The Paradigm Shift from Algorithms to Interaction

Peter Wegner, Brown University, Final Draft, October 14 1996

The paradigm shift from algorithms to interaction captures the technology shift from mainframes to workstations and networks, from number-crunching to embedded systems and graphical user interfaces, and from procedure-oriented to object-based and distributed programming. Interaction is shown to be more powerful than rule-based algorithms for computer problem solving, overturning the prevalent view that all computing is expressible as algorithms. The radical notion that interactive systems are more powerful problem-solving engines than algorithms is the basis for a new paradigm for computing technology built around the unifying concept of interaction.

From Sales to Marriage Contracts

The evolution of computer technology from the 1970s to the 1990s is captured by a paradigm shift from algorithms to interaction. Algorithms yield outputs completely determined by their inputs that are memoryless and history independent, while interactive systems like personal computers, airline reservation systems, and robots provide history-dependent services over time that can learn from and adapt to experience.

Objects are interactive agents that provide services to their clients not expressible by algorithms. Algorithms are “sales contracts” that deliver an output in exchange for an input, while objects are ongoing “marriage contracts.” An object’s contract with its clients specifies its behavior for all contingencies of interaction (in sickness and in health) over the lifetime of the object (till death us do part) [8]. The folk wisdom that marriage contracts cannot be reduced to sales contracts is made precise by showing that interaction cannot be expressed by algorithms. Contracts over time include algorithms that instantaneously transform inputs to outputs as a special case.

Interactive tasks like driving home from work cannot be realized by algorithms. Algorithms automatically executable without looking out of the window cannot handle traffic and other interactive events. Algorithms are surprisingly versatile; but problems are harder to solve when interaction is precluded (closed-book exams are harder than open-book exams). Interaction can simplify tasks when algorithms exist and is the only game in town for inherently interactive tasks like driving or airline reservation.

Smart bombs that interactively check observations of the terrain against a stored mental map of their route are “smart” because they enhance algorithms by interaction. This example shows that interaction enhances *dumb algorithms* so they become *smart agents*. Algorithms are metaphorically dumb and blind because they cannot adapt interactively while they compute. They are autistic in performing tasks according to rules rather than by interaction. In contrast, interactive systems are grounded in an external reality both more demanding and richer in behavior than the rule-based world of noninteractive algorithms.

Objects can remember their past and interact with clients through an interface of operations that share a hidden state (see Figure 1). An object’s operations are not algorithms because their response to messages depends on a shared state accessed through nonlocal variables of operations. The effect of a check-balance operation of a bank account is not uniquely determined by the operation alone, since it depends on changes of state by deposit and withdraw operations that cannot be predicted or controlled. An object’s operations return results that depend on changes of state controlled by unpredictable external actions.

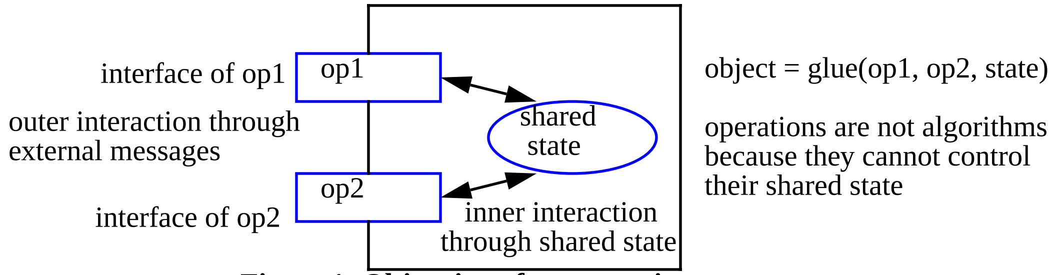


Figure 1: Object interface operations

The growing pains of software technology are due to the fact that programming in the large is inherently interactive and cannot be expressed by or reduced to programming in the small. The behavior of airline reservation systems and other embedded systems cannot be expressed by algorithms. Fred Brooks' persuasive argument [1] that there is no silver bullet for specifying complex systems is a consequence of the irreducibility of interactive systems to algorithms. If silver bullets are interpreted as formal (or algorithmic) system specifications, the nonexistence of silver bullets can actually be proved.

Artificial intelligence has undergone a paradigm shift from logic-based to interactive (agent-oriented) models that parallels that in software engineering. Interactive models provide a common framework for agent-oriented artificial intelligence, software engineering, and system architecture [10].

Though object-based programming has become a dominant practical technology, its conceptual framework and theoretical foundations are still unsatisfactory: it is fashionable to say that everyone talks about it but no one knows what it is. "Knowing what it is" has proved elusive because of the implicit assumption that explanations must specify "what it is" in terms of algorithms. Accepting irreducibility as a fact of life has a liberating effect: "what it is" can be more naturally defined in terms of interactive models.

From Turing to Interaction Machines

Turing showed in the 1930s that algorithms in any programming language have the same transformation power as Turing machines, computing precisely the *computable functions*. This precise characterization of what can be computed established the respectability of computer science as a discipline. However, the inability to compute more than the computable functions by adding new primitives proved so frustrating that it was called the Turing tarpit. Interactive computing lets us escape from the Turing tarpit.

Turing machines transform strings of input symbols on a tape into output strings by sequences of state transitions (see Figure 2). Each step reads a symbol from the tape, performs a state transition, writes a symbol on the tape, and moves the reading head. Turing machines cannot, however, accept external input while they compute: they shut out the external world and cannot therefore model the passage of external time.

