeled by the representation. Soundness and completeness together ensure that a representation is both correct and as expressive as the world being modeled. But completeness restricts the semantics of modeled worlds to those that can be completely expressed by a representation. The requirement of soundness and completeness constrains modeling power to modeled worlds whose behavior can be entirely captured by a syntactic representation.

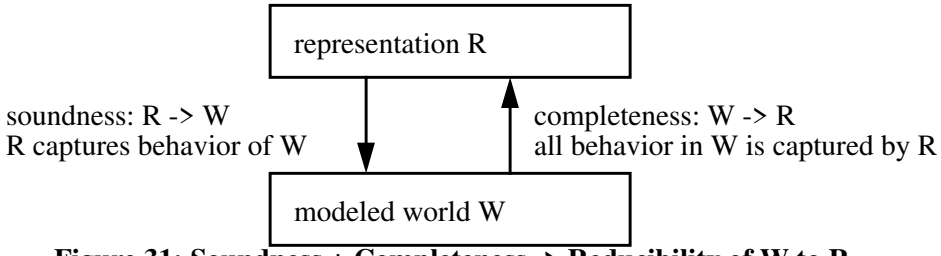


Figure 31: Soundness + Completeness -> Reducibility of W to R

Soundness and completeness imply complete expressibility of a modeled world W by a representation R. In this case we say that W is reducible to R, and also that W and R have the same level of complexity and abstraction. Declarative and imperative modeled worlds W are reducible to syntactic representations R, while empirical interactive worlds, such as the real world, are not. Reducibility of W to R implies completeness of R in expressing properties of W, while incompleteness implies irreducibility. Godel incompleteness of the integers implies their semantics is not expressible by (reducible to) a syntactic representation (God created them).

The notion of reducibility of W to R for models is closely related to the notion of reducibility for algorithms. If algorithm B is reducible to A then A provides a model of behavior of the computational behavior of B that is both sound and complete. The use of algorithm reducibility to calibrate the complexity of algorithms corresponds to the use of model reducibility to calibrate the abstraction of models. If algorithm B is reducible to A it is no more complex than A, while if a modeled world W is reducible to R it is no more abstract than R.

In this section we identify the notion of modeling that scale up from algorithmic to interactive models and review extensions of first-order logic proposed for modeling computation. Two important notions that scale up but subtly change their meaning are that of declarativeness and typing. Interactive declarativeness retains the notion of describing “what” independently of “how” but extend the class of things having what specifications from functions to objects and systems in the real world. Interactive typing likewise extends abstract behavior description from functions to data abstraction and interactive objects.

The extensions of logic considered here include modal, intuitionistic, linear and temporal logics. though each of these expresses significant properties not directly expressible in first-order logic, none can capture interaction machine or open system behavior. Nonmonotonic logic does break out of the algorithmic mold in violating the closed-world assumption, but sacrifices its claim to be a logic by not preserving the monotonicity of truth. It seems better to reserve the term logic for systems that do preserve the monotonicity of truth and invent new terminology for nonmonotonic reasoning systems.

Modal logics, which support reasoning about contingent truth in "possible worlds", extend the propositional calculus with axioms about modal operators. *Linear logic*, which views assumptions as resources consumed by applying rules of inference, modifies both classical and intuitionistic logic so axioms can only be used once, thereby capturing the semantics of interactive streams of data (message passing). Temporal logic supports qualitative reasoning about infinite sequences of events in reactive systems, and has been extended to reasoning about certain quantitative properties of sequences, though it does not capture the "complete" behavior of interactive (real) time. However, all these logics involve reasoning that does not allow inputs during the process of deriving conclusions from assumptions, and is therefore noninteractive. This corresponds to the Turing machine restriction that no new input can be added during a computation and to the closed-world assumption (section 5.5) that proofs are performed in a closed world. The equivalence of Turing machines and first-order logic is attributable to the fact that both model the derivation of conclusions from assumptions in a closed world.

Semantics associates meaning with syntactic expressions of formal and natural languages. Mathematical models theory determines a "clean" notion of meaning for expressions of first-order logic for modeled worlds with a static mathematical domain of discourse and a restricted notion of meaning (being true in all interpretations). This notion of meaning is rich enough to specify meanings of Turing machines and algorithmic programs as predicates (pre and post conditions). But the specification of programs as predicates breaks down for interaction machines and interleaving programs.

Whereas logic focuses on the meaning of logical symbols for all interpretations, semantics is generally concerned with the meaning of nonlogical symbols for specific interpretations. Semantic formalisms include axiomatic (logical) semantics that support program verification, operational semantics that captures the execution properties of implementation, and denotational semantics that associates mathematical denotations with computational objects, aiming to express the abstract Platonic meaning of programs. However, all these models are formalist and fail to express the semantics of interaction.

Model checking, which specifies concurrent semantics of linear and branching time temporal logic by finite automata, can be more powerful than logic, especially if it is extended to Buchi automata with infinite input strings [VW]. Automata are more dynamic that static domains of logic: they permit the modeling of sequential algorithmic time but cannot model interactive time.

The semantics of both interactive and algorithmic components is modeled by their observable behavior, but interactive observable behavior is very different and more complex than that of algorithms. Interactive components cannot be specified by pre and post conditions, but can be described by a weaker analog of "before" and "after" behavior in terms of supplied behavior S and demanded behavior D. Interaction requires that supply exceed demand, and the unconstrained supplied behavior of components is generally too rich to be formally specifiable. However, a formally specifiable subset of guaranteed behavior G can generally be identified that is sufficient for the needs of the client. Interactive behavior in relation to a specific client can be specified by an assume-guarantee relation (A-G) that is adequate if the assumptions A of the client are a subset of the guaranteed behavior of the component [V, AP]. Assume-guarantee compatibility is generally specified by type systems: the type of a component provides the guarantee, while the type of the clients request specifies the assumption. In principle interactive assume-guarantee specifications should specify temporal as well as functional compatibility, but in practice there are system-wide temporal compatibility constraints that are automatically satisfied. Interactive components guarantee correct behavior for clients that respect their guarantee but may exhibit arbitrary behavior otherwise.

Clients determine a *harness* for interactive components (section 1) that constrains behavior to a usually formalizable subset of behavior, even in the absence of explicitly specified guaranteed behavior. For components that serve multiple classes of clients, the harness is the union of the guarantees for all classes of clients. The functional behavior of components is specified by their interface type, while temporal guaranteed behavior is specified by interaction protocols that generally impose strong temporal constraints of serializability and mutual exclusion.

3.1 Irreducibility and incompleteness

Godel incompleteness for the integers reflects the incompleteness of many other domains whose sets of true assertions are not recursively enumerable, including that of interaction machines. It strikes a blow against reductionism, and by implication against philosophical rationalism. The irreducibility of semantics to syntax, which is considered a fault from the viewpoint of formalizability, becomes a feature of empirical models in permitting empirical semantics to transcend the limitations of notation. Plato's despairing metaphor that our view of the real world consists of mere reflections of reality on the walls of a cave was turned around by the development of empirical models that predict and control such partially-perceived reflections of reality. Empirical models accept inherent incompleteness and irreducibility, developing methods of prediction, control, and understanding for inherently partial knowledge. Empirical computer science should, like physics, focus on prediction and control in partially specified interactive modeled worlds, since this provides greater modeling power than completely formalizable algorithmic models.

In showing that the integers were not formalizable by first-order logic, Godel showed that Russell and Hilbert's attempts to formalize mathematics could not succeed. These insights also open the door to showing the limitations of formalization for computing. The idea of incompleteness, introduced by Godel to show that the natural numbers are not reducible to first-order logic, may be used also to show the irreducibility of interaction machines and more generally of empirical systems whose validation depends on interaction with or observation of autonomous environments. Turing machines lose their statues as the natural, intuitive, most powerful computing mechanism but retain their status as the most powerful mechanism having a sound and complete behavior specification. These limitations of logic imply that formal correctness proofs have a limited role as evidence for the correctness of interactive systems. Hobbes' assertion that "reasoning is but reckoning" remains true since logic is a form of computing, but the converse assertion that "reckoning is but reasoning" is false since not all computing can be reduced to logic.

Proofs of correctness for all possible interactions are not merely hard but impossible for software systems because the set of all possible interactions cannot be specified. Testing provides the primary evidence for correctness and other evidence like result checking is needed to check that the result actually produced is adequate. Even partial result checking, like parity checking, can greatly increase confidence in a result. Formal incorporation of on-line result checking into critical computing processes reinforces off-line evidence of testing with post-execution evidence that the actual answer is correct. The nature of evidence for the adequacy of computations, the relation among different kinds of evidence, and the limitations of formal methods and proof in providing such evidence are important fundamental questions.

3.2 Interactive Declarativeness

Declarative specifications describe objects of a modeled world independently of the operations performed on them. Functional and logic programming languages specialize declarativeness to the implementation-independent description of functions. Interactive declarativensss generalizes "what" specification to the description of interactive computations which model persistent objects. Object interface specifications describe functionality independently of execution and are declarative in this broader sense. This broadening of meaning is typical of many terms when they are lifted from the restrictive world of functions and

algorithms to the more general world of interaction

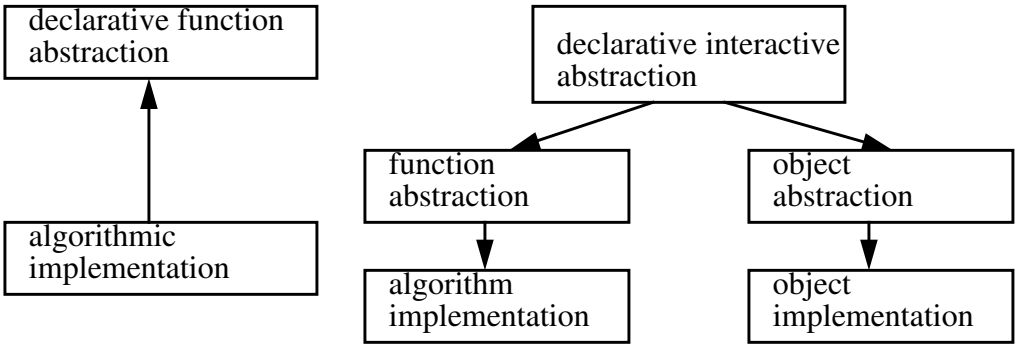


Figure 32: From Functional to Interactive Abstraction

Object interface abstractions are stronger than function abstractions because they abstract not only from function implementation but also from communication implementation by protocols. Even more important, object interfaces abstract from potential object behavior that may be acquired during interaction. Declarative interaction specifications are both implementation independent and behaviorally incomplete.

The declarative, imperative, and interactive paradigms are complementary forms of program specification that address complementary aspects of program behavior. Algorithms can in principle be entirely specified by declarative abstractions, but are in practice specified as programs.

Typed programming languages combine declarative specification for types and imperative specification of algorithmic transformation. Interactive programs are generally specified by a combination of declarative, imperative, and interactive specification. Object-based programs are specified by declarative object classes, imperative method specifications, and interactive message passing protocols.

Declarative, imperative, and interactive paradigms are associated with the mechanisms of classification, state transition, and communication. Classification is the basis for computation in function, logic, and constraint languages, and relational databases. Logic can be viewed as computation by classification: statements like “all men are mortal” assert that the class “man” is a subclass of the class “mortal”, and modus ponens may be interpreted as the inference that belonging to a class implies belonging to a superclass (x is a man implies x is mortal).

Twenty questions is an example of problem solving by classification, narrowing the class of an entity until a singleton class is identified. Quicksort computes by classifying elements to be sorted into equivalence classes with smaller numbers of elements till all equivalence classes have only one element. Types are an example of a partial classification mechanism that can be superimposed on imperative paradigms to reduce execution-time complexity by capturing invariants.

modus ponens: if x is in set A and B is a superset of A then x is in set B twenty questions: identify an object by 20 classification questions quicksort: sort a list of length n by log n (on average) classification steps types: preclassify at compile time, ensure compatibility, efficiency at run time

Figure 33: Four Examples of Computation by Classification

The state transition paradigm is familiar from Von Neumann computers and automata theory, being the preferred operational mechanism for computation in practical computers. Communication is not supported by state-transition paradigms as formulated by Turing or by classification paradigms as expressed by first-order logic. It can be realized by adding input actions to imperative models to extend Turing machines to interaction machines, by adding input types to functional languages or by adding don’t-care nondeterminism to logic languages. State transition paradigms have an operational semantics, classification paradigms have a declarative or denotational semantics, while communication paradigms do not have a complete for-

mally specifiable semantics.

Pure state transition, classification, or communication paradigms are associated with low-level formalisms: the lambda calculus is a low-level declarative formalism, Turing machines are a low-level imperative formalism, and the pi calculus is a low-level interactive formalism. ML and Lisp are examples of primarily declarative formalisms whose practicality is enhanced by imperative features. Higher-level formalisms generally combine state transition, classification, and communication paradigms, suggesting that multiple paradigms allow flexible choice of both the type and granularity of abstractions for problem solving. Typed procedure-oriented languages combine state transition and classification paradigms, IO automata combine state transition and communication paradigms, and functional languages with input types combine declarative with communication paradigms. The strength of the object-oriented paradigm is due to its striking an appropriate balance among state transition, classification, and communication paradigms. It combines state transitions within objects with classification at the level of classes and inheritance and communication at the level of messages. It has the expressive power of interaction machines while supporting the state transition paradigm for algorithmic computation within objects and supports not only first-order classification of values but also second-order classification of classes (figure 34).

