

Interactive Models for Design of Software-Intensive Systems

Dina Goldin, David Keil¹

*Computer Science & Engineering Dept.
University of Connecticut
Storrs, CT USA*

Abstract

The two paradigm shifts from mainframes in the 1950s-1970s to personal computers in the 1980s and networked mobile devices in the 2000s can be conceptually modeled by paradigm shifts from algorithms to sequential interaction and then to distributed (multi-agent, collaborative) interaction. Interactive models provide a domain-independent unifying view of the design space for the design of software-intensive systems (DSIS).

The irreducibility of interactive behavior to algorithms is a primary cause of the software crisis. Moreover, it also explains the difficulty among both theorists and practitioners in establishing integrated principles and models of computation for DSIS technology. The artifacts of design are less often tools and more often systems that interact with other systems. These systems are increasingly complex and increasingly characterized by emergent, unpredictable behavior. By enunciating the interactive modeling principles for DSIS, our work provides a foundation on which the science of software design can be built and points out directions for future research.

Key words: Interactive computing, Models of computation,
Science of design, Software design, Software-intensive systems

1 Introduction

The paradigm shifts from mainframes in the 1950s-1970s to the personal computer technology of the 1980s and the networked computing devices of the 2000's can be conceptually modeled by paradigm shifts from algorithms to sequential interaction and then to distributed (multi-agent, collaborative) interaction. Interactive models provide a domain-independent unifying view of

¹ Email: {dgg,dkeil}@engr.uconn.edu

the design space for software-intensive systems, and will be fundamental to the new discipline of the *science of design*.

The term “Science of Design” was originally coined by the Nobel Laureate Herbert Simon. In his essay by the same name, published in [48], the central insight is that the design process is a *goal-driven search* for a path to an artifact that satisfies design constraints. He drew an analogy with the application of the scientific method, which is a search for process descriptions capable of leading to observed phenomena. Both the design process and the scientific method are *interactive*, with iterative refinements based on observation of partial results. Furthermore, the *goal* of design is typically to create systems of *interacting components*. The organizational structure of these systems and the interaction of their components are at the center of the designer’s concerns; this insight can be found in another essay of [48].

While Simon’s book predated complex software systems, and his observations on complex systems did not refer to software systems, it provides profound insights into today’s software-intensive systems. Thus, three decades ago, Simon posed the problem of a science of design as one that is fundamentally concerned with *interaction*. But this science will not be one of *software design* or even *human-computer interaction*; its central concern will be with *interaction* as such – among humans, between humans and computational systems, among computational systems, between computational systems and embedded physical devices, etc.

The ubiquity of interactive computing makes possible a science of design of a scope much broader than merely the design of software or of human-computer interaction. This science will be united by an approach of *embodied interaction* [16], where the objects of design (software and otherwise) *participate* in their environments. It will be interdisciplinary in origin (drawing from life and social sciences, among others), and driven in part by the users of interactive artifacts under design, with design conceived in a framework of *bounded rationality* [46]. A *theory of sequential and multistream interaction*, based on new formal models [26], will underpin this science. The ubiquity of computing in today’s systems enables empirical experiment as part of design of interactive systems that will be increasingly of an open and adaptive character, where *indirect interaction* [35,24] plays a significant role.

Overview. In this position paper, we discuss interactive modeling principles and research directions for DSIS technology. Section 2 discusses the interactive nature of the Science of Design. Section 3 shows the extension from algorithms to interaction along a number of dimensions. It examines various DSIS models: component models, interoperability, and design patterns. Section 4 reviews models of interaction for finite agents and includes subsections on models of computing, sequential interaction machines and multi-agent machines. Section 5 discusses indirect interaction, an important form of interaction for software-intensive systems that is not modeled by current approaches. We conclude in section 6.

2 A vision of a design science of the future

The science of DSIS will draw as much from the disciplines of economics, architecture and biology as from computer science. The software aspect of design will tend to be in the area of distributed, real-time and embedded (DRE) systems. But the principles involved will not be software- or computation-specific; they will be *domain-independent* and *interaction-related*.

2.1 Design of software vs. design of software-intensive systems

The key dimension of change in software-intensive systems is *interactiveness*. Whereas at one time computing was mostly *batch* – the execution of algorithms, with all input data supplied *a priori* and all output generated *afterward* – today *ongoing interaction* is ubiquitous. This interaction differs semantically from iterated batch computing in that each computing entity maintains a persistent state that evolves and enables computations to be driven by their entire input/output histories.

Accordingly, the design issues of greatest interest relate broadly to the *management of interaction* rather than the development of algorithms. While disciplines like human-computer interaction, networking, and the physical sciences have a place in the science of design, a *domain-independent* study of interaction will be an essential element of such a science. The following computing concepts are all fundamentally interactive and cannot be properly understood until more research is done on domain-independent interactive models and on the principles of interaction:

Emergence, Stigmergy, Self-organization, Evolutionary computation, Autonomous and rational agents, Iterated design, Coordination, Open systems

“Design for software-intensive systems” (DSIS) is widely interpreted as *software design*. However, software is only *one* aspect of software-intensive systems. The problem of design for such systems is converging with the design of systems *in general*, because systems in general are becoming increasingly software-intensive. Processors are embedded everywhere and networked computing is ubiquitous – in vehicles, buildings, appliances, workplaces, and other entities in which humans participate or with which they interact.

What is the origin of the tendency to reduce the notion of the design of software-intensive systems to software design? It may be in that software design has tended historically to occur independent of context, orienting to *non-interactive problems*, using tools appropriate to such problems and based on theoretical underpinnings that are restricted to algorithmic (function-based) rather than interactive models of computing. But designing a software-intensive system is designing for interaction. Hence the conceptual tools currently used in software engineering tend to be *inadequate* for the design of software-intensive systems. One kind of software-intensive system is a human work environment in which multiple computing devices are present – “interactive workspaces” such as the iRoom [34]. Other examples are web

services and sensor-controlled cars. Design for *deviceless interaction* via movements, gestures, etc. [60] requires formal models, even though the interacting entities are not computational ones.