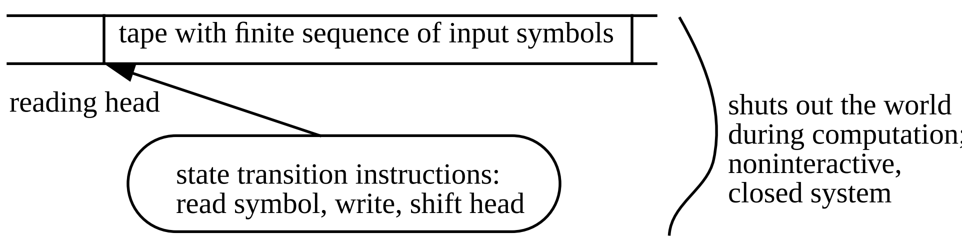


Figure 2: Turing machine as a closed system

The hypothesis that the *formal* notion of computability by Turing machines corresponds to the *intuitive* notion of what is computable, known as Church's thesis, is not a theorem but has been accepted as obviously true for 50 years. However, when the intuitive notion of what is computable is broadened to include interactive computations, Church's thesis is invalid. Though Church's thesis is valid in the narrow sense that Turing machines express the behavior of algorithms, the broader assertion that algorithms capture the intuitive notion of what computers compute is invalid when computing is extended to include interaction.

Turing machines extended by adding *input* and *output* actions that support dynamic interaction with an external environment are called *interaction machines*. Though interaction machines are a simple and obvious extension of Turing machines, this small change increases expressiveness so it becomes too rich to have nice mathematical models. Interaction machines may have single or multiple input streams, synchronous or asynchronous actions, and can differ along many other dimensions. Distinctions among interaction machines are examined in [9], but all forms of interaction transform closed to open systems and express behavior beyond that computable by algorithms:

Claim: Interaction-machine behavior is not reducible to Turing-machine behavior

Informal evidence of richer behavior: Turing machines cannot handle the passage of time or interactive events that occur during the process of computation.

Formal evidence of irreducibility: input streams of interaction machines are not expressible by finite tapes, since any finite representation can be dynamically extended by uncontrollable adversaries.

The radical view that Turing machines are not the most powerful computing mechanisms has a distinguished pedigree. It was accepted by Turing, who showed in 1939 that Turing machines with oracles (like the oracle at Delphi that could predict the future) were more powerful than Turing machines. Milner [3] noticed as early as 1975 that concurrent processes cannot be expressed by sequential algorithms, while Manna and Pnueli [2] showed that nonterminating reactive processes like operating systems cannot be modeled by algorithms. Godel's discovery that the integers cannot be completely described by logic, which demonstrates the limitations of formalism in mathematics, may be extended to show that interaction-machines also cannot be completely described by first-order logic (see Figure 10).

Input and output actions are "logical" sensors and effectors that affect external data even when they have no physical effect. Objects and robots have very similar interactive models of computation: robots differ from objects only in that their sensors and effectors have physical rather than logical effects. Interaction machines can model objects, software engineering applications, robots, intelligent agents, distributed systems, and networks like the internet and the World Wide Web (see Figure 3).

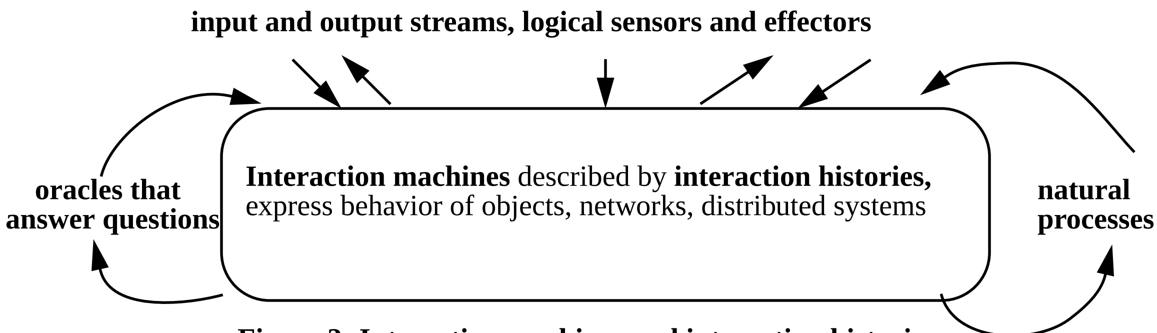


Figure 3: Interaction machines and interaction histories

The observable behavior of interaction machines is specified by *interaction histories* that, in the case of simple objects, like bank accounts with deposit and withdraw operations, are described by streams of operations called traces. Operations whose effects depend on their time of occurrence, as in interest-bearing bank accounts, require time-stamped traces. Objects with inherently nonsequential interfaces, like joint bank accounts accessed from multiple automatic-teller machines, have inherently nonsequential interaction histories. Interaction histories of distributed systems, just like history in history books, consists of nonsequential events that may have duration and interfere with each other.

Whereas interaction histories express the external unfolding of events in time, instruction-execution "histories" express an ordering of inner events of an algorithm without any relation to the actual passage of time. Algorithmic time is intentionally measured by number of instructions executed rather than by the actual time of execution in order to provide a hardware-independent measure of logical complexity. In contrast, the duration and the time that elapses between the execution of operations may be interactively significant. Operation sequences are interactive streams with temporal as well as functional properties, while instruction sequences have a state-transition semantics.