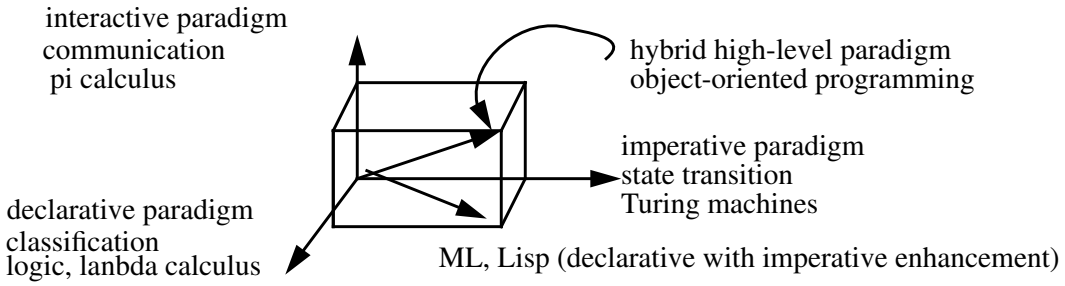


Figure 34: Pure versus Hybrid Paradigms

3.3 Models and theories of type

Types are a particular style of "abstraction by classification" that classify entities into classes with similar properties. The view of type as a form of classification is applicable to both algorithmic and interactive paradigms of computation. Types for functional languages classify functions by their domains and ranges, while types for interactive systems classify interactive entities by their observable behavior. For example, objects in object-based languages are classified by the set of methods in their interface.

The notion of "type" and "classification" was used as an organizing principle for scholarly disciplines long such as mathematics and biology long before the birth of computing. The term type means different things to mathematicians and programmers, but the meanings are sufficiently similar to cause confusion. Set theory, which has sets as its domain elements, is concerned with typing its sets to avoid antimonies concerning the set of all sets that are not members of themselves. Constructive (intuitionistic) type theory aims to bridge the gap between mathematical and computational notions of type, but its notion of type as the set of proofs of a proposition is somewhat contrived.

Intuitionistic type theory, which has proofs of propositions as its domain elements, classifies proofs by the proposition that they prove. The "propositions as types" model views propositions as declarative type specification of a collection of equivalent imperative proofs. The Curry-Howard isomorphism plays a central role in establishing the analogy between propositions and proofs in logic on the one hand and types and computations on the other. This analogy is particularly strong when logic is constrained to be constructive, but it is valid for only for a restricted class of computational models.

Types in programming language classify collections of values into classes with similar computational properties. Types in both constructive type theory and programming languages specify invariant properties of values of the type that allow the construction of compatible composite structures from components. Declarative (functional) languages further determine a dynamic correspondence between computations of

a declarative specification and proofs of a proposition, while imperative and interactive languages do not.

Typing of expressions is useful even in untyped programming languages like Lisp, but statically and strongly typed languages that enforce statically checkable binding of values and variables to a type provide safety and efficiency at execution-time. Strong typing realizes safety, efficiency, and conceptual simplicity by requiring computers and people to make invariant assumptions about run-time behavior, while untyped languages provide greater expressive freedom. Typing initially focused entirely on consistency of usage of local bound variables within an algorithm, but has been extended to constraints on free variables in object and process interfaces. Notions of typing are equally applicable to declarative, imperative, and interactive paradigms, providing a simple calculus of classes whose complexity is independent of the execution complexity of the associated evaluation machine. Interactive types that guarantee the interface behaviors of an object are important in constraining intractable supplied behavior to a tractable subset of used behavior. Type systems approximate behavior by a decidable invariant that acts as a checkable specification of behavior for algorithms and as a harness that constrains interactive behavior to tractable guaranteed subsets. The role of types as specifications of algorithmic behavior differs from their role as guarantees and harnesses that tame richer behavior in interactive systems.

Polymorphism aims to ease the inflexibility of static typing while retaining safety and efficiency though not simplicity. Polymorphism occurs through coercion and overloading as well as through type inference mechanisms that introduce most general (universal) types whose meaning is specialized to a specific type by an execution-time parameter. Types impose a granularity of abstraction on programming languages: programmer-defined types allow control of the granularity of typed abstraction by the programmer. Typed languages superimpose two independent calculi for reasoning on programming notations: one for reasoning about types and one for computing with values. Constructive (intutitionist) mathematics, which aims to express mathematics in terms of constructive notions that reduce mathematics to computing, interprets logic in terms of constructive type theory by interpreting propositions as types and values as proofs that demonstrate the existence of elements of the type. It views propositions and associated types as a fundamental concept from which values are derived by construction, rather than viewing values as the fundamental concept and types as a derivative concept for managing collections of values.

Which comes first: the value or the type? Are types simply a convenient construct for organizing collections of values (like geometrical constructions), or are they the fundamental conceptual building blocks from which values are constructed? Common sense suggests the first view, while constructive type theorists find it tempting and convenient to view constructive types as the conceptual starting point for constructing values and to adopt the Platonic view that types are more real than the values the inhabit them. However, the view of types as a fixed set of categories is applicable only to logics with a fixed modeled world. Interactive models with an incomplete, evolving modeled world also have an evolving associated model of type. It is interesting to reexamine the notions of constructive type theory for interactive models, since the idea that types be constructive still appears valuable, but the notion of a fixed set of types generating an associated fixed set of values must be discarded. Relations among types can be modeled by algebras, by calculi for type inference, or more loosely by inheritance hierarchies, which enrich interactive type systems by providing a framework for the second-order classification of classes.

The term type is overloaded, referring both syntactic type checking properties of expressions and semantic behavior properties of values. The theorist, language designer, compiler writer, and application programmer require different models of type. Their syntactic role in specifying the structure of expressions for type checking may be distinguished from their semantic role in specifying the behavior of data during execution. Object-oriented programming eases the overloading of the term “type” by introducing the term “class” for semantic behavioral properties and reserving the term “type” for syntactic properties of expressions. Object-oriented programming derives its strength in part by seamlessly adapting principles developed for declarative and imperative typing to interactive systems. The extension of the notion of type to the classification of temporal as well as transformational properties of commands is being explored in systems like Hermes (section 4.4).

3.4 Programs are not predicates

The behavior of programs of an interaction machine cannot in general be specified by recursively enumerable predicates. Interactive program behavior is specified by interaction histories where each interaction involves algorithmic computation but the pattern (scenario) of all interactions in an interactions history may be infinite and is not in general recursively enumerable. Interaction histories can be noncomputable even for interactive identity machines and sequential interaction machines. Interaction histories of concurrent interaction machines are not in general describable by linear sequences of events and can be very complex.

The complete behavior of an interactive system is not describable: descriptive incompleteness is a consequence of logical incompleteness. Partial behaviors for harnesses (section 2.4) and use cases (section 5.3) may be described, but for open harnesses even partial descriptions require further conditions of freedom from nonlocal interference to eliminate nonlocal "action at a distance".

Formal specification of programs by predicates plays a role in the analysis of interfaces (harnesses) of an interaction machine, but cannot be used to completely describe program behavior. This explains the relatively low use of formal techniques in object-oriented programming and interactive modeling. The impossibility of formal representation of programs by predicates is psychologically important in establishing different goals and techniques for interactive modeling from those of algorithmic modeling. Techniques of interactive specification by harnesses, design patterns, transactions, planners etc. are explored in sections 4, 5, and 6.

Functions from a domain X to a range Y can be specified as the set (predicate) of ordered pairs (x,y) over X*Y that are true if and only if y is an output for input x. Declarative specifications determine results independently of order of evaluation: operational semantics in logic programming is viewed as logic + control, where logic specifies declarative meaning and imperative order of evaluation does not affect declarative meaning. The Church-Rosser theorem guarantees that the end result is independent of the order of evaluation in pure functional languages, while pure logic programming languages guarantee independence of order of evaluation by don't-know nondeterminism.

Declarative specifications directly capture essential meaning because they define what is to be computed independently of how it is computed. Since declarative specifications ignore irrelevant detail concerning the order of evaluation, they have the potential of being simpler than imperative specifications. But there is a trade-off between expressiveness and tractability of declarative specifications. Though many mathematical problems have a simpler declarative than imperative specification, many algorithmic problems are more simply and elegantly expressed by an algorithm than by a declarative specification.

Turing machines and procedure-oriented program can in principle always be expressed as predicates, though some imperative programs are more simply specified by algorithms than by predicates. But interactive programs cannot in general be specified by a predicate even in principle. Their behavior cannot be specified as a transformation (relation) between inputs and outputs: it is an interaction history over time rather than a transformation or transformation history.

3.5 Modal, intuitionistic, and linear logic

Modal logic [HC] is a variant of fist-order logic for reasoning about contingent assertions true in some interpretations and false in others. Interpretations in first-order logic correspond to possible worlds in modal logic: being true in all interpretations corresponds to being true in all possible worlds. Modal logic extends the propositional calculus with the modal connectives "necessarily true in all possible worlds" and "possibly true in some possible world" which we denote by N and P. The semantics of N and P is related to the quantifiers "forall" and "exist" (denoted by A and E), though N and P quantify over sets of possible worlds while A and E quantify over values of variables. Possibly true can be defined as not necessarily false by analogy with a corresponding duality for quantifiers:

P(F) <--> ~N~(F) corresponds to E(x) F(x) <--> ~A(x)~F(x)

Mints [Min] builds on the correspondence between propositional modal logic and first-order monadic logic to show a correspondence between formulae, proof structures, and models, including properties of

soundness and completeness. However, modal logics consider the auxiliary notion of reachability among possible worlds representable by reachability graphs (Kripke structures). The modal logic S5, which assumes that every possible world is reachable from every other, is the richest modal logic and also the one most directly modeled by first-order monadic logic. The property that all world are possible is expressed by the axiom "PF = NPF" (if F is possible then it is necessarily possible). The weaker modal logics T and S4 are defined by weaker reachability conditions among possible worlds: T restricts the meaning of N and P to a relative notion of necessity and possibility for worlds reachable from the current world, while S4 defines the relation of reachability by a transitive closure relation.

Intuitionistic logic has weaker inference rules than classical logic: it does not admit (A or not-A) as an axiom because there is no way of constructively establishing whether A or not A is true. Intuitionistic logic models computation more precisely than classical logic. Formulae in intuitionistic logic have both a "logical reading" as propositions with rules of inference for deriving conclusions or a "type reading" as types whose proofs, interpreted as elements of the type, correspond to programs. Computations of programs, realized by reduction rules as in the lambda calculus, correspond to logical cut-elimination rules.

The Curry-Howard isomorphism determines a correspondence between the construction of formulae in the propositional calculus and the construction of composite types in programming languages with the cut rule of inference corresponding to the operation of applying a function to its arguments (beta reduction in the lambda calculus). This exact correspondence between proofs in the intuitionistic propositional calculus and computation in the lambda calculus is not as close for imperative as for functional languages, since imperative types must account for history-sensitive variables whose behavior differs from that of declarative variables. The correspondence breaks down completely for imperative programs, though the notion of type as a classification abstraction is still applicable. Though types in constructive logic and in programming languages are examples of abstraction by classification, the relation between propositions and their proofs differs from that between imperative and interactive types and associated values.

Linear logic [Ab, Tr] is a resource conscious logic that captures the semantics of nonreusable resources. It may be modeled by rewriting rules on multisets, where each inference uses elements of the multiset specified in its assumptions and produces elements specified in its conclusions. Petri nets that use tokens on their inputs to produce tokens on their outputs likewise operate according to linear logic. Linear logic interprets propositions and logical variables as nonreusable resources. Its logical rules mimic those of classical logic, but it drops the structural contraction and weakening rules to realize nonreusability of resources. However, it introduces an "of course" operator !A that tags A as a reusable resource, allowing the expressive power of reusable resources as a special case. It thereby trivially has the expressive power of classical intuitionistic logic by supporting both reusable and nonreusable resources, but increases the complexity of inference to gain this extra expressive power. Linear logic has pairs of "logical" introduction and elimination rules for "and", "implies", and "or" that respectively specify constructors for composing formulae from components and selectors (destructors) for selecting components.

Linear logic models nonreusable resources for both imperative assignment and interactive message passing: it is equally applicable to modeling imperative and interactive nonreusable resources. It can also be used to model nonreusability of arguments in declarative reduction rules of the lambda calculus, but is not essential because the cut rule of traditional logic is already available for this purpose.

3.6 Temporal and real-time logics

The expressive power of temporal logic depends on its abstraction of time. Classical temporal logic [MP] represents time by a discrete sequence, providing a framework for qualitative reasoning about states observable at points of a sequence, for example sequences of states executed by algorithms. Temporal logic is applicable in models that express time by sequences, like the interleaving model that maps collections of potentially concurrent sequential processes into the set of all possible interleaved sequences. It permits inferences about concurrently executed processes by reasoning about the associated set of interleaved sequences. However, representing time abstractly by sequences excludes reasoning about quantitative properties of time not expressible by sequences, like duration.

Manna and Pnueli [MP] recognize that the interleaving model does not fully express real concurrency and examine the mechanisms for expressing properties like duration by properties of sequences at a lower level of granularity. They use interleaving models for reasoning about properties of reactive systems like fairness, safety, and liveness. However, temporal logic models only restricted properties of reactive systems for computations expressible by linear sequences (traces). It cannot reason about behavior of reactive systems not expressible by interleaving, capturing only a restricted notion of algorithmic time and not distributed behavior of reactive systems in real time.

Temporal operators include "future operators" like *next, always*, and *eventually* as well as "past operators" like *preceding*, *has always been*, and *once was*. If x is a property of a state, like x>0, then next(p) is true at position j of a sequence if p is true at position j+1, always(p) is true at j if P is true for all positions ≥ j, and eventually(p) is true at j if there exists a state ≥ j for which p is true. Temporal logic allows inferences like "always(p) implies p" and "eventually(p) implies not(always(not(p)))" indicating that *always* and *eventually* are related like *forall* and *exists* in the predicate calculus. The axioms and rules of temporal logic allow proofs that involve reasoning about infinite sequences and may be used to prove properties of concurrent processes like mutual exclusion.