Three trends appear to converge in software development: distributed development, participation of users in quality-assurance efforts, and development processes that are *evolution-oriented* [40]. These features of software development are likely to become features of the design of systems in general. The *condition* under which the potential for a science of general design of software-intensive systems (not just software design!) can be realized is that an interdisciplinary theory of *interaction* (computational and otherwise) exist and be applied. By enunciating the principles of interaction and formalizing models that characterize its expressiveness, our work provides a foundation on which the science of design can be built.

2.2 *Design as an interactive process of search*

[48] points out that design is a goal-driven search for a state that satisfies certain design constraints. The search uses afferent (sensory) and efferent (motor) channels and builds connections between them that lead to goal states. If actions combine “additively” (linearly) to attain goal states (good designs), then standard (first-order) logic suffices for design. If not, which is generally the case, then higher-order (temporal) logic is required.

Today it would be recognized that the role of higher-order logics in design is filled in part by *temporal logic* of reactive systems [39]. Both the design process and most design artifacts are reactive systems. We would add that the paths of action and percept that are the domain of temporal logic are *infinite*; a logic of infinite structures must break the bounds of first-order logic and inductive definitions, resorting to *coinductive* reasoning [5,32]. Part of the mathematics of any science of design will be *coinduction* and *coalgebras* [57].

Simon further points out that a design process that cannot simply *assemble* a design artifact in an *additive* way must often, to be efficient, conduct a branching search, “begin to explore several tentative paths.” Here may be the origins of the notion of *evolutionary design*, and of course that is how life forms came into existence. Moreover, Simon discusses the *resource allocation* necessary in a design process, since not all design paths may be explored. Today his remarks would be situated in the framework of *bounded rationality* [46]; a design process is what is known now as an *anytime algorithm* [7], a potentially infinite procedure that terminates when its results are considered to *suffice* (Simon’s term) relative to the computational resources expended.

Research in *collaborative design* has noted that the design process itself may occur within a complex system characterized by emergent behavior [37]. The design of an interactive system begins by deciding upon constraints on behavior, rather than attempting to specify all behavior [?]. This principle is strikingly illustrated in research in architecture, where the objective of design

is to build environments favorable to particular kinds of interaction with and within the environments. An architectural design framework implemented in software, such as Smart Objects [19], enables the interactive communication of design constraints among the project participants. The interactive nature of the design of safety-critical systems is evident in the *feedback* path linking the development of systems with the analysis of system failures. The notion of a *safety case*, summarized in a graphical notation such as GSN, offers formal support for analysis of feedback [31]. Generalized *interaction models* will offer further support.

A paradox in Plato’s *Meno* asks how we can recognize a desired object as one we seek, when we haven’t ever seen it before. Simon’s solution is that we preconceive only an *operational* description, not a detailed one, and we can easily test what we see in the world against this description, without ever having seen precisely the instance that is in front of us.

In the case where what we seek is a satisfactory *design*, we begin with a description of the qualities of what is to be designed (a specification) and seek “a process description of the path that leads to the desired goal” [48]. This search process is either *interactive* with an environment or is a simulation of interaction, because it proceeds by generating new possible solutions and by testing those solutions in the current environment. This *interactive* method of discovery or design is made more complex when the artifact under design is itself an interactive agent and, especially, when this artifact is to be tested in an environment that is likewise interactive and dynamic.

3 Models of interaction

3.1 Motivation and overview

The irreducibility of interaction to algorithms [58] implies that research goals like reducing computing to logic, or completely formal program development, are not merely hard but actually impossible. This irreducibility is a primary cause of the software crisis. It explains the difficulty among theorists and practitioners in establishing algorithmic principles and models of computation for the design of software intensive systems. Interactive models provide a high-level framework for an integrated approach to software design technology that can contribute to both substantive research and specific models. A research goal is to develop positive models of computation that contribute to the effectiveness of DSIS technology within the framework of these impossibility constraints.

Systematic exploration of the DSIS design space requires modeling abstractions for DSIS technology at a variety of levels and from multiple perspectives. Interactive models express interface behavior, multiple interfaces that share a state, and interoperation and collaboration among interfaces. Thus they offer a unifying framework for building the scientific foundations of the Science of

Design. Interactive principles underlie design representation, reasoning, analysis, decision-making, and synthesis. These principles are domain-independent, applying to software and non-software components alike. Compositional and adaptive behavior of independently developed components is also interactive. Interactive models are naturally suited for environments whose specification is incomplete and unreliable. Understanding of interaction will guide the design languages and modeling frameworks, and will serve to provide guidelines for partitioning large tasks into parts.

Models of interaction provide technical support for all areas of the program announcement – design theory, requirements and specifications, adaptability and evolution, design automation, and quality and productivity – that will greatly benefit from a domain-independent development of the principles of interactive computation. They bridge the technical gap between inward-looking “pure” computer science and outward-looking empirical computer science by extending models of finite computing agents from Turing machines to interaction machines that provide reactive and embedded services over time. This project aims to show that interactive models play an important role in DSIS technology.