Pure Interaction, Judo, and the Management Paradigm

Raw interactive power is captured by interactive-identity machines (IIMs) that immediately output their input without transforming it. IIMs are simple transducers that realize nonalgorithmic behavior by harnessing the computing power of the environment:

loop input(message); output(message); **end loop**
or
while true **do** echo input **end while**
or

P = in(message).out(message).P

IIMs employ the judo principle of using the weight of adversaries (or cooperating agents) to achieve a desired effect. They realize the “management paradigm,” coordinating and delegating tasks without necessarily understanding their technical details. Though IIMs are not intelligent, they can behave intelligently by replicating intelligent inputs from the environment.

claim: interactive identity machines have richer behavior than Turing machines

evidence: an IIM can mimic any Turing machine and any input stream from the environment

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. The chess machine M in Figure 4 makes use of intelligent input actions through one interface to deliver intelligent outputs through another interface: though it is unintelligent by itself, it harnesses the intelligence of one player to respond intelligently to a second player. Clients of M like player A are unaware of the interactive help that M receives through its interface to B. 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. From A’s viewpoint, B’s actions through a hidden interface are indistinguishable from those of a daemon hidden inside the machine.

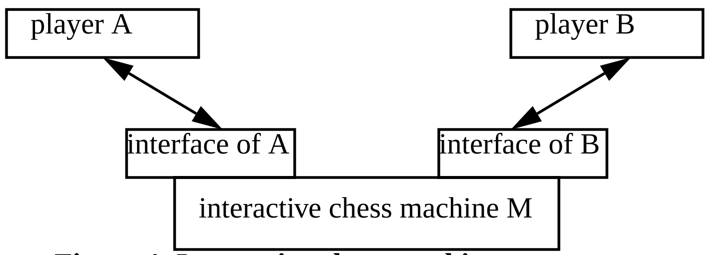


Figure 4: Interactive chess machine

Simon’s well known example [5] of the irregular behavior of an ant on a beach in finding its way home to an ant colony illustrates that complex environments cause simple interactive agents to exhibit complex behavior (see Figure 5). The computing mechanism of the ant is presumed to be simple, but the behavior of the ant reflects the complexity of the beach, whose nonalgorithmic topography causes the ant to traverse a nonalgorithmic path. The behavior of ants on beaches cannot be described by algorithms because the set of all possible beaches cannot be so described.

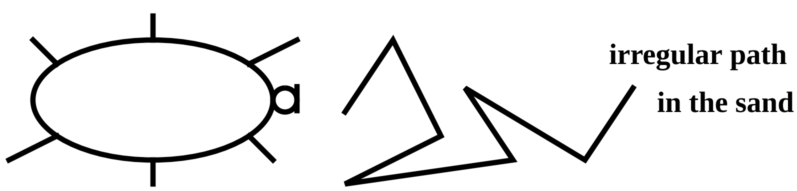


Figure 5: Path of ant reflects complexity of the beach

Though interaction opens up limitless possibilities for harnessing the environment, it is entirely dependent on external resources, while machines with built-in algorithmic cleverness are not. High achievement, whether by machines or people, can be realized either by self-sufficient inner cleverness or by harnessing the environment. The achievements of presidents of large corporations or of the President of the United States are dependent on the effective use of a supporting environmental infrastructure. Interaction scales up to very large problems better than inner cleverness, because it expresses delegation and coordination.

Interaction, Parallelism, Distribution, and Openness

Interaction, parallelism, and distribution are conceptually distinct concepts:

interactive systems interact with an external environment that they do not control

parallelism (concurrency) occurs when computations of a system overlap in time
distribution occurs when components of a system are geographically or logically separated

Parallel and distributed computation can, in the absence of interaction, be expressed by algorithms. Parallel algorithms are the subject of many textbooks and university courses. Distribution likewise increases merely the complexity of systems without necessarily violating algorithmicity.

The recognition that interaction rather than parallelism or distribution is the key element in providing greater behavioral richness is a nontrivial insight that requires reappraisal of the role of parallelism and distribution in complex systems. The horizontal base plane of Figure 6 includes many interesting and important algorithmic systems, while systems not in the base plane are nonalgorithmic.

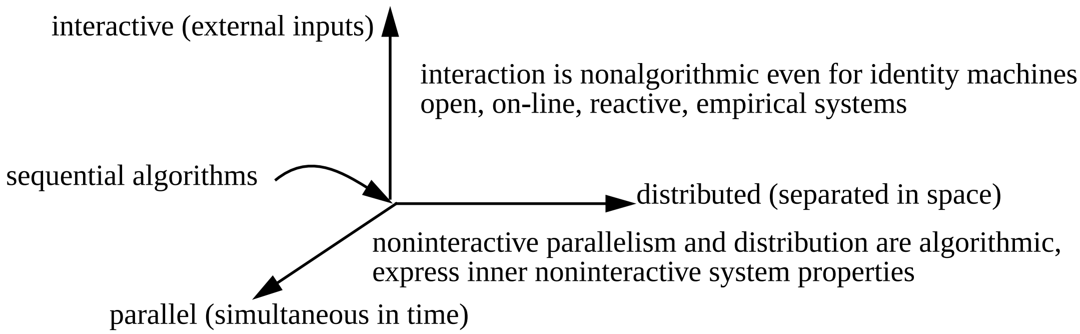


Figure 6: Design space for interactive, parallel, and distributed computing

Agents in the environment can interact in a cooperative, neutral, or malicious way with an interaction machine. Adversaries that interfere with algorithmic goals provide a measure of the limits of interaction machine behavior. Synchronous adversaries control "what" inputs an agent receives, while asynchronous adversaries have additional power over "when" an input is received. Asynchronous adversaries who can decide when to zap you are interactively more powerful than synchronous adversaries.