More expressive models of time have been considered that map states into time-stamped sequences or other metrics reflecting the properties of real time. [AH] classify a wide variety of "real-time logics" by their complexity and expressiveness, concluding that most formalisms for time in the literature have undecidable logics. Though logics that model time by mapping sequences onto real-time metrics capture some quantitative temporal properties and are more expressive than temporal logic, such logics are not rich enough to capture the general notion of interactive time in distributed systems. The approach of [HLP] which interprets one of the state variables as a clock and allows temporal operators to be annotated with temporal constraints is likewise limited in expressive power. It allows the qualitative operator "eventually" to be refined into a quantitative constraint "eventually within a fixed amount of time (say 5 seconds)", but does not escape from the limitation that time is a discrete sequence of instantaneous events. Time in distributed systems cannot, however, be modeled by sequences (section 3.6), and even when time can be sequentially described, sequences corresponding to interaction-machine behavior may not be recursively enumerable if generated by oracles or natural processes. Logics of time cannot completely describe temporal behavior in computing systems: they can at best provide algorithmic approximations to inherently non-algorithmic interactive temporal behavior.

3.7 Logic and constraint programming

Proving theorems from axioms by rules of inference corresponds to computing final states from initial states by (nondeterministic) state transition rules. Traditional logic is strongly noninteractive in that no new axioms can be added during theorem proving: Theorem proving that allows heuristic help or provable theorems to be interactively added constrains interaction to information that could in principle have been derived algorithmically, showing that interactive computing can reduce the complexity of algorithmic tasks as well as increase expressive power.

Prolog programs call theorems "goals" and their axioms, called "Horn clauses", have the form "H :- B1,B2,...,BN", where H is the head and B1,B2,...,BN are the body. They prove goals (theorems) from clauses (axioms) by a reduction rule called "resolution" that unifies a goal with a clause head and replaces the matched goal by goals of the clause body. Unification is a bidirectional form of once-only assignment for logic variables whose values may be expressions that contain variables. Clauses "H :- B1,B2,...,BK" have a declarative reading "H is true if B1 and B2 ... and BN is true" and an imperative (procedural) reading "to solve H solve B1 and B2 ... and BN". Goal sets have the declarative reading "is G1 and G2 ... and GN true?" and the imperative reading "solve G1 and G2 ... and GN". Logic programs are executable specifications whose declarative reading captures their role as specifications and whose imperative reading captures their role as executable programs. Since goals G can in general unify with more than one clause, computation in logic programs is nondeterministic. Nondeterministic exploration of a proof tree, realized by backtracking, precludes interaction, allowing only a yes or no answer when the search is complete.

Concurrent logic programs replace complete nondeterministic exploration of the proof tree by committed choice (don't care) nondeterminism that sacrifices completeness to realize interaction.

Logic programming can be modeled as constraint solving by viewing the set of logic variables of a store with each resolution step imposing an additional constraint (partial value assignment) on the store. Constraints specify partial information that is progressively refined into complete information to yield the solution. Constraint solving by progressively imposing constraints is monotonic: each additional constraint is a restriction (refinement) of an already existing set of constraints.

Constraint programming is a declarative problem-solving paradigm that uses "classification" as its paradigm of computing. Adding a new constraint is referred to as a "tell" operation while checking whether a particular constraint is implied by given constraints is called an "ask" operation. Since constraints are noninterfering they can be computed concurrently. Constraint logic programming can be realized by concurrently solving constraints corresponding to goals in a goal set. If the set of constraints generated by this process is consistent the problem is solved, while inconsistent constraints show the problem has no solution. However, the constraint approach, like the corresponding logic programming approach, must sacrifice completeness when dealing with reactive computations that arise for robots and multiple interacting agents. The analog of don't-care nondeterminism in reactive constraint systems forces committed choice that may exclude desired solutions. Committed choice in applying constraints, just as in applying rules, may exclude regions of the space that include the solution. Reactive constraint systems for robot motion and hybrid control are discussed in [SH].

4. Distributed Interaction

A system of components is *physically distributed* if its components are geographically separated and is *logically distributed* if its components have different name spaces and interaction among components is more expensive than internal communication. Distribution and concurrency are logically distinct notions, since distribution involves separation in space while concurrency involves simultaneity in time. Interaction and concurrency are also logically distinct notions (see Figure 17). Systems that are concurrent and distributed are closed and can in principle be modeled algorithmically, while interactive systems are open and may have richer than algorithmic behavior even when they are neither concurrent nor distributed.

Section 4 focuses on the role of interaction in distributed systems. The analysis of process models in section 4.1 confirms that it is interaction rather than distribution or concurrency that is the source of noncompositionality and nonalgorithmicity. The notion of "concurrent composition" of process models such as CCS and the Pi Calculus would have been better named "interactive composition", since it is not necessarily concurrent but necessarily interactive. Process models developed by Milner extend the lambda calculus to interaction, complementing the extension of Turing machines to interaction machines. Milner's concurrent (interactive) composition mechanism and his broadcasting mechanism, which mimics that of biological and chemical computing mechanisms, can be directly used for interaction machines.

Section 4.2 shows that transaction correctness is a condition of local noninteraction that permits individual transactions to be treated as algorithms. The term "concurrency control" would have been better called "interaction control" since it eliminates interaction among transactions while permitting them to execute concurrently. Serializability does not necessarily restrict concurrency, but strongly restricts interaction. Transactions provide a mechanism for taming distributed intractability by guaranteeing local noninteractiveness as a sufficient condition for algorithmic behavior in a concurrent, distributed environment.

The role of time in distributed systems is examined in section 4.3. Objects and processes interact through ports that may have temporal as well as functional behavior. When temporal protocols are constrained to synchronous input, temporal behavior need not be explicitly modeled and it is sufficient to model only functional behavior. But airline reservation systems or automatic teller machines with multiple input ports in distributed locations require temporal protocols to be explicitly specified. Temporal behavior in process models can be specified by guarded input actions, message-passing protocols, time stamping, duration constraints, nonserializable transaction protocols, or harnesses that constrain modes of use. Temporal logics that model time as a sequence of equally-spaced instants do not capture duration, real-time, or

multiple overlapping inputs at distributed locations that require nonlinear notions of interactive time.

Research in distributed systems focuses on principles of interactive modeling shared by the operating system, database, software engineering, and artificial intelligence communities. Distributed systems with multiple input ports making concurrent demands on shared resources, like airline reservation systems, require more complex interaction abstractions than objects or processes with a single input stream. Multi-tape Turing machines illustrate that distributed noninteractive behavior does not increase expressive power and can be handled algorithmically. Synchronous interactive systems like process-control computers are algorithmically analyzable [BWM] though interactions may be generated by nonalgorithmic oracles or natural processes. Asynchronous interaction machines are more expressive than synchronous machines [We2] because they can model richer classes of interactive environments.

Distributed input at multiple locations complicates interactive observability because observation of (interaction with) such systems cannot be represented by a linear temporal input stream [BBHK]. Algorithms are the discrete analog of differential-equation systems with one-point boundary conditions, while distributed interactive systems have distributed boundary conditions (input actions) over space and time.

4.1 Process models

Process models for programming languages such as CSP [Ho] or process calculi such as CCS or the pi calculus [Mi] restrict inputs to a sequential stream, either by a single input queue as in actors or monitors or by an internal control mechanism as in CSP or Ada. Guarded commands of the form “guard: action” whose guards may specify temporal as well as state-sensitive conditions for the action to take place, thereby allowing the semantics of actions to depend on temporal as well as transformational properties. The insight that guarded commands can specify temporal as well as functional behavior is an important one. Input protocols specified by guarded commands can be nondeterministic to reflect uncertainty and lack of control over external inputs.

Process models may be characterized by action structures [Mil2] with strongly controlled interaction. Process interaction is modeled functionally in the pi calculus as a form of lambda calculus reduction where the binding operator "lambda" is interpreted as a "receive" command": "lambda(x)M applied to N" is interpreted as "receive(x)M interacting with send(N)". Though processes persist and may send and receive streams of values while functions consume their arguments (as in the logical cut rule), the transformation semantics is identical. Once lambda abstraction is identified with receiving and function application with interaction between a receiver and a sender, the modeling of processes by a message-based extended lambda calculus with concurrent beta reduction is natural. The pi calculus transmits names rather than values across send-receive channels, facilitating computation that changes the channel topology and strengthening the correspondence with the lambda calculus, which also identifies names and values.

The lambda-calculus like simplicity of the pi calculus reflects the fact that computation rules of sequential and concurrent computation are identical and it is only protocols of interaction that differ. The composition (interaction) P|Q of two processes P and Q as “*P and Q acting side by side interacting in whatever way we have designed them to interact*”, thereby explicitly passing the buck on defining composition protocols to designers of interaction protocols. The simplicity gained by quantifying over “all possible modes of interaction” is similar to that realized by quantifying over all possible interpretations in first-order logic.

Both CSP and the pi calculus employ a “broadcasting protocol” that broadcasts messages to all receivers tuned in to a named “wavelength” but delivers them only to only a single eligible receiver. The scope of broadcasting is controlled by a two-way restriction operator that blocks outer access to inner structure as well as inner access to outer structure. The restriction that only a single receiver gets the broadcast message, which violates the semantics of radio and television broadcasting, captures the idea of broadcast messages as nonreusable entities with a binding affinity to any eligible receiver and is called “unicasting”.

The unicasting protocol captures the notion of mobile processes whose open input ports are bound to senders only at message receiving time and corresponds to the chemical abstract machine model (CHAM) whose "send" and "receive" instructions are viewed as ions of opposite polarity, and whose reductions are the coming together of a positive and negative ion. DNA computers (see section 1.5) can be viewed as

CHAMs with two kinds of ions (A-G and C-T) that act as binary digits of addresses of complementary pairs like CTCG and GAGC. The interpretation of DNA strings as codes for potential addresses of communication ports whose communication protocols mimic those of the pi calculus establishes a remarkable correspondence between biological computers and independently-developed models of concurrent interaction. The robustness of this model suggest that it might be a stable interaction protocol in future concurrent models of computing. Milner indicates in the title of his Turing lecture (CACM January 1993) that he views processes as "elements of interaction".

Both the Pi calculus nad interaction machines extend the traditional Church-Turing model of computation to interaction. The Pi calculus is an interactive extension of the lambda calculus while interaction machines are an interactive extension of Turing machines.

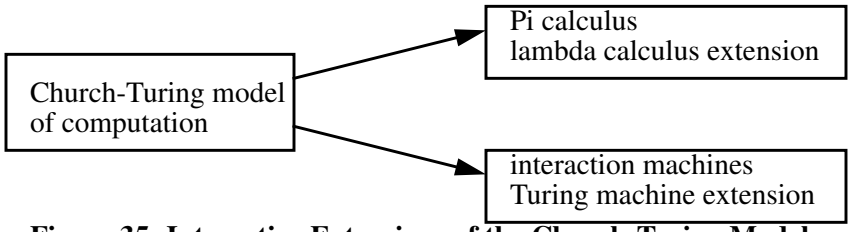


Figure 35: Interactive Extensions of the Church-Turing Model

The question of whether declarative and imperative extensions of the Church-Turing model have the same expressive power is open. The idea that the noninteractive equivalence between lambda calculus and Turing machines may give rise to a corresponding observational equivalence of interactive analogues for an appropriate notion of observational equivalence is plausible but should not be taken for granted. Equivalent expressiveness of noninteractive models may not carry over to the interactive world, since interactive imperative components may have more powerful compositional behavior than their declarative counterparts [WLM].

4.2 Transaction correctness as an interaction constraint

The correctness condition for transactions was first expressed in terms of serializability [BHG] and later by atomicity [LMWF], but may equivalently be expressed as "local noninteractiveness". Presenting transaction correctness as an "interaction constraint" that constrains distributed concurrent behavior to piecewise noninteractive algorithmic segments, presents transaction correctness in a new light as a condition for the applicability of local algorithmic analysis. Transactions constrain intractable interactive systems to locally algorithmic behavior. They determine a *harness constraint* that allows distributed, concurrent computation to be broken down into algorithmically tractable noninteractive segments.

Classical concurrency control [BHG] expresses correctness for transactions by serializability: the effect of executing transactions must be equivalent to their execution in a serial order. This definition views serial order as a normal form for a much larger class of nonserial executions. Since equivalence for concurrent processes is notoriously difficult to define, serializability does not provide a complete decision procedure for correctness, but it does allow some distributed, concurrent computations to be proved correct by reducing them to serial computations. Since Serializability is dependent on execution-time properties, this view of correctness encourages an implementation-dependent conceptual perspective.

Transaction correctness can be more abstractly expressed in terms of atomicity: the transaction executes as if atomically without interruption by other transactions and the originator of the transaction is informed of completion or abortion of the transaction. Atomicity is a local constraint on individual transactions that in principle allows local testing for atomicity, and encourages an abstract view of transaction semantics in terms of an interface with users rather than with a scheduler. But in practice, testing of atomicity for all possible forms of interaction is unrealistic and demonstrating an equivalent serial execution is usually the most practical way of demonstrating atomicity.

Transaction correctness can also be expressed as noninteractiveness: equivalence to a noninteractive

computation dependent only on the initial input and system state. The insight that atomicity corresponds to local noninteractiveness is in itself fairly trivial, but allows transaction correctness to be viewed as a guarantee of local algorithmicity expressible by algorithmic tools like pre- and post-conditions. It expresses the essential property of transactions as a temporally local harness constraint directly related to the requirement that distributed systems have locally algorithmic subcomputations whose granularity is programmer defined. The requirement of noninteractiveness for transactions is seen as a dynamic generalization of static noninteractiveness that constrain Turing-machine tapes to tractable (computable) behavior.