3.2 Interaction as a unifying framework

The evolution of computer architecture from mainframes to networked PDAs, of software engineering from procedure-oriented to object-oriented and component-based systems, and of AI from logic-based to agent-oriented and distributed systems have followed parallel paths. Software technology has evolved through the following stages [55]:

1950s: machine language, assemblers, hardware-defined action sequences

1960s: procedure-oriented languages, compilers, programmer-defined action sequences

1970s: structured programming, composition of action sequences, algorithm composition

1980s: object-based languages, personal computers, sequential interaction architecture

1990s: structured object-based programming, networks, distributed interaction architecture

2000s: mobile and embedded devices, ubiquitous and pervasive computing systems

According to this capsule history, the 1950s through the 1970s were concerned with the development and refinement of algorithm technology for mainframes and noninteractive off-line computing. Sequential interaction became the dominant technology of the 1980s, while distributed interaction became the dominant technology in the 1990s. Whereas the shift from machine to procedure-oriented languages involves merely a change in the granularity of actions, the shift from procedures to objects is more fundamental, involving a

qualitative paradigm shift from algorithms to interaction. The extension from sequential to distributed interaction architectures requires a further fundamental paradigm shift in models of computation, especially relevant for DSIS. Models of sequential interaction (SIMs) and of multi-agent interaction (MIMs) are reviewed later in this paper.

Interaction is ubiquitous in computing today. It provides a unifying abstraction and a common basis for the study of the science of design, object-oriented programming, evolutionary computation, emergent behavior, AI agents, and empirical computer science.

3.3 *The Design of component-based Systems*

Though object-based programming has become a dominant technology, its foundations are still shaky. Everyone talks about it but no one knows what it is. “Knowing what it is” has proved elusive because of the implicit belief that “what it is” must be defined in terms of algorithms. Interactive models have the liberating effect of providing a broader framework for defining “what it is” than algorithms. Component-based software technology is even less mature than object-based technology: it is the technology underlying interoperability, coordination models, pattern theory, and the World-Wide Web. Knowing what it is in turn requires liberation from sequential object-based models.

Interoperability is the ability of two or more software components to interact despite differences in language, interface, and execution platform. It is a scalable form of reusability, extending the reuse of server resources to clients whose accessing mechanisms may be plug-incompatible with sockets of the server. As with electrical appliances, incompatibility of software plugs and sockets can be mediated by adapters. Software adapters are part of the interaction architecture rather than the functionality of a software system. Interoperability architectures like the *web services* technology are effectively elaborate adapters that provide both static and dynamic compatibility among heterogeneous components.

Structured programming technology was the basis of an attempt to formalize software engineering associated with Dijkstra and other researchers [14]. However, it ultimately proved too weak as a model for program structure, because the transition to component-based programming made procedural structured programming, based on composing programs with *while* statements and procedures, obsolete. Component-based systems have behavior that cannot be compositionally expressed in terms of the behavior of the components. Structured programming for actions (verbs) can be formally defined by function composition, while structured programming for components (nouns) is modeled by design patterns that have no compositional formal specifications [20]. As a consequence, the study of component composition is an art rather than a science.

Though compositionality is a desirable property for formal tractability of

programs, and has led to advocacy of functional and logic programming as a basis for computation, it limits expressiveness by requiring the whole to be expressible as the sum of its parts. Component-based systems exhibit noncompositional emergent behavior. There are inherent trade-offs between formalizability and expressiveness that are clearly brought out by the expressive limitations of compositionality. Arguments in the 1960s that go-tos are considered harmful [14] for formalizability can now be paralleled by arguments that compositionality is considered harmful for expressiveness.

3.4 *Interactive software technology*

Programming in the large (PIL) is not determined by size, since a program consisting of a sequence of a million addition instructions is not PIL. PIL is rather synonymous with interactive programming, differing qualitatively from programming in the small in the same way that interactive programs differ from algorithms. Embedded and reactive software-intensive systems that provide services over time are PIL, while noninteractive problem solving is not PIL even when the algorithm is complex and the program is large.

Open systems can be precisely defined as interactive systems: interactive models provide a tool for classifying forms of openness and for analyzing open-system behavior. The ability of models of interaction to precisely define imprecise notions that do not fit into the Turing paradigm of computation reinforces the view that the Turing notion of computing is too narrow and that models of interaction are an appropriate extension that better accounts for computing practice.

Principles of interactive computing provide a basis for the analysis and evaluation of mature software design and engineering models. We focus on component models, interoperability, and design constraints, which model interactive design, mediation, and specification. The account below provides a starting point for a deeper analysis of more mature versions of these models from the perspective of interactive modeling.

3.5 *Interoperability: mediated interaction*

Clients and servers from different software platforms or programming languages can talk to each other through mediators (adapters) that convert data formats. Mediation in information systems as an organizing principle for interoperation of heterogeneous components is reviewed in [59]. Though mediators can handle a wide range of differences in data formats and recognized differences of representation, like that between Cartesian and polar coordinates, the general problem of reusing procedure functionality is unsolvable, since functional equivalence is undecidable.

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

interface standardization: map client and server interfaces to a common representation

interface bridging: two-way map between client and server

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

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

3.6 Design patterns: interactive specification of design

Patterns are a general tool for describing regularities in any domain of discourse. In the context of interactive modeling, patterns are useful in the description of observable behavior of software components, static structure of software systems, and processes of product development and design. Design patterns are reusable regularities of products or processes of design. The product of design is a component whose observable behavior can in principle be specified as a pattern of interaction. Designers are concerned with regularities that help them create a design from a high-level specification. Design patterns specify high-level regularities at the level of the problem rather than machine-language interaction patterns.

Developing a high-level language for patterns of design is harder than developing high-level algorithmic languages. Design patterns are specified in [20] by the problem, the solution, and design trade-offs associated with the problem. This format for specifying design patterns is independent of both domain and granularity and can be specialized when applied to object-oriented

design. Object-oriented patterns are described by their scope, which specifies whether the pattern applies to classes or objects, and by their primary purpose, which may be to create a class or object (creational), to compose a structure out of components (structural), or to specify the interaction of a group of components (behavioral).