Interactiveness provides a natural and precise definition of the notion of open systems. Open systems are defined as systems that can be modeled by interaction machines, while closed systems are modeled by algorithms. Interaction machines provide a precise definition not only for open systems but also for other fuzzy concepts like empirical computer science and programming in the large. They robustly capture many alternative notions of interactive computing just as Turing machines capture algorithmic computing [9].

Open systems have very rich behavior to handle all possible clients, while the individual interface demands of clients are often quite simple. Open systems whose unconstrained behavior is nonalgorithmic can become algorithmic by strongly constraining their interactive behavior [9]:

supplied behavior of interactive system >> demanded behavior at a given interface

Interfaces are a primitive building block of interactive systems, playing the role of primitive instructions. Interactive programmers compose (plug together) interfaces just as algorithmic programmers compose instructions. Interactive software technology for interoperability, patterns, and coordination can be expressed in terms of relations among interfaces [10]. Interfaces express the mode of use or pragmatics of an interactive system, complementing syntax and semantics (see Figure 9).

Closed systems with algorithmic behavior have open subsystems with nonalgorithmic behavior. For example, an engine of a car may behave unpredictably when a spark plug is removed. Animals (or persons) with an established behavior routine may behave erratically in unfamiliar environments. Subsystems with predictable (algorithmic) constrained behavior have unpredictable (nonalgorithmic) behavior when the constraint is removed and a greater range of possible behaviors must be considered.

Interactiveness (openness) is *nonmonotonic* in that decomposition of systems may create interactive unpredictable subsystems, or equivalently composition of interactive systems may produce noninteractive algorithms. In contrast, concurrency and distribution are monotonic: if a system is not concurrent or not distributed all subsystems also have this property. Nonmonotonicity implies that noninteractive systems with algorithmic behavior may have interactive subsystems whose behavior is nonalgorithmic.

Interfaces as Behavior Specifications

The negative result that interaction is not expressible by algorithms leads to positive new approaches to system modeling in terms of interfaces. Giving up the goal of complete behavior specification requires a psychological adjustment, but makes partial system specification by interfaces respectable. Since the complete elephant cannot be specified, the focus shifts to specifying its parts and its forms of behavior (its trunk or mode of eating peanuts). Complete specification must be replaced by partial specification of interfaces, views, and modes of use. Airline reservation systems can be specified by multiple instances of the following kinds of interfaces:

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

Airline reservation systems have a large number of geographically distributed interfaces, each of which has a normal mode of use that may break down under abnormal overload conditions. Requirements of an airline reservation system may be specified by the set of all interfaces (modes of use) it should support. The description of systems by their modes of use is a starting point for system design: interfaces play a practical as well as conceptual role in interactive system technology.

A system satisfies its requirements if it supports specified modes of use even though correct behavior for a given mode of use is not guaranteed and complete system behavior for all possible modes of use is unspecifiable. Though correctness of programs under carefully qualified conditions can still be proved, result checking is needed during execution to verify that results actually obtained are valid. Techniques for systematic on-line result checking will play an increasingly important practical and formal role as a supplement to off-line testing and verification. Result checking for tasks like driving with visual feedback is performed automatically by people, but must be performed by instruments or programs as safeguards against airplane crashes and other costly embedded-system failures.

Interface descriptions are called *harnesses* since they serve both to constrain system behavior (like the harness of a horse) and to harness behavior for useful purposes. Harnesses have both a negative connotation as constraints and a positive connotation as specifications of useful behavior, while interfaces focus primarily on the positive connotation. We distinguish between *open harnesses* that permit interaction through other harnesses or exogenous events and *closed harnesses* that cause the system together with its harness to become closed and thereby algorithmic (see Figure 7).

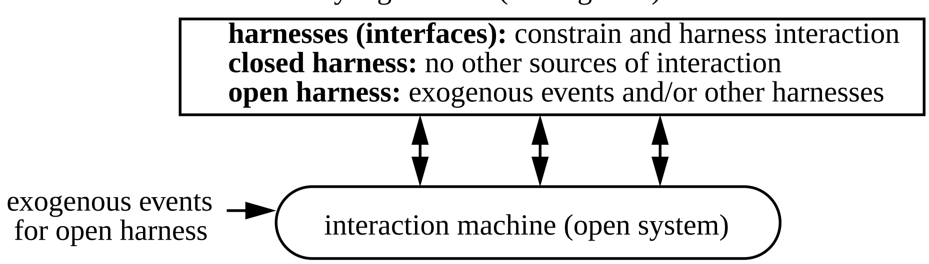


Figure 7: Harnesses that constrain interactive behavior

Airline reservation systems are naturally described by a collection of open harnesses. Microsoft's Component Object Model (COM) describes components by the interfaces they support. The key property possessed by every COM component is an interface directory through which the complete set of interfaces may be accessed: every component has an interface I-unknown with *queryinterface, addinterface*, and *deleteinterface* operations. Industrial-strength object-based models suggest that open harnesses are a more flexible framework than inheritance for modeling complex objects. Interaction machines conform more closely to industrial models like COM than to inheritance models.