Since atomicity is not directly testable, correctness of transactions is in practice shown by equivalence to a serial implementation [LMWF]. However, atomicity determines a broader notion of equivalence than serializability, including nested and typed transaction systems. Noninteractiveness provides an alternative abstract criterion directly related to the goal of to be modeling transactions algorithmically as closed noninteractive computation. Atomicity is a binary all-or-nothing concept while interactiveness is incremental: relaxing noninteractiveness by introducing controlled interaction is easier to contemplate than introducing a little nonatomicity. Computationally tractable ways of relaxing the correctness condition to introduce cooperative transactions can be naturally expressed, allowing strictly noninteractive correctness conditions to be extended to correctness that allows limited forms of interaction (figure 36).

serializability: implementation-dependent interface with a scheduler, global condition
operational, implementation-dependent semantics of correctness
atomicity: implementation-independent interface with a client, local constraint
applicable to nested and typed transaction models, broader notion of equivalence
noninteractiveness: harness constraint on interaction, direct relation to incremental algorithmicity
can be incrementally relaxed to broader notion of correctness allowing controlled interaction

Figure 36: Alternative Correctness Conditions for Transactions

Transactions control interaction by imposing a programmer-defined granularity of atomicity on distributed, concurrent systems. This *temporal modularity* complements (is dual to) the *spatial modularity* of software components: temporal atomicity determines a scope of temporal abstraction that is the counterpart of a scope of spatial abstraction for components. The assumption that operations on objects are atomic and "as if instantaneous" holds automatically for sequential object-oriented languages. However, two procedures P1, P2 arriving at different input ports of a concurrent object may have nonserializable behavior corresponding to an arbitrary interleaving of their atomic operations. Though this behavior could in principle be modeled by serial execution of lower-granularity atomic operations, such changes in the level of abstraction is precluded by observability-driven rules of modeling. Nonserializability is avoided by requiring the procedures P1 and P2 to be atomic transactions. In this example the scope of the object abstraction determines a derivative temporal granularity of abstraction. In general, transactions may span multiple objects and the level of granularity of temporal abstraction is determined by the application rather than by the scope of object abstractions.

The all-or-nothing (commit/abort) property of transactions guarantees that failure by transactions is detectable and only completed transactions have a permanent effect on the computation. Transactions have a fail-stop property that guarantees detection of failures so that actions to handle failure can be taken. Thus transaction systems combine locally algorithmic behavior with failure detection.

Transactions pay a price in efficiency of execution and in expressiveness for their piecewise algorithmicity and fail-stop properties. The temporal granularity of transactions, controlled by the programmer, determines the level of concurrency. Serializability may be too restrictive for certain kinds of computation, such as cooperative access by many agents to a shared data structure, but it provides a powerful mechanism for reducing actions of a concurrent interactive world to equivalent actions of a formalizable noninteractive world. Serializable transactions are a dominant model of concurrent programming for reasons similar to the dominance of Turing machines for sequential programming: they constrain concurrency so it retains much of its power but is tractable. The trade-off between expressiveness and tractability can be

controlled by the programmer according to the needs of the application.

Input-output (I/O) automata [LMWF] may be used to provide an operational semantics for transactions by simulating their creation, termination, commitment, and abortion. They have internal and output actions under the control of the automaton and nonblocking input actions under the control of the environment that provide the expressive power of interaction machines I/O automata may be nondeterministic and have a composition operation for communicating automata that broadcasts output operations instantaneously to all similarly-named input operations. The external operations of an automaton consists of all output operations and those input operations not named by any output operations. Input operations are viewed as external only if they are not associated with any input operation, and a collection of automata is closed if it has no external input operations. I/O automata modeling a transaction system are generally closed. the behavior of an I/O automaton is defined to be its sequences of external (input and output) actions (behavior traces). An automaton A is said to implement B if its set of behavior traces of A is a subset of those of B: A implements B means that the behavior of A is consistent with the behavior constraints imposed by B: not that A captures the complete behavior of B.

The correctness condition for transactions is defined [LMWF] by a "serial system" that provides a baseline for defining correct serial behaviors that in turn serve to define a larger set of "equivalent" nonserial behaviors. Relevant behaviors of the transaction system are modeled by I/O automata:

transaction automata have create, create child, request and report commit, and abort operations
objects accessed by automata create subtransactions for every operation on the object
the scheduler is an automaton that manages serial scheduling of transactions and object accesses

Atomicity is defined by equivalence to a serial executions specified by the serial scheduler, refining classical serializability by admitting a larger class of serial behaviors, including nested transactions and time-stamped as well as order constrained specifications of behavior. The equivalence transformations further refine classical serializability by allowing a larger class of equivalent behaviors, including equivalence for replicated implementations. By focusing on atomicity rather than serializability as the correctness condition, correctness is defined in terms of visibility at the user interface, allowing the class of correct implementations to be broadened to include schedules that give the appearance of serial (atomic) execution at the user interface though they are not internally serializable. For example, internal record-keeping operations that do not affect the result are excused from the requirement of serial execution by a notion of "equieffectiveness" that captures the notion of "full abstraction". Though atomicity is ultimately defined in terms of serializability, it provides conceptually higher-level criteria for defining the level at which serializability should be considered.

4.3 Time, asynchrony, and consensus

Algorithmic time is represented as a discrete sequence of execution events whose metric (complexity) is simply the number of elements in the sequence. Interactive time has a much more complex semantics. Trace models of interactive systems represent time as discrete sequences of instantaneously occurring events with no notion of duration either between or during events. The application semantics of object execution may, however, attach significance both to intervals between events, as in deposit and withdraw events of a bank account (section 4.2) and to duration of events that require substantial algorithm execution time. The assumption of instantaneous occurrence is unrealistic when algorithmic response time is too long, when a component is called upon to execute too many requests simultaneously, or when a component delegates execution to another component. Traces do not provide an accurate or complete model of time in discrete distributed systems, just as linear Newtonian time does not provide a complete model of continuous distributed time in the physical world.

Distributed systems that associate a time line with each component employ a "calculus of time" that describes interaction in terms of composition of frames of reference associated with threads of each component. However, components with multiple interfaces do not permit the progress of a system to be decomposed into a collection of interacting time lines. Moreover, even systems with a global notion of time may be undescribable algorithmically because time sequences may be generated by oracles or natural

processes. Though interactive time is perceived as a global flow, the effect of time on multithreaded, interactive, distributed systems cannot in general be described by a simple calculus or logic of linear sequences. Algorithms are time-independent transformations with a relative inner notion of time but no mechanism for keeping track of the passage of external real time.
Threads have a relative (algorithmic) notion of time which was extended by Lamport [La2] to a distributed causal ordering such that event2 is (potentially) caused by event1 if there is a chain of local steps connected by message links from event1 to event2. Interaction machines can model time in the environment by *synchronous interaction machines* with nonblocking input actions or by *asynchronous interaction machines* with blocking input actions. This simple difference in the power of input actions makes a fundamental difference in the relation between machines and their clients. Once interaction is admitted as a legitimate form of behavior, it becomes legitimate to classify interaction machines by the forms of external interaction they can model. Asynchronous interaction machines appear to be more powerful than synchronous machines because they can express richer behavior in the environment. Asynchronous interaction machines can be shown also to have richer internal behavior that synchronous machines because they have nondeterministic and even "chaotic" behavior [We1].

Einstein's special theory of relativity is so called because time is relative to a frame of reference: simultaneity is subjective (relative) while causality is objective (invariant) for frames of reference. Though relativistic effects are negligible for distributed systems (unless distributed across spaceships), the principle that time is relative to a frame of reference with a subjective notion of simultaneity and an objective notion of causality still applies. Drift of physical clocks and unpredictability of message transmission times makes distributed synchronization impossible. Distributed systems with autonomously executing components have no notion of global time, though they can agree on a common approximate time by clock synchronization algorithms.

The consensus problem, defined as the problem of n processes communicating through shared objects agreeing on a common action, is a litmus test for adequacy of a distributed system to compute correctly in the presence of failures, since being able to solve consensus implies solvability of any other agreement problem. The conditions on a distributed system for the consensus problem to be solvable as well as the complexity of consensus (the number of steps required to reach agreement) have been extensively studied. Ability to solve the consensus problem implies that the system can function correctly even in the presence of malicious (Byzantine) failures. If it is assumed that malicious failures do not occur; for example that all component failures are recognizable by other components and simply cause components requesting services of a failed component to issue an alternative request for service, then consensus is not needed and weaker requirements on agreement among components are sufficient. The classification of failures by the severity of the remedial protocols needed to deal with them is important in determining acceptable modes of interaction. Transactions are tractable not only because of their local noninteractiveness but also because of their guaranteed fail-soft failure property. Malicious failure requires more complex modes of correct interaction explored in consensus models of computation.

4.4 Abstraction and transparency

Abstraction is an interdisciplinary tool of thought for simplifying situations or processes by focusing on relevant attributes and ignoring (hiding) irrelevant ones. Distributed abstraction, called transparency in distributed systems to focus on hiding complexity rather than revealing functionality, deals with an entirely different set of properties from the abstractions of procedure and object-oriented programming. Access and location transparency hide network structure, allowing the user to treat the system as though it were not distributed. Concurrency transparency hides internal concurrency, allowing the user to treat the system as though it were sequential. Performance transparency, which hides the execution time and resource utilization of the system, is a form of abstraction common to both algorithmic and distributed abstraction. Failure transparency, which allows systems to continue functioning in the presence of failure, is the most fundamental form of distributed transparency, allowing robust reliability in the presence of failures. Failures of systems may be classified, with crash failures being the easiest to detect and deal with. Message omission

is the next easiest to deal with, followed by fail-stop failures that are detectable. The hardest failures are malicious (Byzantine) failures. These forms of transparency are harder to model and implement than procedure and data abstraction, which hide the implementation of actions and data representation.

Abstractions may be classified by their goals and the structure of their models of computation. The high-level distinction between declarative and interactive abstraction can be refined by considering abstraction modes within each of these categories. Turing showed that the declarative abstraction paradigm could be expanded to include the imperative paradigm, providing the impetus to define computers science as the study of declarative and imperative abstraction mechanisms. Interactive abstractions, however, provide a richer and more varied set of entirely new abstractions whose models and classification categories we are only beginning to understand. The different kinds of transparency and the analysis of different kinds of failure transparency form the beginnings of such a classification.

4.5 Distributed programming languages

Programming languages were initially perceived as dealing entirely with algorithmic abstractions (procedures), but the object-oriented paradigm has recognized that they need to deal with interactive abstractions. Distributed programming languages must deal with algorithmic abstractions for inner computation and interactive abstractions for communication and coordination. Coordination languages like Linda that separate inner computation primitives from interaction primitives can be very simple. Linda's repertoire of interaction primitives consists of two output and two input instructions that allow processes to communicate with each other through a shared "blackboard" called a tuple space. Processes send data to the tuple space by the output command "out(t)" and can also send "live process tuples" that turn into data when evaluation completes by the command "eval(t)". The can retrieve data from the tuple space by the command "in(t)" which removes a tuple associatively matched by t or "rd(t)" that copies the associatively-matched tuple.

These commands determine a blackboard architecture with a set of primitives for communication between processes and the blackboard compatible with many programming languages, leading to systems like C-Linda that connect Linda's communication primitives with processes whose algorithms are specified in C. Linda's independent (orthogonal) specification of interaction primitives for communication and coordination provides a framework for separate design and analysis of interaction and algorithm sublanguages as well as modular plugging in of interaction sublanguages to multiple algorithm sublanguages. Linda's processes cannot interchange messages directly, communicating only indirectly through the tuple space. Though Linda's primitives for coordination are simplistic, it points the way to separate specification of interactive coordination primitives and algorithmic computation primitives. This separation in principle allows multiple coordination languages to be plugged in to a given programming language just as it allows multiple programming language to be plugged in to a given coordination language.

Hermes is a strongly-distributed language whose processes require all information about external resources, including the name of the creating process, to be dynamically transmitted as message data. It places strong statically checkable type constraints on receive ports and associated matching send instructions associated with types communication channels. It strengthens the notion of type, introducing "typestates" which allow checking for properties like initialization that may vary for different occurrences of a variable though they are invariant for any given occurrence. Sending ports have a type that requires sent messages to conform to both the behavioral type and dynamically determined typestate of the receiver. These notions enhance type safety in distributed environments by allowing temporal interaction invariants as well as function invariants to be checked at compile time.

Though Hermes is too pure to be widely accepted as a practical programming language its treatment of type safety in distributed environments is both technically and philosophically significant. The designers of Hermes have extended the notion of statically checkable invariants to protocols that express the order in which operations of an interface are executed. The extended notion of interface and channel compatibility requires not only typed behavior compatibility but also typestate and protocol compatibility. Such an extension of checkable invariants from algorithmic to interactive invariants is a natural consequence of

extending models of computation to interactive (distributed) computation.

4.6 Distributed operating systems

Operating systems are primarily concerned with the management of interactive (reactive) resources by interactive abstractions. They encapsulate physical resources of a computing system such as processors, memory, peripherals, and communication channels by resource managers called servers. They present physical resources to the user as abstract logical resources like names, channels, processes, hiding physical processor and network structure. Names assigned to memory resources correspond only indirectly to physical memory in which the information is stored, particularly in the case of replicated information. The gap between abstractions and their implementation is much greater for distributed than for nondistributed operating systems: for example, failure and replication transparency hide forms of complexity that have no analog for nondistributed systems.

Abstractions for distributed operating systems include not only resource but also client abstractions. They must capture both performance and communication requirements of clients. Models of authentication and authorization [La1] model human clients as "principals" that make requests to "request interpreters" that communicate with objects. Authentication formalizes assertions about principals P making statements S so that proving "P says S" corresponds to authenticating that P was the originator of the statement or request S. Requests are granted only if the channel on which it is received "speaks for" a principal whose credentials can be looked up by the receiver in a trusted database or certified by a trusted certification authority. This model provides an abstract view of clients as principals that make statements and makes use of system abstractions like channels, illustrating that useful abstractions of clients that interact with a distributed operating system can be very simple.

The client-server paradigm models clients as making requests satisfying the interface expectations of a server. Sequential paradigms guarantee that servers process at most one request at a time, while concurrent systems require synchronization to ensure that requests are processed sequentially. Requests can be viewed behaviorally as procedure calls in both distributed and nondistributed systems. But they hide potentially complex access mechanisms such as searching along a chain of links to implement inheritance, and communication protocols that implement remote procedure calls. The client-server model generally assumes that requests by clients are handled sequentially by nonoverlapping execution controlled by servers rather than by real-time deadlines imposed by clients. The client and resource abstractions of the client server paradigm are more detailed and also very different from those authentication paradigms, though both involve models of clients that share an interactive service.