The model-view-controller (MVC) paradigm has been widely used to illustrate the role of patterns in design. It specifies design by three interdependent perspectives: a model that represents the application object, multiple views that capture syntactic interfaces, and a controller that defines the mode of execution. The model and controller correspond to the object and dynamic models of OMT, while MVC views are omitted in OMT but correspond to use cases in Jacobson's object design model [33]. Though the MVC paradigm is a design pattern, its granularity is too coarse for inclusion by [20] in their design catalog. MVC is described in terms of three component patterns:

observer: a pattern to describe one-to-many sharing of a resource (object) by multiple interfaces

composite: a pattern to describe composite nested structures

strategy: a pattern for run-time selection among multiple implementations of the same algorithm

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

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

Patterns facilitate codifying of design experience that has been perceived in other design domains, such as architecture, to be elusive and unformalizable. A systematic method of describing, cataloging, and using design patterns

provides clues to the process of design for both software systems and other artifacts.

4 Models of Finite Computing Agents

The radical claim that Turing machines are not the most powerful model of computation [54,58] is the basis for a paradigm shift in both the theory and practice of computing. The four subsections below present a perspective on models of computation, a brief introduction to sequential and multi-agent interaction machines, and a brief introduction to mathematical models of interaction. The goal is to establish a modeling framework that is both acceptable to theorists and usable in practice.

4.1 Models of Computation

Algorithms and Turing machines (TMs) have been the dominant model of computation for computer science, playing a central role in establishing computer science as a discipline and determining the scope and content of theoretical computer science. The technology shift from mainframes to networks has widened the gap between algorithmic theory and interactive practice. Models of interaction challenge the dominance of the TM model on the grounds that TMs are too weak to model interactive systems [58].

The Church-Turing thesis, which asserts that Turing machines (TMs) and the lambda calculus express the intuitive notion of effectively computable functions, led to the widespread belief that effective computability of actual computers was expressed by TMs. Because researchers believed that questions of expressiveness of finite computing agents had been settled once and for all, they did not explore alternative notions of expressiveness. They focused instead on questions of complexity, performance, and design for the fixed notions of computability and expressiveness of Turing machines. The hypothesis that interactive finite computing agents are more expressive than algorithms opens up a research area that had been considered closed, requiring fundamental assumptions about models of computation to be reexamined.

Turing's seminal paper [50] was not intended to establish TMs as a universal model for computing but on the contrary to show undecidability and other limitations of TMs. Turing actually distinguishes between automatic machines (*a*-machines, now known as TMs) and interactive choice machines (*c*-machines) in [50] and excludes *c*-machines in fashioning his model of computation, presumably because they do not conform to his deliberately limited notion of computability. TMs model "automatic" computation, while interaction machines (IMs) extend the notion of what is computable to nonautomatic (interactive) computation. IMs can model TMs with oracles, which Turing showed to be more powerful than TMs in his Ph.D. thesis [51].

The irreducibility of interactive systems to algorithms was noticed by [39]

for reactive systems, by [41] in the context of process models, by [46] for intelligent agents, and by many other researchers. We identify interaction as a domain-independent and language-independent cause of greater computational expressiveness, viewing interaction as normal behavior that needs to be analyzed rather than as exceptional unanalyzable behavior.

4.2 Interaction Machines

Interaction machines for finite agents have a role comparable to that of Turing machines (TMs) for algorithms. Sequential interaction architecture is modeled by sequential interaction machines (SIMs), while *persistent Turing machines* (PTMs) are a canonical model for sequential interaction whose role parallels that of TMs for algorithms [26]. Distributed interaction architecture is modeled by multi-agent interaction machines (MIMs). Expressiveness of finite agents is specified by observational equivalence, and is measured by the level of refinement of their *observational equivalence classes*. With this technique, SIMs are shown to be more expressive than TMs [26] but are conjectured to be less expressive than MIMs.

Two finite agents are distinct relative to a class of observers (testers) if there is an observation, called a distinguishability certificate, that distinguishes them. Equivalence can be finitely falsified [43] by a distinguishability certificate but cannot be finitely verified. Greater distinguishing power implies both the ability of observers to distinguish a greater range of behavior and the ability of finite agents to exhibit a greater range of behavior (solve a larger class of problems). Observation-based expressiveness defines systems to be more expressive if they can deal with (adapt to) a larger range of inputs (observations) from their environment. Late (dynamic) binding of interactive inputs is more expressive than early (static) binding not because functions have greater transformation power but because they have richer input domains that cannot be inductively specified by strings. Greater distinguishability by observers implies a greater range of behavior by agents, which in turn implies a greater input domain for interactive streams over predetermined strings. Computation power is extended beyond Turing computability by extending input domains from strings to streams.

Domain elements of functions computable by TMs are completely specified by predetermined string at the start of a computation. Computable functions $f : X \rightarrow Y$ have a domain X of input strings (integers) and determine a unique $y = f(x)$ for each x in X . Stream transductions are not functions from integers to integers because streams have incrementally generated domains not representable by integers. Streams have the form $(a_1, o_1), (a_2, o_2), \dots$, where output o_k is computed from action a_k but precedes and can influence a_{k+1} [11]. This *input-output coupling* violates the separation of domains and ranges, causing dynamic dependence of inputs on prior outputs that occurs in interactive question-answering, dialog, and control processes.

The computational extension from string to streams is mathematically expressed by an extension from inductive to coinductive definition and reasoning principles. Streams and string have inductively specified domains while streams have coinductively specified domains that are not enumerable. Turing machines are mathematically modeled by induction-based principles of constructive mathematics while interaction machines are modeled by an emerging coinductive paradigm [5] that expresses observation (testing) of objects in an already constructed world.