The goal of proving correctness for algorithms is replaced for interactive systems by the more modest goal of showing that components have collections of interfaces (harnesses) corresponding to desired forms of useful behavior. Interaction machines that specify software systems are described by multiple interfaces that express functionality for different purposes. People are likewise better described by collections of interfaces: the behavioral effect of “thinking” is better described by interaction machines with multiple interfaces than by Turing machines (see Figure 11).

From Rationalism to Empiricism

Plato’s parable of the cave, which compares humans to cave dwellers who can observe only shadows of reality on the walls of their cave but not actual objects in the outside world, shows that observation cannot completely specify the inner structure or behavior of observed objects (see Figure 8). Projections of light on our retina serve as incomplete cues for constructing our world of solid tables and chairs.

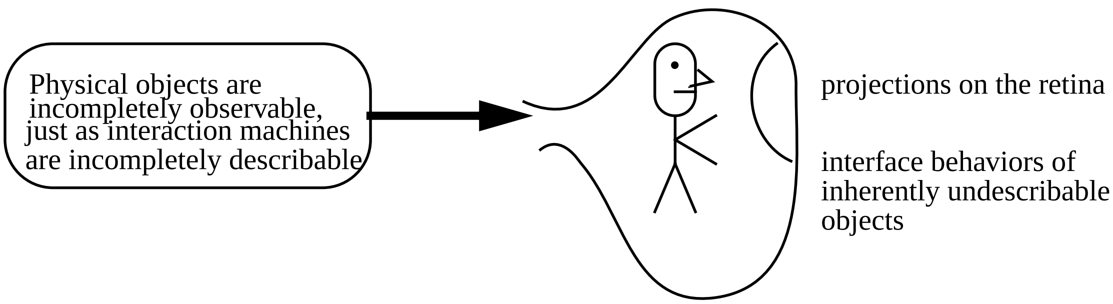


Figure 8: Plato’s cave as a metaphor for empirical computer science

Plato concluded that abstract ideas are more perfect and therefore more real than physical objects like tables and chairs. His skepticism concerning empirical science contributed to the 2000-year hiatus in the evolution of empiricism. Empiricists accept the view that perceptions are reflections of reality but disagree with Plato on the nature of reality, believing that the physical world outside the cave is “real” but unknowable. Fortunately, “complete” knowledge is unnecessary for empirical models of physics, because they achieve their pragmatic goals of prediction and control by dealing entirely with observable reflections.

Modern empirical science rejects Plato’s belief that incomplete knowledge is worthless, using partial descriptions (shadows) to control, predict, and understand the objects that shadows represent. Differential equations capture quantitative properties of phenomena that they model without requiring a complete description. Similarly, computing systems can be specified by interfaces that describe properties judged to be relevant while ignoring properties judged irrelevant. Plato’s cave, properly interpreted, is a metaphor for empirical abstraction in both natural science and computer science.

Turing machines correspond to Platonic ideals in focusing on mathematical at the expense of empirical models. To realize logical completeness, they sacrifice the ability to model external interaction and real time. The extension from Turing to interaction machines, and from procedure-oriented to object-based systems, is the computational analog of liberation from the Platonic world view that led to the development of empirical science. Interaction machines liberate computing from the “Turing tarpit” of algorithmic computation, providing a conceptual model for software engineering, AI agents, and the real (physical) world.

The contrast between algorithmic and interactive models parallels interdisciplinary distinctions in other domains of modeling. It arises in its purest form in philosophy, where the argument between rationalists and empiricists has been a central and hotly debated issue for over 2000 years. Descartes’ quest for certainty led to the rationalist credo “cogito ergo sum,” which succinctly asserts that thinking is the basis of existence and implies that “certain” knowledge of the physical world is possible only through inner processes of algorithmic thinking. Hume was called an empiricist because he showed that inductive inference and causality are not deductive and that rationalist models of the world are inadequate. Kant, "roused from his dogmatic slumbers" by Hume, wrote the *Critique of Pure Reason* to show that "pure" reasoning about necessarily true knowledge was inadequate to express contingently true knowledge of the real world.

Rationalism continued to have great appeal in spite of the triumph of empiricism in modern science. Hegel, whose "dialectical logic" extended reason beyond its legitimate domain, influenced both political philosophers like Marx and mathematical thinkers like Russell. George Boole's treatise *The Laws of Thought* demonstrated the influence of rationalism in equating logic with thought. Mathematical thought in the early 20th century was dominated by Russell's and Hilbert's rationalist reductions of mathematics to logic. Godel's incompleteness result showed the flaws of rationalist mathematics, but logicians and other mathematicians have not fully appreciated the limitations on formalism implied by Godel's work.

The term "fundamental," as in "fundamental particles" or "foundations of mathematics," is a codeword for rationalist models. The presence of this same codeword in terms like "religious fundamentalism" suggests that social and scientific rationalism have common roots in the deep-seated human desire for certainty. Rationalism is an alluring philosophy with many guises (disguises).

Though empiricism has displaced rationalism in the sciences, Turing machines reflect rationalist reasoning paradigms of logic rather than empirical paradigms of physics. Algorithms and Turing machines, like Cartesian thinkers, shut out the world during the process of problem solving. Turing was born in 1912 and matured just around the time that Godel delivered his coup de grace to formalist mathematics. But the effects of Godel's incompleteness result were slow to manifest themselves among logicians like Church, Curry, and Turing who shaped the foundations of computer science.