Distributed systems generally have a layered structure, as illustrated by the OSI reference model. whose layers include a lowest-level physical layer and data link, network, transport, session, presentation, and application layers. Each layer represents the same communication information at a different level of abstraction by physical signals, bit streams, data frames, network packets, logical messages, encrypted dialog, and semantic phrases. Each layer has an associated conceptual compiler implementation problem: the layers add an additional dimension of structure to traditional system programming models. Layered models that represent the same abstract structure at different levels of abstraction are common in computers: successive phases of the software life cycle (section 4.1) may also be viewed as layered representations of the same abstraction, where the layers represent different stages of development rather than different levels of information abstraction.

5. Software Engineering Models

Software engineering emerged in the 1960s because algorithms proved to be an insufficient basis for a technology of software systems. Software systems are more accurately described by interactive, embedded processes rather than by algorithms.

Programming in the large (PIL) was observed to be qualitatively different from programming in the small (PIS) as early as the 1960s, but the nature of the difference was not well understood. In section 5.1. we claim that the difference between PIL and PIS is precisely that between interactive and algorithmic

computation. This provides a precise characterization of PIL by interaction machines, expresses the qualitative differences between PIL and PIS, and implies the irreducibility of PIL to PIS. It elevates PIL into an independent first-class area of study with its own models and conceptual frameworks, liberating practitioners from the burden of having to justify their models in terms of algorithmic notions. Expressing PIL in terms of algorithmic concepts of PIS is not merely hard but impossible. Moreover, such an attempt to fit a round peg into a square hole is counterproductive, directing research away from potentially fruitful directions into sterile areas.

Life-cycle models aim to specify software products at different levels of abstraction corresponding to different stages of development to establish traceability relations so that properties specified at one level can be identified with corresponding properties at other levels. In section 5.2 we characterize the relation between process models of the life cycle and product models of system structure as a pragmatic difference between human clients whose goals are development and evolution and computer clients whose goals are reliable and efficient execution. The noninteractive waterfall model of software development has been superseded by the interactive spiral model in recignition of the fact that software development is an inherently interactive process. Both the software system life cycle and software execution are both interactive processes modeled by interaction. More precisely, software development is a process engaged in by humans to produce a product that is interpreted as a process when executed on computers. Whether a program is a product or a process depends on its pragmatics of interpretation.

software development is modeled as a process for both algorithms and interactive processes
software systems extends computing from algorithms to interactive processes

Object-based models provide a framework for similarly-structured specifications at different levels of abstraction, and have a natural, reusable, and adaptable modularization, allowing both process models of the life-cycle and product models of programs to be modeled by the same interaction paradigm.

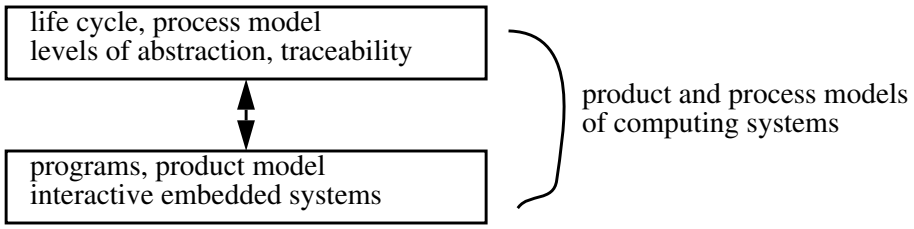


Figure 37: Uniform Modeling Paradigm for Life Cycle and Program Models

Section 5.3 examines the limitations of trace models of the observable behavior of objects, while section 5.4 considers software design and use-case models that provide a harness for constraining (harnessing) the behavior of interactive systems to tractable subsets of behavior. The object modeling technique OMT has a three-level structure: an object model that describes the static relations among objects, a dynamic model that describes interaction histories and use cases for systems of objects, and a functional model that captures the functional behavior of operations serving as a canonical representative of object-design methods. .

Work on design patterns and frameworks in the 1990s may be viewed as an attempt to develop interactive principles of structured object-oriented programming that parallel those of structured procedure oriented programming. The need for design patterms is a direct consequence of the inadequacy of algorithmic techniques in designing interactive systems. Design patterns are reusable interactive building blocks of flexible granularity: they are the interactive analog of procedures.

Autonomy, heterogeneity, and interoperability for very large systems are another inherently interactive application area whose relatively unstructured design methods are due to inherent difficulties and lack of understanding of design techniques for interactive systems. Document engineering is viewed as a generalization of software engineering whose techniques for managing interaction of large long-lived documents

are similar to those for managing interaction of large long-lived programs.

5.1 Programming in the large

Programming in the large (PIL) is a misnomer, since largeness is neither a sufficient nor a necessary condition for a program to be a PIL system. A program with one million assignment statements or a case statement with one million cases is not a PIL program. PIL deals with interactive (reactive, embedded) systems like airline reservation or banking systems that may interact with an autonomous environment, possibly in real time. The criterion of interacting with an external environment is more accurate than mere largeness in distinguishing PIL systems from algorithms. Small interactive systems with only hundreds of instruction are PIL systems while pure algorithms with millions of instructions are not.

Algorithmic problems may be precisely defined by a formal specification, are implemented by programs whose correctness may in principle be proved, and have a precise notion of complexity defined by a size parameter. Interactive problems such as "the airline reservation problem" are like algorithmic problems in that they have instances implemented by programs. But they have no complete specification or precise notion of correctness or complexity and their requirements differ qualitatively from algorithms.

The notion "algorithmic problem" has been carefully defined in the literature of computational complexity [GJ] as a general question whose complexity can be captured by a quantitative parameter. Interactive problems like the "airline reservation problem" are also general questions, though the complexity and domain of variation cannot be precisely expressed by a numerical parameter:

algorithmic problem: *general question with a quantitative complexity parameter*
sorting problem, traveling salesman problem, precise problem size
each value of the complexity parameter determines an instance of the problem

interactive problem: *general question with a qualitative notion of complexity*
airline reservation systems with increasingly ambitious requirements, imprecise notion of size
instances of software problems can vary along many dimensions, imprecise notion of problem

Instances of an algorithmic problem are strongly similar to each other, distinguished by one-dimensional variation of the complexity parameter. The variability among instances of an interactive problem is much greater. The airline reservation problem cannot be tidily specified by one or more parameters: it has fuzzy boundaries but we can recognize an instance of an airline reservation problem when we see one. Interactive problems are in this respect like everyday objects: the concept "table" cannot be precisely defined by a set of attributes but we know one when we see one. The class of airline reservation systems, like the class of tables, is a "natural kind", that cannot be precisely defined by a set of attributes. However, the airline reservation problem can be approximately characterized by a set of use cases that it should support, just as the concept "table" can be approximated by attributes like number of legs and type of surface.

Behavior of algorithms is specified by a "sales contract" whose client supplies a value satisfying a precondition and is guaranteed a result satisfying a postcondition. Systems determine a behavior contract over time like a marriage contract rather than a sales contract. Resource requirements are measured by time and space complexity for algorithms and by life-cycle cost for software systems. Whereas correctness is more important than efficiency for algorithms, cost is often more important than correctness for software systems. Algorithmic problems are formulated in an ideal world where behavior requirements dominate resource requirements, while resource requirements dominate behavior requirements in the real world of software systems (figure 38).

PIL differs from programming in the small (PIS) in terms of both products and processes. PIL products are interactive rather than algorithmic: they determine a marriage contract over time with the user rather than a sales contract specified by outputs for given inputs. PIL processes span the life cycle while PIS processes focus on the efficiency (computational complexity) of execution. Both have a notion of cost, but algorithmic cost is measured by execution-time resources while life-cycle costs span a longer time horizon

with human as well as computer resources.

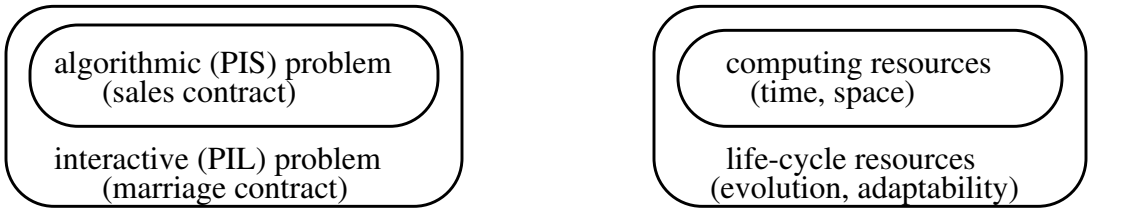


Figure 38: Requirements for Algorithmic and Interactive Problems

Whereas algorithms have correctness as their primary (hard) requirement and efficiency as a secondary (soft) requirement, software systems have life-cycle cost as the primary requirement and may sacrifice correctness or reliability to satisfy limitations of cost. Software complexity is a qualitatively-elusive analog of algorithmic complexity. Software scalability is a qualitative analog of algorithmic tractability. There are many other qualitative PIL analogs of quantitative PIS notions. However, in spite of the qualitative fuzziness of PIL, systematic methods of software design and life-cycle management are emerging.

5.2 Life-cycle models of the software process

Life cycle models shift the focus of modeling from execution-time performance of software products to the process of software development and evolution. They view software systems from the pragmatic perspective of the software developer rather than the computing agent:

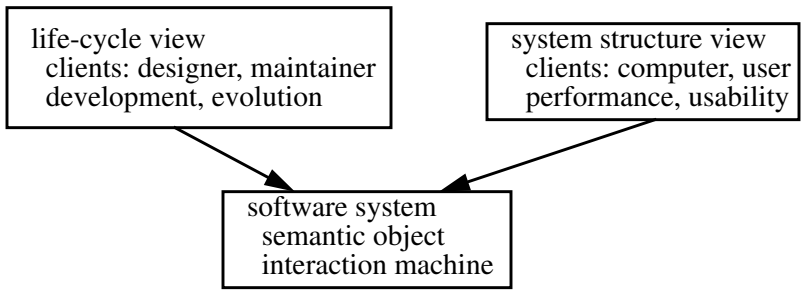


Figure 39: Pragmatic Perspectives of Software Systems

The waterfall model is an essentially noninteractive life-cycle model, identifying a sequence of static levels of modeling during the development of software products, including requirements, analysis, design, implementation, testing, and operations and maintenance. Partitioning the life cycle into static noninteractive phases is simplistic but nevertheless useful in classifying activities and products of different phases of the software process. The spiral model views the relation among life-cycle phases as interactive: requiring wach life-cycle phase to be modeled by an interaction machine rather than an algorithm.. Life-cycle models must coordinate representations at different levels, guaranteeing that they are compatible, that changes at one level are properly reflected at other levels, and that requirements and properties at any given level are tracable at other levels.

Each level of life-cycle modeling has a different purpose and class of clients. The requirements specification is a contract with the end user, the analysis model supports conceptual analysis of ideal models, while the design model provides a bridge from conceptual analysis to implementation. The testing model validates the implementation against the expectations of the end user, while the operations and maintenance phase imposes requirements of adaptability to harnesses earlier models for a new set of uses.

Models are pragmatically judged by their adequacy for a set of uses: for example a design or implementation model is judged by different standards if it is used just for creating a once-off product than if it is

used as a basis for maintenance and enhancement. Use case models [Ja] distinguish behaviorally similar systems by differences in their intended uses: the waterfall model views design and implementation models primarily in terms of product behavior, while the spiral model explicitly considers life-cycle modes of use for each level of modeling. Adaptability requires each model to be viewed as a process in time for which modification and enhancement are important modes of use.

Life cycles embed software products in a development process that requires coordinated development and enhancement of behaviorally equivalent models of a software product at different levels of abstraction. This process is similar for algorithms and interactive systems, but software engineering (programming in the large) is primarily concerned with interactive products whose specification and mode of use are more complex than for algorithms. When both the product and the life cycle process are interactive, object-oriented models have a great advantage in providing a uniform framework for modeling the different levels of product abstraction as well as the life-cycle process itself.

5.3 Observable behavior of objects

The observable behavior of objects is described by *interaction histories* that specify messages received and sent by an object over time. Objects in object-oriented language slike Smalltalk and C++ generally have *sequential interaction histories* consisting of algorithmic episodes separated by periods of inactivity. Algorithm execution is assumed to occur instantaneously, the duration of periods of inactivity between algorithmic episodes is assumed to be insignificant, and the order of arrival of messages is assumed to uniquely specify observable behavior.

Even sequential interaction histories can give rise to nonalgorithmic behavior not specifiable by any computable function, since the infinite sequence of input messages of an object need not be recursively enumerable. However, algorithms do not in general execute instantaneously, the duration of periods of inactivity may be significant, there may be hidden activity within object, and interaction histories may be nonsequential, consisting of inherently concurrent overlapping messages. The design space of observable behavior, which was discussed in section 1.6 in the context of object interaction machines, is reexamined here from the viewpoint of observable behavior.

Both algorithms and objects determine a contract with clients validated by observable behavior. As pointed out at the beginning, algorithms specify a sales contract that yields a value determined by a post-condition for every argument of a precondition, while objects specify a contract over time that is like a marriage contract. Algorithms have a time-independent observable behavior while object behavior has both transformational (functional) and temporal properties:

object behavior = transformational + temporal properties

We revisit the bank account example of section 1.6. to consider how the observable behavior of bank accounts may be described:

object *bank-account*
interface
procedure *deposit (argument: Money);*
procedure *withdraw (argument: Money);*
body *that includes procedure bodies and a hidden state (bank balance) shared by nonlocal variables of the deposit and withdraw operations*

The observable behavior of bank-account objects with deposit and withdraw operations can, as a first approximation, be specified by sequential interaction histories called traces:

deposit(100), withdraw(50), withdraw(20), deposit(200), ...