Question-answering has been used both by logicians [36] and by Turing [52] as a framework for investigating the expressiveness of models. Finite agents that transform string can answer only enumerable classes of inductively specifiable questions, while finite stream-processing agents can distinguish among nonenumerable questions (situations) [5]. For example, finite agents can distinguish among the nonenumerable real numbers in the sense that any pair of distinct reals, represented by infinite binary string, has a finite distinguishability certificate.

The irreducibility of interaction to algorithms and of computation to first-order logic reinforces the view that tools of algorithm analysis and formal methods cannot, by themselves, be used to establish the science of design. Software tools for design patterns, life-cycle models, embedded systems, and collaborative planning are not modeled by algorithms and their behavior cannot inherently be expressed by first-order logic. Fred Brooks' claim [10] that there is no silver bullet for systems can be proved if we define "silver bullet" to mean "algorithmic or first-order logic specification".

Computable functions are too rigid an abstraction to model the complete behavior of actual computing systems, sacrificing the ability to model interaction and time to realize tractability. Negative impossibility results useful in avoiding expenditures on unsolvable problems lead to positive formal models of computation that provide a framework for a technology of interactive computing. New classes of models are needed to express the technology of interaction, since software technology has outstripped algorithm-based models of computation.

4.3 *Persistent Turing Machines*

In this section, we discuss *Persistent Turing machines* (PTMs), a new way of interpreting Turing-machine computation, one that is both interactive and persistent; this is an overview of [26]. A PTM is a nondeterministic 3-tape Turing machine (N3TM) with a read-only input tape, a read/write work tape, and a write-only output tape. Upon receiving an input token from its environment on its input tape, a PTM computes for a while and then outputs the result to the environment on its output tape, and this process is repeated forever. A PTM performs *persistent computations* in the sense that a notion of "memory" (work-tape contents) is maintained from one computation step

to the next, where each PTM computation step represents an N3TM computation.

Persistence extends the effect of inputs. An input token affects the computation of its corresponding macrostep, including the work tape. The work tape in turn affects subsequent computation steps. If the work tape were erased, then the input token could not affect subsequent macrosteps, but only “its own” macrostep. With persistence, an input token can affect all subsequent macrosteps; this property is known as *history dependence*.

Our treatment of PTMs has proceeded along the following lines. We first formalized the notions of interaction and persistence in PTMs in terms of the persistent stream language (PSL) of a PTM. Given a PTM, its persistent stream language is coinductively defined to be the set of infinite sequences (interaction streams) of pairs of the form (w_i, w_o) representing the input and output strings of a single PTM computation step. Persistent stream languages induce a natural, stream-based notion of equivalence for PTMs. *Decider PTMs* are an important subclass of PTMs; a PTM is a *decider* if it does not have divergent (non-halting) computations.

We then defined a very general kind of effective transition system called an *interactive transition system* (ITS), and equipped ITSs with three notions of behavioral equivalence: *ITS isomorphism*, *interactive bisimulation*, and *interactive stream equivalence*. We showed that ITS isomorphism refines interactive bisimulation, and interactive bisimulation refines interactive stream equivalence.

Our first result concerning ITSs is that the class of ITSs is isomorphic to the class of PTMs, thereby allowing one to view PTMs as ITSs “in disguise”. A similar result is established for decider PTMs and decider ITSs. These results address a question heretofore left unanswered concerning the relative expressive power of Turing machines and transition systems. Until now, the emphasis has been on showing that various kinds of process algebras, with transition-system semantics, are capable of simulating Turing machines in lock-step [8,13,4,6,12,53]. The other direction, namely – what extensions are required of Turing machines so that they can simulate transition systems? – is answered by our results.

We also defined an infinite hierarchy of successively finer equivalences for PTMs over finite interaction-stream prefixes and showed that the limit of this hierarchy does *not* coincide with PSL-equivalence. The presence of this “gap” can be attributed to the fact that the transition systems corresponding to PTM computations naturally exhibit *unbounded nondeterminism*. This is an important phenomenon for *specification*; for example, modeling unbounded nondeterminism is crucial for supporting refinement between dialects of *timed* and *untimed* CSP [9]. In contrast, it is well known that classical Turing-machine computations have bounded nondeterminism, i.e., any nondeterministic TM can produce only a finite number of distinct outputs for a given input string.

We further introduced the class of *amnesic* PTMs and a corresponding notion of amnesic stream language (ASL). In this case, the PTM begins each new computation with a blank work tape. We showed that the class of ASLs is strictly contained in the class of PSLs. We additionally showed that ASL-equivalence coincides with the equivalence induced by considering interaction-stream prefixes of length one, the bottom of our equivalence hierarchy; and that this hierarchy collapses in the case of amnesic PTMs. ASLs are representative of the classical view of Turing-machine computation. One may consequently conclude that, in a stream-based setting, the extension of the Turing-machine model with persistence is a nontrivial one, and provides a formal foundation for reasoning about programming concepts such as objects with static attributes.

Finally, the notion of a *universal PTM* was introduced and a proof of its existence presented. Analogously to a *universal Turing machine*, a universal PTM can simulate the behavior of an arbitrary PTM. We also introduced the class of *sequential interactive computations*, illustrating it with several examples. In an analogous fashion to the Church-Turing Thesis, we hypothesized that anything intuitively computable by a sequential interactive computation can be computed by a persistent Turing machine. This hypothesis, when combined with results earlier in the paper, implies that the class of sequential interactive computations is more expressive than the class of algorithmic computations, and thus is capable of solving a wider range of problems.