Abstraction is a key tool in simplifying systems by focusing on subsets of relevant attributes and ignoring irrelevant ones. Incompleteness is the key distinguishing mechanism between rationalist, algorithmic abstraction and empiricist, interactive abstraction. The comfortable completeness and predictability of algorithms is inherently inadequate in modeling interactive computing tasks and physical systems. The sacrifice of completeness is frightening to theorists who work with formal models like Turing machines, just as it was for Plato and Descartes. But incomplete behavior is comfortably familiar to physicists and empirical model builders. Incompleteness is the essential ingredient that distinguishes both interactive from algorithmic models of computing and empirical from rationalist models of the physical world.

Models in Logic and Computation

Models in both logic and computation aim to capture *semantic* properties of a domain of discourse or modeled world by *syntactic* representations for the *pragmatic* benefit of users. They express properties of physical or mathematical *modeled worlds* in a form that is pragmatically useful.

A model M = (R, W, I) is a representation R of a modeled world W interpreted by a human or mechanical interpreter I that determines semantic properties of W in terms of syntactic expressions of R. R, W, and I respectively determine the syntax, semantics, and pragmatics of the model (see Figure 9).

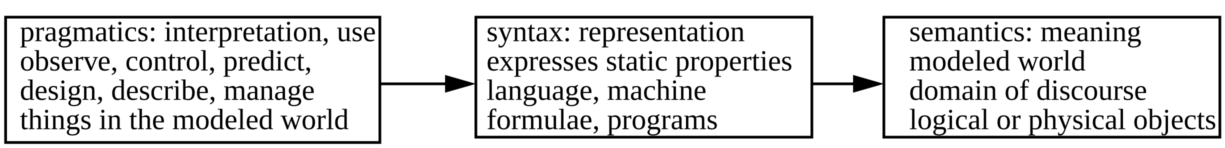


Figure 9: Interdisciplinary model for modeling

Interactive models have multiple pragmatic modes of use while algorithms have a single intended pragmatic interpretation determined by the syntax. The goal of expressing semantics by syntax is replaced by the interactive goal of expressing semantics by multiple pragmatic modes of use. The goal of *complete* behavior specification is replaced by that of harnessing useful forms of *partial* behavior through interfaces.

Logical proof involves step-by-step progress from a starting point to a result:

logical system -> *programming language*

well-formed formulae -> *programs*

theorem to be proved -> initial input
rules of inference -> nondeterministic rules of computation
proofs -> sequential algorithmic computations

Reasoning is less ambitious and therefore weaker than interactive computing or physics for modeling and problem solving [7]. Hobbes was correct in saying that “reasoning is but reckoning,” but the converse assertion “reckoning is but reasoning” is false, since interactive reckoning is richer than reasoning.

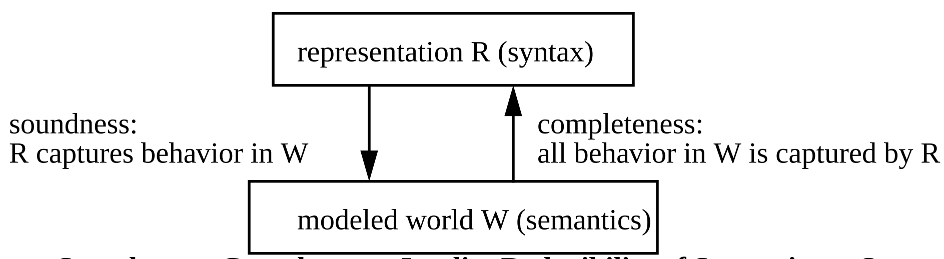
The inherent incompleteness of interactive systems has the practical consequence that maximal goals of logic and functional programming and of formal methods cannot be achieved. The result that logic programming is too weak to model interactive systems, presented by the author at the closing session of the fifth-generation computing project in Tokyo in 1992 [7], showed that the project could not have achieved its maximal software-engineering goals even with a tenfold increase in effort or a ten-year extension.

Algorithms and logical formulae take their meanings in the same semantic world of computable functions, but logic is a purer paradigm that expresses the relation between syntax and semantics with fewer distractions. Well-formed formulae are semantically interpreted as assertions about a *modeled world* that may be true or false. Formulae true in all interpretations are called *tautologies*.

A logic is *sound* if all syntactically provable formulae are tautologies and *complete* if all tautologies are provable. Soundness and completeness measure the adequacy of syntactic proofs in expressing semantic meaning. They relate the syntactic representation R of a logical model to its semantic modeled world W.

soundness implies that syntactically proved theorems express meaning in the modeled world
completeness implies that all meanings can be syntactically captured as theorems

Soundness ensures that representations correctly model behavior of their modeled worlds, while completeness ensures that all possible behavior is modeled. Soundness and completeness together ensure that a representation is correct and that it captures all behavior in the world being modeled. But completeness restricts the semantics of modeled worlds to those completely expressible by a representation. It constrains modeling power to syntactically expressible behavior (see Figure 10).



Soundness + Completeness Implies Reducibility of Semantics to Syntax

Figure 10: Completeness constrains expressiveness

Soundness ensures that proofs are semantically correct, while completeness measures the comprehensiveness of the proof system in expressing semantic meaning. Soundness and completeness together imply that W is reducible to R (W and R are isomorphic abstractions). Reducibility of W to R implies completeness of R in expressing properties of W, while incompleteness implies irreducibility.