Traces are limited in their ability to specify object behavior: they specify only relative time and not absolute time needed to compute interest. Interest computation is supported by time-stamped traces that anchor operations in absolute time:

deposit(100, timestamp); withdraw(50, timestamp) -- time-stamped operations

Event-triggered protocols specified by simple traces are adequate when the service provided by an object is time-independent, but time-triggered protocols modeled by time-stamped traces are essential in safety-critical applications with hard real-time constraints. These two paradigms of specifying observable behavior are associated with two entirely different research communities (the simulation and real time communities). Converting an ordered into a time-stamped trace may overspecify it by prematurely forcing relative ordering to become anchored in absolute time. Conversely, converting a time-stamped into an ordered trace makes it less specific and incapable of modeling operations dependent on absolute time.

Strategies for specifying interaction histories include the following:

traces: *ordered sequences of events occurring in the lifetime of an object*
time-stamped traces*: sequences of events anchored in absolute time*
use cases: *partial set of interaction histories for a particular interface or mode of use*
fully abstract behavior: *specification of all possible observable behaviors (not formalizable)*

The specification of observable object behavior is already intractable even for sequential interaction histories, and is even more complex for distributed interaction histories. When bank account behavior is scaled up to distributed automatic teller machines with simultaneous (overlapping) withdrawals at multiple sites, the behavior becomes nonserializable unless interaction is constrained by transactions.

History described as a linear progression of events does not do justice to reality, since object history, just as national or world history, may consist of a jumble of overlapping concurrent events. Object interaction histories are generally approximated by linear interaction sequences, since concurrent interaction histories are too complex to design or implement. Concurrent histories can in principle be represented by notations like Petri nets, but useful concurrent interaction histories are difficult to specify in a formal notation. There are many open questions concerning object specification by interaction histories:

5.4 Object-based design and use-case analysis

Object-based design is concerned with the abstract description of object machines that cooperatively solve interactive problems. Both static object structure and dynamic object behavior must be described in an implementation-independent way. It turns out that mainstream object-design methods of software engineering have a remarkably stable structure.

The object modeling technique OMT [Ru] can serve as a canonical representative of object-design methods. It has a three-level structure: an object model that describes the static relations among objects, a dynamic model that describes interaction histories for systems of objects, and a functional model that captures the functional behavior of operations. These three levels capture behavior at the system structure, object dynamics, and algorithm execution level.

object model: *describes the static computation space of all interactions*
nonformalizable static description of the complete software system
dynamic model: *describes interaction histories (scenarios) for systems of objects (inter-object dynamics)*
dynamic partial inter-object behavior at the level of interface behavior of interaction machines
functional model: *describes behavior of specific functions (intra-object dynamics)*
dynamic intra-object behavior at the level of algorithms

The relation to corresponding procedure models is indicated in indicated in Figure 40:

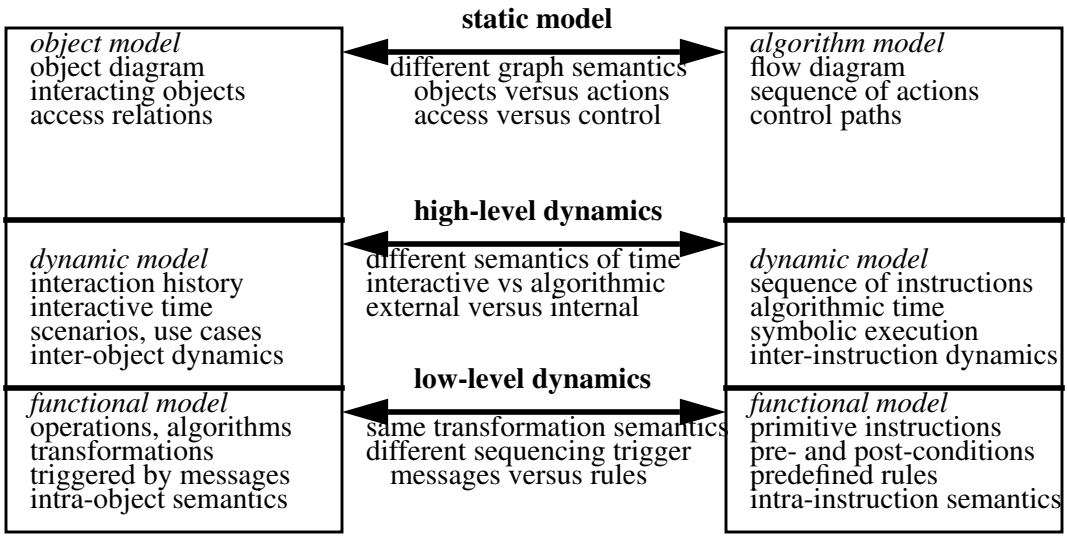


Figure 40: Comparison of Object Model and Algorithm Model

Designers express system behavior at the level of the object model, dynamic behavior of objects and of threads of computation spanning multiple objects by a dynamic model, and the behavior of algorithms by the functional model. The object model plays the major role during analysis and design, being created by establishing a correspondence between objects of the modeled domain and objects of the model. The dynamic model supports use cases, called scenarios in OMT, as a basis for testing that the system has desired behavior. The three-level model of OMT expresses static object semantics in its object model, dynamic operational semantics of specific modes of use in its dynamic model, and algorithmic behavior in its functional model:

OMT model = (static object semantics, dynamic modes of use (pragmatics), algorithmic behavior)

The pragmatic model (dynamic model) is concerned entirely with the pragmatics of interaction, the pragmatics of algorithm execution is ignored as too low level. OMT is a "second-order model" from the point of view of logic, being concerned almost entirely with patterns of algorithm execution and not with values computed by specific algorithms. However, it keeps things simple by considering only patterns that are sequences and that allow algorithms to be modeled as time-independent instantaneous transformations. The higher-level notion of a mode of use is not emphasized in OMT but can be expressed as a collection of similarly-structured scenarios (use cases). The idea of use cases as similarly-structured groups of scenarios is analogous to the algorithmic idea of grouping similarly-structured execution paths, though the criteria for grouping as well as the interpretation of grouped sequences is very different. The comparison of object models in chapter 16 of [Ja] suggests a convergence of techniques among object modeling techniques for which the three-level model of OMT may be regarded as the canonical form.

The use-case driven approach to object-oriented design [Ja] focuses on specification of the interactive interface between a system and its clients. The use-case model specifies intended modes of use of a system by agents (actors) that interact with it. Instances of modes of use are specified by *use cases*. The uses of a system can generally be specified by a small set of desired modes of use (interaction). For example, modes of use of an airline reservation system may include:

travel agents: *making reservations on behalf of clients*
passengers: *making direct reservations*
desk employees: *inquiring on behalf of travel agents, passengers, flight crews*
flight attendants: *use of reservation system to aid passengers during the flight itself*
accountants: *auditing and checking financial transactions*

system builders: *access to the system for modification and during emergencies*

Use case analysis may serve not only to specify behavior but also to constrain modes of interaction to a "safe" and predictable range of behavior. For example, the constraint that elevators carry a load of less than 2000 pounds can be thought of as a use-case constraint. Constraints guaranteeing safe usage of airline reservation systems cannot be locally specified, since nonlocal effects such as the breakdown of a distant computer or too many simultaneous users can cause a local deterioration or breakdown of services. The set of all possible behaviors of an interactive system is generally much greater than its set of intended uses (supplied behavior is greater than demanded behavior). By constraining system behavior to a specific set of use cases the intractably-rich supplied behavior of a system need not be completely modeled: only specific demanded behavior need be considered. For example, by considering only elevator behavior for loads of less than 2000 pounds we avoid considering the unpredictable and unspecifiable behavior complete behavior of elevators.

The transition from procedure-oriented to object-oriented programming requires a radical conceptual shift from the view of programs as sequences of actions (verbs) to the view of programs as collections of interacting nouns. Nouns are inherently more expressive than verbs in both natural and computer languages. Languages consisting entirely of verbs allow us to say "sit" and "stay" to a dog, but nouns are needed to realize the expressiveness of natural language. Nouns describe open persistent entities with unpredictable environment interactions, while verbs describe closed timeless transformations. Models of noun interaction include both theoretical models like the pi calculus and IO automata, and practical object-oriented language and design models. Models of object design suggest that the computational semantics of nouns may be modeled by a three-tier model whose three levels respectively capture static interaction among collections of nouns, dynamic interaction histories among nouns in real or conceptual time, and state transformations within nouns triggered by interactions.

5.5 Design patterns and frameworks

The need for design patterms is a direct consequence of the inadequacy of algorithmic techniques in designing interactive systems. Design patterns are reusable interactive building blocks of flexible granularity: they are the interactive analog of procedures. Whereas procedures are composed of instructions and can in turn be composed with other procedures to create larger procedures, interaction primitives (for example objects) are nonalgorithmic to start with and are noncompositional in the sense that interactive composition of two objects does not yield an object (it yields a subsystem with two objects that communicate through interaction protocols). Subsystems created by the composition of obects or interaction machines do not conform to any accepted notion of structure and are very hard to characterize, though they do determine subsystems that exhibit reusable regularities of interface behavior. The term pattern is used to denote reusable regularities of behavior exhibited by interactive subsystems created by the composition of interaction primitives..

Patterns in any domain of discourse are abstract properties that describe the domain by its regularities. Design patterns are regularities of designs, while software design patterns are regularities of the product and/or process of software design. Algorithms may be designed by a process of refinement that decomposes high-level declarative or imperative specifications into components using techniques like structured programming. Design for interactive systems cannot be described by tree-structured refinement: structured programming does not generalize from procedures to objects. Whereas algorithms can be designed by top-down refinement of a specification, interactive design starts from a bottom-up description of the interaction environment. [GHJV] describes reusable design patterns for interactive object-oriented software by four components:

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 high-level components: a *model* that represents the application object, multiple *views* that capture syntactic representations and a *controller* that defines the mode of execution. This three-part decomposition parallels that of semantics, syntax, and pragmatics in section 1:

MVC = (semantic model, syntactic views, pragmatic controller)

Though the MVC paradigm is a high-level design pattern its granularity is generally considered too great for inclusion in a design catalog. [GHJV] describe MVC in terms of component patterns: a pattern to describe one-to-many sharing of a resource (object) whose changes automatically affect many dependent objects, a pattern to describe composite structures by a nested tree of components, and a pattern that allows run-time selection among multiple functionally equivalent implementations of the same algorithm.

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 that help in understanding 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 user interfaces (apis)that may contain implementations of several patterns, and 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 which 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 Gamma, Helms, Johnson, and Vlissides (Addison Wesley 1994) include patterns for object composition.

Computers have caused a change in the technology of producing, managing, and distributing documents comparable to that caused by the invention of printing. Authors control document production, have at their disposal powerful multimedia editing tools, can rapidly incorporate changes and manage document evolution, can incorporate the work of others into their document, and have powerful methods of mass distribution through e-mail and the World-Wide Web. The new technology supports new ways of working for groups of cooperating authors. Computer-supported cooperative work (CSCW) supports human cooperation in the development of documents. CSCW is in turn a form of coordination: coordination theory is concerned with both human and computer cooperation. New forms of organization are emerging because computers are changing the way people work together.

5.6 Autonomy, heterogeneity, and interoperability

Autonomy 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* that localizes access to data. Asynchrony allows components to execute autonomously in time. Heterogeneous interfaces are still another form of autonomy (represnetation autonomy) that requires translation (transduction) mechanisms to be interposed between senders and receivers.

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

Autonomy are natural in a world where programmers develop autonomous software applications 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. One solution, adopted in [WWC], is to introduce a "megaprogramming language" that regulates interaction among autonomous modules (megamodules) and where necessary translates 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 termonology (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.

Interaction among systems of heterogeneous independently developed systems of autonomous compoents is referred to as interoperation. Interoperation is a major software challenge in the mid 1990s: systems under development include OMG's CORBA, Microsoft's COM/OLE, IBMs DSOM, and Sun's distributed object environment (DOE). Though a conceptual framework for interoperation does not yet exist, common principles are emerging that involve treating interfaces as first-class entities transformable by wrappers, mediators, and other interface management software. Interoperability can be realized by mediators and wrappers that interpose a mechanism for mediation between heterogeneous communicating components. Wrappers are unary operations on generated data streams (filters) that allow interfaces developed in one context of use to be adapted for other contexts. Mediators are binary (in general n-ary) operators on data streams that coordinate communication among collections of autonomous components by interposing an extra layer of software between clients and servers.

Interoperability is a form of software scalability: wrappers and mediators distribute the management software for heterogeneous systems so that management complexity grows only slowly (linearly) with systems size. Architectures for interoperation have three logical layers: an application layer that includes graphical user interfaces, a shared services layer of "middleware" for gluing together interoperating software components, and a data layer of databases services. Interoperation requires agreement among heterogeneous components at a variety of levels including data representation, object models, interfaces, and communication protocols. Extensibility and evolution can be facilitated by making architectural structure explicit through techniques like reflection. Software architectures for gluing together (composing) heterogeneous components require a distinctive technology of modeling whose outlines are beginning to emerge.

The World-Wide Web is probably the most successful example of widespread interoperation. Its success is due to its simple architecture and the hypertext markup language HTML which supports communication through universal resource locators (URLs), and typeless links. Markup languages like SGML for literary texts, HTML for World-Wide Web documents and VRML for virtual reality multimedia docu-

ments are much simpler than programming languages but achieve a high degree of interoperation for documents. Interoperation of object-oriented programs at the object level, supported by systems like CORBA and COM/OLE, is a much harder problem than interoperation of documents since interfaces consisting of operation signatures are more intricate than interfaces that are untyped links.

5.7 Document engineering and coordination

Progams are a specialized class of documents: all programs can be viewed as documents but documents include books, spreadsheets, and legal contracts that are not programs. The management of collections of programs in software engineering can be viewed as a specialization of the management of collections of documents. Document engineering, defined as the disciplined creation and management of large multimedia documents, can be viewed as a generalization of software engineering.