4.4 *Multiple-interface model: concurrent interaction*

Multi-stream distributed interaction machines (MIMs) are finite agents that interact with multiple autonomous agents. Examples of MIMs are: the World-Wide Web, distributed databases, multi-agent games, multi-agent negotiation. MIMs allow higher-level behavior such as collaboration and coordination to be precisely modeled. Airline reservation systems are prototypical applications with multiple autonomous, concurrently acting agents share common information. They may have multiple instances of each 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

SIM interaction has the property that an observer (stream) together with the observed system is closed, while autonomous (multi-stream) MIMs cannot be closed by composition with any single agent because of autonomous interaction with other agents. MIMs are shown to be more expressive than and not reducible to sequential interaction, contrasting with the fact that multitape

TMs are no more expressive than single-tape. Adding autonomous streams (observation channels) to a finite agent increases expressiveness, while adding noninteractive tapes simply increases the structural complexity of predetermined inputs, and does not increase expressive power.

Though there is strong evidence that MIM behavior is not expressible by SIMs, MIM behavior is harder to formalize and greater expressiveness is harder to prove. MIMs support the behavior of nonserializable transactions and true concurrency, while SIMs support only serializable transactions and interleaving concurrency. Concurrent execution of MIMs is dual to the sequential execution of SIMs. Extension of algorithmic to interactive concurrency for MIMs parallels the extension of algorithmic to interactive sequential computation of SIMs. The extension from SIM interaction between two agents to MIM interaction among three or more agents parallels the extension in physics from the tractable two-body problem to the intractable three-body and n-body problems.

A MIM is a tool for studying the interaction of $k+1$ entities, where one plays a special role as an observed system and the remaining entities are observers, modeled by streams. For interacting SIMs, $k = 1$ and the observer with the observed system form a closed system. When the observed system is a MIM, $k > 1$ and a primary observer representing the experimenter is distinguished from $k - 1$ secondary observers. MIMs appear nondeterministic from the viewpoint of a primary observer who cannot predict or control the effects of secondary observers and may be unaware of their presence. Observers perceive MIMs as subjectively nondeterministic even when, viewed by an omniscient multi-agent observer (God), they are objectively deterministic.

The argument that MIMs are more expressive than SIMs is based on the fact that input-output actions of streams are transactions, and that transactions of multiple autonomous streams cannot be serialized by a single stream. The condition that the transactional integrity of multiple streams can be preserved under merging corresponds to serializability. MIMs support the behavior of nonserializable transactions, which are a form of true concurrency [44]. The result that MIMs are not expressible by SIMs is related to the nonexpressibility of true concurrency by interleaving.

5 Indirect interaction

Indirect interaction occurs when computing entities communicate by acting on, and observing, the persistent state of their *common environment* [35]. Characteristics that distinguish indirect from direct interaction [24], such as late binding of communication recipients, give it a different semantics from direct interaction. Indirect interaction is an important phenomenon in many fields, including distributed artificial intelligence, software engineering, programming languages, sociology, and economics. It is an important aspect of the design of many systems.

5.1 *The characteristics of indirect interaction*

Direct interaction is interaction via *messages*; the recipient's identifier is specified in the message [35]. But often, agents affect each other's computation without interacting directly, when one of them makes changes to their shared environment that the other later observes. This constitutes *indirect interaction*. *Indirect interaction* is interaction via *persistent, observable state changes*; recipients are any agents that will observe these changes.

Indirect interaction has many characteristics not present in message passing.

- *late binding of recipient*: the identity of the observer of given state changes may be determined by dynamic events occurring after the change is made;
- *anonymity*: even when the recipient's identity is bound statically, it need not be known to the originator of the state change;
- *time decoupling (asynchrony)*: due to persistence of the environment, there may be a delay between the change and its observation, the length of delay may be determined by dynamic events;
- *space decoupling*: indirect interaction of mobile agents need not imply co-location; the first may leave after making changes, and the second later arrive to observe these changes;
- *non-intentionality*: indirect interaction does not require an intent to communicate, nor an awareness that interaction is occurring; the agents may be changing their environment or observing it simply as part of carrying out their own autonomous task;
- *analog nature*: for embedded or robotic agents, the real world can serve as the medium of indirect interaction. Observations of the real world lack the fidelity of digital shared values and data structures, and such systems are in essence *hybrid*.

The *decoupling* between sender and receiver in indirect interaction makes it natural for systems consisting of a multitude of simple agents whose ability to perceive and act upon their environment is *localized* [61].

5.2 *Examples: Dining philosophers and foraging ants*

Dining Philosophers [15] is a classic problem in concurrency and shared resources. Several philosophers sit around a circular table, with one chopstick between each pair of diners. They autonomously decide when to eat, and pick up the two chopsticks next to them; once full, they put down the chopsticks and think, until hungry again. If each philosopher simultaneously picks up the left (or right) chopstick, and holds it until the right one is available, they will all starve. The goal is to let the philosophers carry on while avoiding starvation. Described more abstractly, this problem is to define a protocol for a ring-shaped arrangement of communicating processes, each communicating

only with its two neighbors, such that all processes are allowed to move forward under the constraint that no two adjacent ones may execute simultaneously.

Dining Philosophers can be modeled as direct interaction between philosophers and utensils, as in the original solution and in [3]. However, the semantics of the problem are of interaction between diners, not between diners and utensils. In the semantics of this problem, chopsticks are not autonomous computing entities, like philosophers; they are *passive*, initiating no action. Their only role is to reflect the states of the philosophers next to them.

Indirect interaction allows us to model this problem more naturally. Each philosopher interacts with his neighbors *indirectly*, by changing the state of their shared chopstick (on-table or with-diner). This problem has the following properties that are ideal for modeling with indirect interaction:

- *Anonymity*: diners need not know each other’s names or even whether their neighbors exist;
- *Asynchrony*: diners don’t necessarily pick up a chopstick as soon as it is put down;
- *Non-intentionality*: diners pick up chopsticks to eat with and put them down to think, with no intent to communicate;
- *Locality*: diners can only see neighboring chopsticks.

The other properties of indirect interaction discussed in Section 5.1 (*late binding*, *space decoupling*, and *analog nature*) are not present in this example. However, they are present in the Foraging Ants example, later.