Though soundness and completeness are desirable formal properties, they are often abandoned for practical reasons. For example, logics for finding errors in programs are sound if they generate error messages only when the program has an error and complete if they discover all errors:

soundness: if error message then error
completeness: if error then error message

In this case insisting on soundness conservatively excludes useful error messages because they are occasionally wrong, while complete logics recklessly generate many spurious error messages in their quest for completeness. Practical logics are neither sound nor complete, generating some erroneous messages and missing some errors to strike a balance between caution and aggressiveness.

The goal of error analysis is to check that a syntactically defined error-detection system captures an independent semantic notion of error. Since the semantic notion of error in dynamically executed programs cannot be statically defined it cannot be completely formalized, but the semantic notion of error can be syntactically approximated. In choosing an approximation the extreme conservatism of soundness and the extreme permissiveness of completeness are avoided by compromising (in both the good and bad senses of the word) between conservatism and permissiveness.

Godel's incompleteness result for a particular mathematical domain (arithmetic over the integers) has an analog in computing. The key property of incomplete domains is irreducibility. Completeness is possible only for a restricted class of relatively trivial logics over semantic domains reducible to syntax. It restricts behavior to that describable by algorithmic proof rules. Models of the real world and even of integers sacrifice completeness in order to express autonomous (external) meanings. Incompleteness is a necessary price to pay for modeling independent domains of discourse whose semantic properties are richer than the syntactic notation by which they are modeled:

open, empirical, falsifiable, or interactive systems are necessarily incomplete

Mathematically, the set of true statements of a sound and complete logic can be enumerated as a set of theorems and is therefore recursively enumerable. Godel showed incompleteness of the integers by showing that the set of true statements about integers was not recursively enumerable using a diagonalization argument. Incompleteness of interaction machines can be proved by showing that the set of computations of an interaction machine cannot be enumerated. This follows from the fact that dynamically generated input streams are mathematically modeled by infinite sequences. The set of inputs of an interactive input stream has the cardinality of the set of infinite sequences over a finite alphabet, which is not enumerable.

The proof of incompleteness of an interaction machine is actually simpler than Godel's incompleteness proof, since interaction machines are more strongly incomplete than integers. It follows from nonenumerability of infinite sequences over a finite alphabet.and does not require diagonalization.

Conventional wisdom assumed that proving correctness was in principle possible and simply needed greater effort and better theorem provers. However, incompleteness implies that proving correctness is not merely hard but impossible. The goals of research on formal methods must be modified to acknowledge this impossibility. Proofs of existence of correct behavior (type-1 correctness) are in many cases possible, while proofs of the nonexistence of incorrect behavior (type-2 correctness) are generally impossible.

The Interactive Turing Test

Turing [6] proposed a behavioral test, now called the Turing test, that answers the question "can machines think?" affirmatively if the machine responds indistinguishably from humans to a broad range of questions. He not unexpectedly equated "machines" with Turing machines, assuming that machines always answer questions sequentially. Turing permits machines can delay their answer in games like chess to mimic the slower response time of humans, but does not consider that machines may sometimes be inherently slower than humans, or interact through hidden interfaces while they answer questions.

Agents that can receive help from oracles, experts, and natural processes as in Figure 11 have greater question-answering ability than Turing machines. Though IIMs have less "thinking power" than clever algorithms, their range of potential behaviors dominates that of Turing machines.

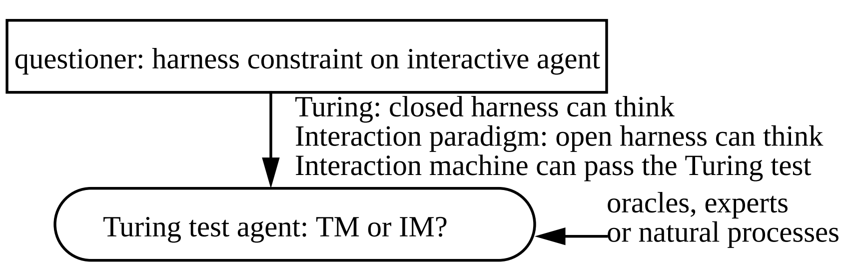


Figure 11: Interactive Turing-test agents

Interaction machines can make use of external as well as inner resources to solve problems more quickly than disembodied machines. They are more expressive in solving inherently nonalgorithmic problems, but also solve certain algorithmic problems more efficiently by using interactive techniques. They can play chess, do scene analysis, or even plan a business trip with partial help from an expert more efficiently as well as more expressively than Turing machines. Outside help can in general be obtained without knowledge of the client (questioner).

The Turing test constrains interaction to closed-book exam conditions, while interaction machines that support open-book exams can expect superior performance. Open-book exams that allow access through a computer to the Library of Congress and the World Wide Web amplify exam-taking power through interactive access to an open, evolving body of knowledge. Note that open-book exams, which simply allow students to add a fixed set of textbooks, become closed exams for an augmented but closed body of knowledge, while exams with access to e-mail and the World Wide Web are truly open and interactive, and allow a larger, nonalgorithmic set of questions to be answered.

Interaction 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 equate such interactive problem solving with thinking.