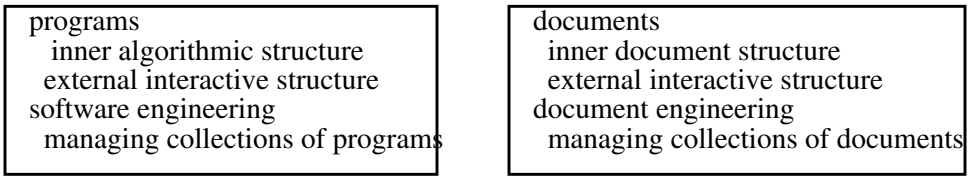


Figure 41: Document Engineering as a Generalization of Software Engineering

Programs have a specialized inner algorithmic structure, but their requirements of linking, coordination, and maintenance are similar to those of other classes of documents. The inner differences between algorithmic program strucutre and structure of other documents are substantial, but the differences between modes of interaction of programs and other documents are much smaller.

As computing matures, classes of documents like spreadsheets, two-dimensional multiuser interfaces, electronic books, and database repositories are becoming increasingly important The number of users of spreadsheets and word processors far exceeds the number of professional programmers. Computers augment our mental abilities by facilitating richer forms of self-expression just as physical machines augment physical abilities by harnessing physical forces. Though programs enhance our problem-solving power, self-expression is more directly realized through graphical user interfaces and multimedia documents.

As computer science matures, algorithms and programs may lose their place as the central focus of study in computer science departments. Algorithms will always retain their place as the circuit theory of computing, but may lose their place as the primary focus of study in first courses in computer science, just as circuits have become less central in electrical engineering. The study of communication, sharing, and coordination may come to be viewed as more central to computing than the study of algorithms.

6. Artificial Intelligence Models

The evolution of artificial intelligence from declarative (logic-based) to imperative (procedural) and interactive (agent-oriented) models has been analyzed and debated to a much greater extent than in software engineering. Knowledge representation focused on declarative models in the 1960s, procedural models in the 1970s, and is increasingly emphasizing interactive models in the 1990s [RN]. The AI agenda was set by formalists in the 1960s and much of the 1970s [Gr]. The symbol system hypothesis espoused by Newell and Simon and the development of domain-specific expert systems shifted the focus from declarative to imperative procedural models, while agent-oriented models and planners for embedded dynamical systems are focusing increasingly on interaction:

declarative (logic-oriented) -> imperative (procedure-oriented) -> interactive (agent-oriented)

The realization that interactive models are more expressive than declarative or imperative models has a profound effect on the conceptual framework of AI. It shifts the focus of modeling from logic, search, and algorithms to interactive mechanisms that derive their intelligence wholly or in part from the environment.

It destroys the argument that intelligence is expressible by Turing machines, and requires the question whether computation can express intelligence to be reexamined in the context of a more powerful interactive notion of computation. The argument that interaction machines can express intelligence is a good deal stronger than the argument that Turing machines express intelligence (section 6.7).

Artificial intelligence can be defined narrowly as an engineering discipline for building computational models that realize behavior judged by humans to be intelligent, but deals with conceptual as well as computational models of intelligence and with cognitive processes of thinking and understanding. Its subfields include knowledge representation, planning, learning, vision, and natural language understanding.

6.1 Knowledge representation

Knowledge representation (KR) systems represent knowledge in a knowledge base as an explicit (reified) thing so it can be used to perform intelligent tasks. Knowledge bases support queries by "ask" operations and insertion (knowledge acquisition) by "tell" operations. Knowledge can be explicitly represented by the predicate calculus, semantic networks, programs of traditional computing systems, and agents. Early KR systems represented knowledge propositionally in a declarative KR language and interpreted it by an inference engine for declarative reasoning. KR systems like the "blocks world" that represented knowledge imperatively (*procedurally)* were more practical for applications like expert system, just as in software engineering (SE).

However, the choice between declarative and procedural KR, which parallels that between declarative and imperative programming languages, is becoming a side issue because interactive KR systems that encapsulate declarative or procedural knowledge in interactive agents are becoming the standard for KR. The transition from toy problems to real systems has proceeded more successfully in SE than in KR, in part because AI was slow to abandon the declarative paradigm and even slower to adopt interactive agent-oriented representations. The conceptual gulf between algorithmic and interactive models is enormous, and the paradigm shift from neat mathematical or algorithmic models to ad hoc interactive models is more painful for conceptually motivated AI researchers than for pragmatic software engineers.

KR and SE models have similar goals in representing knowledge about the world in a form that can be effectively used for interactive problem solving and cannot be formalized for the same reason. While some concepts like is-a hierarchies are already shared by the KR and SE communities, there are many parallel models (for example semantic nets and object models) that could benefit from common terminology spanning KR and SE.

6.2 Intelligent agents

Agents perform tasks on behalf of someone or something else, functioning continuously and autonomously in an environment that may contain other agents. They encapsulate knowledge about the task they are performing, perceive their environment through sensors, and act on their environment through effectors. They are situated in an environment that may change dynamically and be imperfectly predictable. Agents have goals specified by a performance measure and are said to be rational if they act to maximize the probability of achieving that goal. In preforming its task the agent makes use of its encapsulated (built-in) knowledge, the sequence of inputs (perceptual sequence) it has received through its sensors, and its performance measure that defines its degree of success.

Agents whose actions depend on acquired knowledge (experience) are said to be autonomous. Autonomy enhances the ability to adapt to the environment, but agents must balance built-in cleverness with autonomous adaptability to perform significant tasks. Agents can be classified by the kinds of tasks they perform, the environment in which they operate, and the goals (performance measures) that drive the agent. Tasks may be classified by their degree of algorithmicity: tasks like solving simultaneous equations are inherently algorithmic, others, like selling cinema tickets, can be performed interactively with little algorithmic skill, while still others, like driving from San Francisco to New York, can be performed algorithmically by the use of a map or interactively by consulting road signs. Algorithms for a task generally have greater complexity than corresponding interactive procedures, since they must take into account all

possible contingencies in advance, while interactive procedures can deal with contingencies as they arise.

Russell and Norvig [RN] use the metaphor of building programs for intelligent agents as a unifying principle for the whole of AI. For example, planning is done by planning agents that construct plans to achieve goals and then execute them: note that planning is an interactive task that could not be expressed by a verb but is nicely captured by the nounlike construct of a planning agent. Once domain-independent principles for agent design are developed, domain dependent agent programs in subfields of AI can be defined. The identification "AI = agent-oriented programming" reflects that AI programs are interactive (reactive) rather than algorithmic. This characterization establishes a fundamental similarity between programs of AI and software engineering: agents in AI and objects are both embedded systems. Both AI agents and software objects are better modeled by interaction machines than Turing machines. IO automata are also an appropriate model: their input actions are realized by sensors and their output actions are realized by effectors. Agents, objects, and models of interaction have a common perceptual structure that captures empirical modeling and is more expressive than Turing machines.

6.3 Planning agents for dynamical systems

A planner is an agent for controlling or using actions of a second agent (dynamical system) that performs an application task: for example a travel planner that uses a transportation system. A dynamical system D may have both predictable behavior (airline schedules) and unpredictable effects (strikes and earthquakes). Dynamical systems are interactive agents whose behavior may be harnessed by input actions of a planning agent to realize desired performance expressed by a performance measure. Planners have a predictive model M(D) of the dynamical system and a measure of desired performance as input, and produce as output a set of plans (possible courses of action) to control or simply use system behavior. Planners that can observe and dynamically react to the actions of the dynamical system are *on-line interactive planners*, while planners that cannot are *off-line planners*. On-line plans, referred to as policies, can be dynamically changed in response to feedback. Planners P can be viewed as systems P(M(D), F) parameterized by the model M(D) of the dynamical system D, and a feedback stream F of observations (interactions) that can serve as a reality check on whether D's actual behavior conforms to M(D) and can dynamically modify the model M(D) and plan P to close the gap between the model and the reality (figure 42).

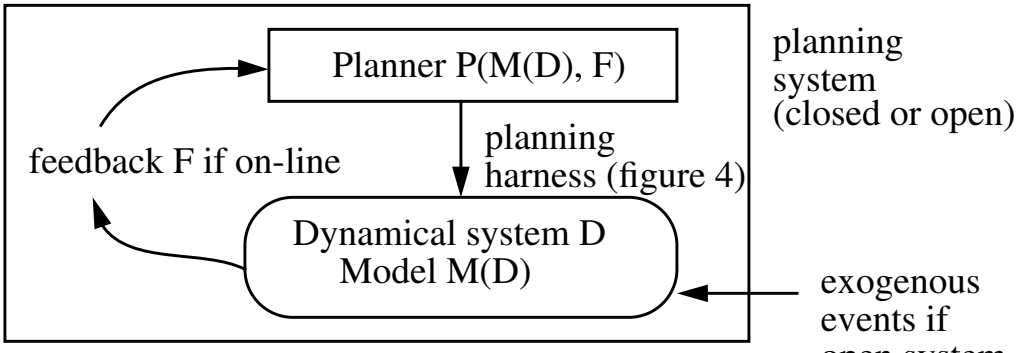


Figure 42: Planners with Feedback and Exogenous Events

Planning systems can be classified by whether the planner is a closed harnesses (section 1.4) that completely determines the behavior of D or an open harnesses that provides a nondeterministic or probabilistic approximation because of exogenous events not under the control of the planner. When D is an inherently open system with unpredictable exogenous inputs like earthquakes, then the planning system cannot guarantee behavior and is nondeterministic or probabilistic. Nondeterminism also occurs when D is deterministic but has behavior that is hidden or too complex to model, like the weather. From the point of view of the user of a planning system, it is irrelevant whether the approximation is due to inherent unpredictability or to practical limitations of modeling. Nondeterministic and probabilistic modeling techniques, whether due to incomplete knowledge or to inherent properties of dynamic behavior, are a technique for providing a

closed model for purposes of planning that avoids having to deal directly with questions of dynamic interaction and partial knowledge but introduces unpredictability. Planners may be classified by whether the model M is deterministic and by whether they are on line:

off-line planner, closed planning system
on-line planner closed planning system
off-line planner, open planning system
on-line planner, open planning system

Off-line deterministic planning models can in principle be analyzed entirely algorithmically, though such analysis may in practice turn out to be combinatorially intractable (NP-hard). On-line planning can in this case reduce complexity: travel plans that can be revised when flights are canceled or to take knowledge of earthquakes and hurricanes into account are combinatorially simpler than off-line plans which consider these contingencies in advance. Expanding the solution space from algorithms to interaction can reduce the need for algorithms and simplify the planning task by delaying the binding time of decisions. The interactive solution space not only reduces complexity but can also extend the expressive power, allowing the solution of problems that are not only intractable but algorithmically unsolvable. The problem of planning illustrates that interactive solution of problems can reduce the complexity of off-line algorithmic problems as well as solve problems that have no algorithmic solution.

Nondeterministic approximation introduces an additional dimension of analysis for planning problems that makes on-line feedback more important as a reality check, but may in principle be combined with off-line planning. The characterization of the planning problem S(P, D) by whether the model M(D) is precise or approximate and by whether planning is off-line or on-line provides a two-dimensional framework for classifying planning problems. The first dimension is closely related to algorithmic versus nonalgorithmic models of dynamical systems, while the second is related to algorithmic versus interactive models of the planning process. The planning problem illustrates the trade-offs between algorithmic and interactive modeling of systems of interactive agents for a concrete application domain. Dynamical systems are inherently open (interactive) whether or not the planning system of which they are a part is interactive. Planning is an exercise in taming inherently open dynamical systems so their analysis becomes manageable and can be analyzed as a problem of two-agent interaction of open versus closed subsystems.

The characterization of components of a system by whether they are open or closed tells us something fundamental about their qualitative behavior and their expressive power. For example, the design of off-line planners involves single-minded algorithmic considerations while on-line planners involve considerations of interaction. Dynamical systems are prototypical open systems whose study in control theory by differential equations models of continuous interaction machines provides a body of knowledge from which we can learn much about modeling discrete interaction machines. In the general case, differential equations can be turbulent or chaotic, but planners generally arrange for inputs that cause dynamical systems to be well-behaved. The discrete analog of dynamical control systems suggests that chaos and turbulence have discrete analogs when atomic operations cannot be serially executed [We1], but that such turbulence can be controlled by planners or less intrusively by transaction systems that guarantee serializability of atomic operations without commitment to any specific plan. Chaos can be avoided by guaranteeing serializable execution of events. Planners are in some respects more intrusive by requiring goal-directed execution of events, but may have to compromise on serializability by techniques like just-in-time algorithms that compromise on accuracy in order to meet internal as well as external time constraints.

6.4 Formal versus empirical (connectionist) models of intelligence

Formalists believe intelligence can be expressed entirely in the language of logic while their opponents (connectionists) believe that intelligent systems have nonlogical interactive properties. McCarthy advocates formalism, believing that logical reasoning at the metalevel by logic-based programs like the “advice taker” allows purely logical programs to realize learning and intelligence. Newell and Simon’s physical symbol system hypothesis that “a physical symbol system has the necessary and sufficient means for intelligent action” is generally considered to be formalist, though the nonlogical nature of interaction machines

suggests otherwise.The argument that intelligence is a purely logical or algorithmic process is much older than AI, being exemplified by Cartesian rationalism, formalist mathematics, cognitive behaviorism, and the Turing test. Godel incompleteness showed the inadequacy of formalism and contributed to a reaction against formalism, first in mathematics and the physical sciences and more recently in computer science and artificial intelligence. The formalist position was bolstered by the view that computing as represented by Turing machines is inherently rationalist. However, the empirical expressive power of interaction machines shows that computing machines are perfectly capable of empirical behavior and thereby knocks another nail in the coffin of purely formalist models of intelligence.