To exemplify the properties of *late binding* and *space decoupling*, we can create a more advanced version of this problem to involve *mobility*, where philosophers may leave or switch places. Their neighbors continue to rely on the state of the chopstick to make their decisions, not aware of the change in their neighbor’s status. To exemplify the property of *analog nature*, we can replace binary chopstick states by analog observations, such as perhaps how clean the chopstick is.

Another example of indirect interaction is foraging ants. Ant colonies solve the problem of efficiently foraging for food sources by indirect interaction, where each ant deposits *pheromones* as it carries food, and each ant tends to follow the pheromone trails it finds [49]. Ants strengthen existing trails in a way that reinforces shorter paths. The ants interact indirectly via these pheromone trails.

A centralized way to accomplish the same task would entail a “command center” which is aware of the location of food sources, and is in charge of deciding which location each ant should exploit next. Without such a command center, decentralized direct interaction mechanisms would force the ants to communicate via a decentralized message system of great complexity, developing special “message-handling” ants, needing an identification mechanism, learning a language of discourse, broadcasting of all messages, analyzing all

the messages, etc.

5.3 *Toward a formalization of indirect interaction*

The above examples, of ants interacting via a real-world environment, and philosophers interacting via binary semaphores, present two very different applications of indirect interaction. It occurs in many fields under many names, both within and outside computer science:

- *Operating systems*: Processes exchange information via semaphores in shared memory;
- *Programming languages*: The Linda language uses *tuple spaces* to enable coordination by indirect interaction [21];
- *Anatomy*: Cells exchange information via hormones in the bloodstream;
- *Social biology*: In stigmergy, social insects interact indirectly by leaving trails of pheromone chemicals [49];
- *Sociology*: Most group dynamics consist of actions or percepts whose destinations or sources are other than one's immediate partner in a communication;
- *Multiagent systems*: Agents communicate indirectly either through intermediary agents or through changes in the environment;
- *Economics*: the storage and publication of *stock market listings* enables large numbers of buyers and sellers to interact indirectly to negotiate prices.

The persistent character of the environment (i.e., the ability to make and observe persistent changes to it) is a prerequisite for indirect interaction. All other properties are a natural consequence. While most of these properties can be simulated with message passing by employing special protocols, an eventual formalization of indirect interaction would allow us to model them explicitly, without a need for an intermediate protocol layer.

The Foraging Ants example shows that indirect interaction for highly multiagent decentralized systems is both more scalable and more effective than direct interaction. It is also better at reflecting the semantic distinction between active agents and their passive interaction medium. Models of open systems based solely on direct interaction are inadequate even when the medium of interaction is digital; the argument is more compelling when the interaction takes place via an analog environment such as the real world. If the characteristics of a problem include indirect interaction, an adequate model must reflect these features. A formalization of indirect interaction will contribute towards truer models for open systems. Domain independent models of indirect interaction will provide an underpinning for new forms of reasoning for the Science of Design and beyond.

6 Conclusions

Algorithms and Turing machines (TMs) have been the dominant model of computation for computer science, playing a central role in establishing computer science as a discipline and determining the scope and content of theoretical computer science. The technology shift from mainframes to pervasive computing has widened the gap between algorithmic theory and interactive practice. The work on interactive models promises to bridge the gap between theory and practice by showing that interactive models can play a significant practical role in mainstream software technology. Models of interaction bridge the technical gap between inward-looking “pure” computer science and outward-looking empirical computer science by extending models of finite computing agents from Turing machines to interaction machines that provide reactive services over time. They provide technical support for “broadening the horizons of academic research and undergraduate education while sustaining the core effort of CS&E,” as originally proposed in [42].

Design is an embryonic science. The artifacts of design are less often tools and more often systems that interact with other systems. These systems are increasingly complex and increasingly characterized by emergent, unpredictable behavior. Their design is currently more an art than a science. Tools and theoretical underpinnings from outside the mainstream of software engineering are needed. If the design of software-intensive systems is to be a science, then it must undergo a paradigm shift, in which the central role played by *interaction* in defining design problems is recognized.

References

- [1] P. Aczel. *Non Well-Founded Sets*. CSLI Lecture Notes #14, Stanford, 1988.
- [2] A. Anton & C. Potts. Functional Paleontology: System Evolution as the User Sees it. *Proc. Int’l Conf. Software Eng., Toronto, May 2001*.
- [3] F. Arbab. Reo: A Channel-Based Coordination Model for Component Composition. CWI Report SEN-0203, 2002.
- [4] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen’s fair abstraction rule. *Theoretical Comp. Science*, 51(1/2):129-176, 1987.
- [5] J. Barwise and L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI Lecture Notes #60. CSLI Publications, 1996.
- [6] B.S. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can’t be traced. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, 1988.
- [7] M. Boddy and T. Dean. Decision-Theoretic Deliberation Scheduling for Problem Solving in Time-Constrained Environments. *Artificial Intelligence* 67(2)(1994), pp. 245-286