Skeptics who believe that machines cannot think can be divided into two classes:

intensional skeptics who believe that machines that simulate thinking cannot think:

behavior does not capture inner awareness or understanding

extensional skeptics who believe that machines have inherently weaker behavior than humans

machines cannot inherently model physics or consciousness

Searle argued that passing the test did not constitute thinking because competence did not imply inner understanding, while Penrose [4], asserted that Turing machines are not as expressive as physical models. We agree with Penrose that Turing machines cannot model the real world, but disagree that this implies extensional skepticism because interaction machines can model physical and mental behavior.

Penrose builds an elaborate house of cards on the noncomputability of physics by Turing machines. However, this house of cards collapses if we accept that interactive computing can model physics. Penrose's error in equating Turing machines with the intuitive notion of computing is similar to Plato's identification of reflections on the walls of a cave with the intuitive richness of the real world. Penrose is a self-described Platonic rationalist whose arguments, based on the acceptance of Church's thesis, are disguised forms of rationalism, denying first-class status to empirical models of computation. Penrose's argument that physical systems are subject to elusive noncomputable laws yet to be discovered is wrong, since interaction is sufficiently expressive to describe physical phenomena like action at a distance, nondeterminism, and chaos [9], which Penrose cites as examples of physical behavior not expressible by computers.

Penrose's dichotomy between computing on the one hand and physics and cognition on the other is based on a misconception concerning the nature of computing that was shared by Church and Turing and has its historical roots in the rationalism of Plato and Descartes. The insight that the rationalism/empiricism dichotomy corresponds to algorithms and interaction, and that "machines" can model physics and cognition through interaction, allows computing to be classified as empirical along with physics and cognition. By identifying interaction as the key ingredient that distinguishes empiricism from rationalism and showing that interaction machines express empirical computer science, we can show that the arguments of Plato, Penrose, Church, Turing, and other rationalists are rooted in a common fallacy concerning the role of noninteractive algorithmic abstractions in modeling the real world.

Concluding Remarks

The paradigm shift from algorithms to interaction is a consequence of converging changes in system architecture, software engineering, and human-computer interface technology. Interactive models provide a unifying framework for understanding the evolution of computing technology, as well as interdisciplinary connections to physics and philosophy.

Irreducibility of interaction to algorithms enhances the intellectual legitimacy of computer science as a discipline distinct from mathematics and, by clarifying the nature of empirical models of computation, provides a technical rationale for calling computer science a science. Interfaces of computing systems are the computational analog of shadows on the walls of Plato's cave, providing a framework for system description that is more expressive than algorithms and captures the essence of empirical computer science.

Trade-offs between formalizability and expressiveness arise in many disciplines but are especially significant in computer models. Overemphasis on formalizability at the expense of expressiveness in early models of computing is evident in principles like "Go to considered harmful" and the more sweeping "assignment considered harmful" of functional programming. Functional and gotoless programming, though beneficial to formalizability, are harmful to expressiveness. However, they merely make certain kinds of programs more difficult to write without reducing algorithmic problem-solving power. The restriction of models of computation to Turing machines is a more serious harmful consequence of formalization, reducing problem-solving power to that of algorithms so that the behavior of objects, personal computers, and network architectures cannot be fully expressed.

Computer science is a lingua franca for modeling that allows applications in a variety of disciplines to be uniformly expressed in a common form. Interaction machines provide a unifying framework not only for modeling practical applications but also for talking precisely about the conceptual foundations of model building, so that fuzzy philosophical distinctions between rationalism and empiricism can be concretely expressed in computational terms. The crisp characterization of rationalist versus empiricist models by "algorithms versus interaction" expresses philosophical distinctions by concepts of computation, allowing the interdisciplinary intuition that empirical models are more expressive than rationalist models to be precisely stated and proved. The insight that interactive models of empirical computer science have observably richer behavior than algorithms challenges accepted beliefs concerning the algorithmic nature of computing, allowing us to escape from the Turing tarpit and to develop a unifying interactive framework for models of software engineering, artificial intelligence, and computer architecture.

Annotated References

[1] Brooks, F. *The Mythical Man-Month: Essays on Software Engineering*, second edition, Addison-Wesley, 1995. (there is no silver bullet for expressing systems by algorithms)

[2] Manna, Z. and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems*, Springer Verlag, 1992. (reactive systems are not expressible by algorithms or Turing models)

[3] Milner, R. Elements of Interaction, *CACM*, 36-1 (1993), pp 78-89. (models of concurrency)

[4] Penrose, R. *The Emperor's New Mind*, Oxford, 1989; also *Shadows of the Mind*, Oxford, 1994. (claim that physics is noncomputable but deterministic)

[5] Simon, H. *The Sciences of the Artificial*, MIT Press, second edition, 1982. (ant on a beach)

[6] Turing, A. Computing Machinery and Intelligence, *Mind*, 59 (1950), pp33-60. (the Turing test)

[7] Wegner, P. Trade-offs Between Reasoning and Modeling, in *Research Directions in Concurrent Object-Oriented Programming*, Eds Agha G. Wegner P. Yonezawa A, MIT Press (1993), pp 22-41. (logic programming cannot express the behavior of interactive software systems)

[8] Wegner, P. Interactive Foundations of Object-Based Programming, *IEEE Computer*, 28-10 (1995), pp 70-72. (readable three-page presentation)

[9] Wegner, P. *Interactive Foundations of Computing*, Brown University Technical Report, October 1996. (detailed 40-page technical presentation)

[10] Wegner, P. Interactive Software Technology, in *Handbook of Computer Science and Engineering*, CRC Press (1997), Chapter 151. (applies interactive computing to interoperability, coordination, etc.)