Logic is useful for reasoning about static worlds whose properties are completely specified in advance, but breaks down for empirical worlds whose properties are partially known and progressively revealed. The notion of reasoning as a process of drawing conclusions from assumptions remain valid for empirical worlds, but the notion of a fixed set of assumptions (axioms) that are monotonically augmented by theorems derived entirely from axioms does not. Logics that support weaker forms of nonmonotonic reasoning have been developed, but these systems abandon the study of static truth in a completely defined (closed) world. Empirical models abandon the notion of monotonic truth and completeness, while retaining the notions of derivability and soundness. According to Papert [Gr], formalist AI was dominant in the 1960s and 1970s in part because Minsky and Papert demonstrated the limitations of perceptron models, but parallel distributed-processing (interactive) research experienced a resurgence in the mid-1980s. Brooks [Br] presents arguments for the orthogonality of intelligence and reasoning by demonstrating the existence of mechanisms that behave intelligently without reasoning by harnessing the intelligence of the environment.

6.5 Intelligence without reason

Intelligent programs, just as people, may be characterized on a scale of algorithmicity versus interaction. At one end of the scale there are computer-intensive programs that, like autistic people, rely entirely on their inner resources. At the other extreme, intelligent behavior in ants, robots, and humans can be realized by “mindless” (stateless) stimulus-response patterns that respond in a Pavlovian fashion to data and intelligence in the environment.

The complex behavior of an ant on a beach in returning to its ant colony was discussed in section 1.2. The behavior of the ant “echoes” the complexity of the beach, which acts as an oracle (or natural process) whose nonalgorithmic input is transformed by the ant into behavior over time that is not algorithmically describable. Though the ant has minimal reasoning power, and can function only when local one-step goals can yield the eventual global goal, it is nevertheless capable of nonalgoithmic behavior. Even interactive identity machines that simply replicate inputs at the outputs have nonalgorithmic behavior when replicating nonalgorithmic inputs. Since they can replicate any algorithm execution provided as input (the management paradigm), interactive identity machines in principle have richer behavior than Turing machines, though they are unlikely to exhibit useful behavior unless strongly prompted, just as monkeys at a typewriter can in principle produce great literature but are unlikely to do so.

Programs like Eliza harness intelligence in the environment by echoing a slightly transformed version of received messages back to the sender. The echo principle is used also in a chess machine that wins half its games by echoing the moves of one player on the board of another without understanding the principles of chess. Interactive intelligence may be viewed as the ability to manage and coordinate activities: the productivity of managers is measured by their effectiveness in coordinating the activity of subordinates rather than by their technical cleverness.

Von Neumann computers and Turing machines bias our models towards active inner intelligence within computers rather than passive harnessing of external intelligence. Biological models differ from formal models by being situated, embodied, reactive, and emergent. Situated models compute by responding to their environment rather than to inner computed results, according to the principle that "the world (in which they are situated) is its own best model". Embedded systems interact in real time with the actual physical world; their physical (empirical) grounding prevents infinite regress of primitive building blocks and forces the designer to deal with all the issues. Reactive systems are designed to interact with the envi-

ronment over time rather than to compute functional values.

Emergent systems have behavior emerging from inner evolution that cannot be explicitly specified; because it cannot be ascribed to any subpart (homunculus), emergent behavior captures the notion that the whole is greater than the sum of its parts. Reactive architectures that are situated and embodied can exhibit certain kinds of intelligent behavior without reasoning or symbol representation by totally decentralized computation. The distinction between centralized and decentralized systems has been extensively studied in economics and the social sciences. The hypothesis that free decentralized societies have an evolutionary advantage over totalitarian centralized societies suggests that distributed computing systems will ultimately prove more robust than centralized single-processor systems and that centralized formalization may be too brittle a form of intelligence to survive evolutionary competition.

6.6 Nonmonotonic logic and the closed-world assumption

Traditional reasoning is monotonic because the set of true assertions monotonically increases as we prove more theorems. Reasoning about knowledge bases that acquire knowledge interactively may be nonmonotonic in that assumptions considered provisionally true may be invalidated by later data. The closed world assumption (CWA) reestablishes monotonicity in interactive systems by assuming that any unprovable statement will remain unprovable and may therefore be treated as false. Systems satisfying the CWA are effectively closed, since they cannot receive new information to make false statements true, while open systems do not satisfy the CWA because the interactive acquisition of knowledge can make false statements true. Nonmonotonic logics are more flexible than monotonic logic in reasoning about empirical worlds in which increasing knowledge changes the truth value of assertions. But nonmonotonic logics are unwieldy as practical tools of reasoning. The connection between nonmonotonicity and interaction made explicit by the CWA suggests that nonmonotonicity is an inherent property of interaction machines that could be used as an alternative basis for proving that interaction machines cannot be expressed by first-order logic. Alternatively, nonexpressibility of interaction machines by first-order logics could be used to prove that nonmonotonic logics cannot be so expressed, but this is already obvious.

The frame problem is the problem of inferring that a state of affairs will not change unless it is explicitly changed by an event. Formal guarantee of this property generally requires an enormous number of frame axioms to take into account all possible contingencies. The frame problem is hard because the task of statically specifying all possible dynamic contingencies before they occur is intractable. It disappears if we allow contingencies to be handled dynamically at the time they occur in interactive systems, but interactive knowledge acquisition violates the CWA and monotonicity. For example, objects whose state changes only by executing operations in their interface do not require formal frame axioms because the property that frame axioms are designed to ensure is implicitly realized by a mechanism with an implicit nonformalized notion of encapsulation. The CWA facilitates monotonic reasoning but is incompatible with open interactive knowledge acquisition. Conversely, open, interactive systems are nonmonotonic and do not satisfy the CWA. Models of computation must choose between closed worlds with monotonic reasoning and open empirical worlds in which reasoning is not monotonic.

6.7 Machines that can think

The Turing test [Tu2] provides a criterion for testing if a machine can think by a series of questions to a disembodied question-answering agent (oracle). Turing implicitly assumes that oracles have the expressiveness of Turing machines. His arguments must be reexamined because his oracles may be interaction machines with greater expressiveness than Turing machines. Turing assumes that oracles always receive questions sequentially and have enough time to answer them: he considers oracles that delay their answer to mimic the response time of humans but not the possibility of problems for which Turing machines might be slower than humans. Humans can in fact answer certain questions faster than Turing machines because they can interact as well as compute in answering questions. They can solve some NP-hard questions in polynomial (usually linear) time by prompts concerning the next step from an oracle or human expert.

Interaction machines can in general make use of external as well as inner resources to solve certain

problems more quickly than any disembodied machine. They are more expressive not only in solving certain nonalgorithmic problems, but also solve certain algorithmic problems more efficiently by using interactive nonalgorithmic techniques. Since people are embodied machines, they can solve certain problems like playing chess, scene analysis, or even planning a business trip with partial help from an expert, and thereby perform more efficiently as well as more expressively than Turing machines. Outside help can in general be obtained without knowledge of the interrogator. The special case when input streams are under control of the interrogator allows interaction to be constrained to closed "exam" conditions equivalent to Turing machine computation. But liberal interrogators who allow open book exams can make even controlled interaction more powerful than Turing machines. Embodied machines that solve problems by a combination of algorithmic and interactive techniques are more human in their approach to problem solving than Turing machines, and it is plausible to view such mechanisms as machines that can think.

The relation between the questioner and Turing test agent corresponds to that between harnesses and harnessed systems in figure 26: agents with the power of Turing machines correspond to closed harnesses while agents with the power of interaction machines correspond to open harnesses (figure 43):

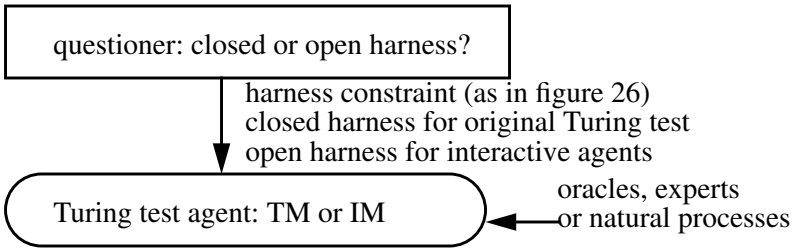


Figure 43: Turing Test as a Harnessed Interaction Machine

Turing's rationalist belief that computers would in time be able to answer questions indistinguishably from humans (pass the Turing test) was challenged by Searle, who argued that passing the test did not constitute thinking because competence did not imply inner (intentional) understanding, illustrating his objections with the "Chinese room" Gedankenexperiment. Penrose [Pe] argued that machines will never be able to pass the test because laws of physics are more expressive than computable functions and are therefore more expressive than any computer. However, Penrose's arguments, though valid for Turing machines, is invalid for the intuitive notion of computing of interaction machines: distributed computers can model precisely the kind of action at a distance that are the primary example used by Penrose as evidence for the noncomputability of physics. Interaction machines have the expressive power of empirical models in physics and other disciplines. They are a plausible model of both thinking and physical phenomena. Though Lagrange's model of the universe as a closed synchronous clock must be discarded, the model of the universe as a collection of asynchronous interacting computing agents remains a plausible model of both physical and mental phenomena.

DNA computers provide a chemical model of computation closely related to process models of interaction. By interpreting DNA sequences as codes for addresses of potential communication ports, a close connection is established between biological computers and models of interaction based on broadcasting (unicasting) such as the pi calculus. Such models support mobile processing and massive parallelism and make algorithmically intractable problems like scene recognition potentially tractable. The chemical mechanism of digital communication that underlies DNA computers could well be a key architecture for modeling the thinking of the future. Thought is interactive as well as algorithmic and machines that include algorithmic computation along with a version of the chemical model of interaction of DNA computers may be able to capture both intentional and extensional aspects of the process of thinking.

7. References

[Ab] Samson Abramski, Computational interpretation of linear logic, Theoretical Computer Science 111 p3-57, 1993.

[Ad] Leonard Adleman, On Constructing Molecular Computers, USC report, January 1995.

[AH] Rajeev Alur and Thomas Henzinger, Real-Time Logics: Complexity and Expressiveness, LICS 1990.

[AP] Martin Abadi and Gordon Plotkin, A Logical View of Composition, Theoretical Computer Science, June 1993.

[BBHK] G. Boudol, I. Castellani, M. Henessy, and A. Kiehn, Observing Localities, Theoretical Computer Science, June 1993.

[BHG] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman, Concurrency Control and recovery in Database Systems, Addison Wesley 1987.

[BWM] Henk Barendregt, Hanno Wupper, Hans Mulder, Computable Processes, CWI report, 1994.

[Bo] Peter van Emde Boas, Machine Models and Simulation, in Handbook of Theoretical Computer Science, Elsevier 1990.

[Br] Rodney A. Brooks, Intelligence Without Reason, *Proc 12th International Joint Conference on Artificial Intelligence*, August 1991.

[GHJV] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley 1994.

[DK] Thomas Dean and Subbaro Kambhampati, Planning and Scheduling, Handbook of Computer Science and Engineering, CRC Press 1996.

[GJ] Michael Garey and David Johnson, Computers and Intractability, Freeman 1979.

[Gr] Stephen Graubard, The Artificial Intelligence Debate, MIT Press 1988

[HC] G. Hughes and M. Creswell, An Introduction to Modal Logic, Methuen 1968.

[Ho] Anthony Hoare, Communicating Sequential Processes, CACM August 1978.

[HLP] Eyal Harel, Orna Lichtenstein, and Amir Pnueli, Explicit Clock Temporal Logic, LICS 1990.

[Ja] Ivar Jacobson, *Object-Oriented Software Engineering,* Addison-Wesley/ACM-Press, 1991.

[KRB] Gregor Kiczales, Jim des Rivieres, and Daniel Bobrow, The Art of the Metaobject Protocol, MIT Press 1991.

[La1] Butler Lampson, Authentication in Distributed Systems, in Distributed Systems, Edited by Sape Mullender, Addison Wesley 1993.

[La2] Leslie Lamport, Time, Clocks and Ordering of Events ina Distributed System, Comm ACM, July 1978.

[LMWT] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete, *Atomic Transactions,* Morgan Kaufman, 1994.

[Mil] Robin Milner, Elements of Interaction, *CACM*, January, 1993.

[Mil2] Robin Milner, Action Structures in the Pi Calculus, Nato Advanced Study Institute 1994.

[Min] Grigory Mints, A Short Introduction to Modal Logic, CSLI Lecture Notes, #30, 1992.

[MP] Zohar Manna and Amir Pnueli, The Temporal Logic of Reactive and Concurrent Systems

[Pe] Roger Penrose, *The Emperor's New Mind*, Oxford, 1989.

[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.

[Si] Herbert Simon, *The Sciences of the Artificial,* MIT Press, Second Edition, 1982.

[SH] Vijay Saraswat and Pascal van Hentenryk, Principles and Practice of Constraint Programming, MIT Press 1995

[Tr] Anne Troelstra, Lectures in Linear Logic, CSLI Lecture Notes #29, 1992.

Tu1] Alan Turing, Systems of Logic based on Ordinals, Proc London Math Soc 1939.

[V] Moshe Vardi, On the Complexity of Modular Model Checking, LICS 1995.

[Tu2] Alan Turing, Computing Machinery and Intelligence, *Mind,* 1950.

[VW] Moshe Vardi and Pierre Wolper, Reasoning about infinite sequences, Information and Computation, November 1994.

[We1] Peter Wegner, Interactive Foundations of Obejct-Based Programming, IEEE Computer, October 1995

[We2] Peter Wegner, Interaction as a Basis for Empirical Computer Science, Computing Surveys, March 1995.

[We3] Peter Wegner, Tradeoffs between Reasoning and Modeling, in Research Directions in Concurrent Object-Oriented Programming, eds Agha, Yonezawa, Wegner, MIT Press 1993.

[WLM] Peter Wegner, Evelina Lamma, and Paola Mello, The Role of State in Open Systems, Brown Technical report, Sept 1994.

[WWC] Gio Wiederhold, Peter Wegner, and Stefano Ceri, Towards Megaprogramming, CACM, Nov 1992.