- [8] G. Boudol. Notes on algebraic calculi of processes. In K. Apt, ed., *Logics and Models of Concurrent Systems*, pages 261-303. LNCS, Springer-Verlag, 1985.
- [9] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating sequential processes. In *Proceedings, NSF-SERC Seminar on Concurrency*. Springer Verlag, 1985.
- [10] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1995.
- [11] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer Verlag, 2001.
- [12] P. Darondeau. Concurrency and computability. In I. Guessarian, ed, *Semantics of Systems of Concurrent Processes*, Proc. LITP Spring School on Theoretical Computer Science, La Roche Posay, France, LNCS 469, Springer-Verlag 1990.
- [13] R. de Simone. Higher-level synchronizing devices in meije-sccs. *Theoretical Comp. Science*, 37:245-267, 1985.
- [14] E. Dijkstra. *The Discipline of Programming*. Prentice Hall 1976.
- [15] E. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Inform.* 1, pp. 115-138, 1971.
- [16] P. Dourish. *Where the Action is: The Foundations of Embodied Interaction*. MIT Press, 2001.
- [17] C. L. Dym, J. W. Wesner and L. Winner. A Report on Mudd Design Workshop III: Social Dimensions of Engineering Design. *Journal of Engineering Education* 92 (1), 105-107, January 2003.
- [18] E. Eberbach, D. Goldin, P. Wegner. Turing's Ideas and Models of Computation. Book chapter in Christof Teuscher, Ed., *Alan Turing: Life and Legacy of a Great Thinker*, Springer 2004.
- [19] D. Eggink, M. D. Gross, E. Do. Smart Objects: Constraints and Behaviors in a 3D Design Environment. *Proc. Education in Computer Aided Architectural Design in Europe (ECAADE)*, Helsinki, Finland, 2001, pp. 460-465.
- [20] E. Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [21] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Comm. ACM* 35(2), 1992, pp. 97-107.
- [22] D. Goldin. Persistent Turing Machines as a Model of Interactive Computation. In: K-D. Schewe and B. Thalheim (Eds.), *FoIKS'2000* (First Int'l Symp. on Foundations of Information and Knowledge Systems). LNCS 1762, Springer-Verlag, Berlin 2000, pp.116-135.
- [23] D. Goldin, D. Keil. Interaction, Evolution, and Intelligence. Proc. *CEC'01* (Congress on Evolutionary Computation), May 2001, Seoul, Korea.

- [24] D. Goldin, D. Keil. Towards a Domain-Independent Formalization of Indirect Interaction. Proc. *Theory and Practice of Open Computational Systems* (TAPOCS), June 2004.
- [25] D. Goldin, D. Keil, P. Wegner. “An Interactive Viewpoint on the Role of UML.” Book chapter in *Unified Modeling Language: Systems Analysis, Design, and Development Issues*. Eds. Keng Siau, Terry Halpin. Idea Group Publishing 2001.
- [26] D. Goldin, S. Smolka, P. Attie, E. Sonderegger. Turing Machines, Transition Systems, and Interaction. *Information & Computation Journal*, Nov. 2004.
- [27] D. Goldin, S. Srinivasa, V. Srikanti. Active Databases as Information Systems. Proc. *8th Int’l Database Engineering & Applications Symp.*, Coimbras, Portugal, July 2004, IEEE Press.
- [28] D. Goldin, S. Srinivasa, B. Thalheim. Information Systems = Databases + Interaction: Towards Principles of Information System Design. Proc. *ER’00*, Salt Lake City, Oct. 2000.
- [29] D. Goldin, P. Wegner. Paraconsistency of Interactive Computation’. Proc. *Workshop on Paraconsistent Computational Logic* (PCL), Denmark, July 2002.
- [30] W. S. Greenwell, J. C. Knight and E. A. Strunk. Risk-Based Classification of Incidents. Proc. *IRIA 03 Workshop on Investigation and Reporting of Incidents and Accidents*, Williamsburg, VA, Sep. 2003.
- [31] W. S. Greenwell, E. A. Strunk, and J. C. Knight. Failure Analysis and the Safety-Case Lifecycle. *IFIP Working Conf. on Human Error, Safety and System Development* (HESSD), Toulouse, France, August 2004.
- [32] B. Jacobs, J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin* 62, 1997.
- [33] I. Jacobson. *Object-Oriented Software Engineering*. Addison-Wesley/ACM Press, 1991.
- [34] B. Johanson, Armando Fox, Terry Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, April-June 2002.
- [35] D. Keil, D. Goldin. Modeling Indirect Interaction in Open Computational Systems. Proc. *Theory and Practice of Open Computational Systems* (TAPOCS), June 2003.
- [36] S. C. Kleene. Turing’s Analysis of Computability, and Major Applications of It. In Rolf Herken Ed., *The Universal Turing Machine - A Half-Century Survey*, Springer Verlag, 1994.
- [37] M. Klein, P. Faratin, H. Sayama, Y. Bar-Yam. The Dynamics of Collaborative Design: Insights From Complex Systems and Negotiation Research. *Concurrent Engineering Research and Applications J.* 12 (3), 2003.

- [38] H. Li, S. Krishnamurthi and K. Fisler. Modular Verification of Open Features Through Three-Valued Model Checking. Accepted to the *Journal of Automated Software Engineering*, July 2003.
- [39] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.
- [40] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. *Proc. 26th IEEE/ACM Int'l Conf. on Software Eng. (ICSE)*, Edinburgh, Scotland, May 2004.
- [41] R. Milner. Operational and Algebraic Semantics of Concurrent Processes. *Handbook of Theoretical Computer Science*, J. van Leeuwen, editor, Elsevier, 1990.
- [42] National Research Council. *Computing the Future*. National Academy Press, 1994.
- [43] K. Popper. *The Logic of Scientific Discovery*. Harper Torchbooks, 1965.
- [44] V. Pratt. Chu Spaces and their Interpretation as Concurrent Objects, in *Computer Science Today: Recent Trends and Developments*, Ed. Jan van Leeuwen, LNCS #1000, 1995.
- [45] J. Rumbaugh, M. Blaha, William Premerlani, Frederick Eddy, and William Lorenzen. *Object-Oriented Modeling and Design*. Prentice Hall, 1990.
- [46] S. Russell and E. Wefald. *Do the Right Thing: Studies in Limited Rationality*. MIT Press, 1991.
- [47] J. Rutten. A Tutorial on Coalgebras and Coinduction. *EATCS Bulletin* 62, 1997.
- [48] H. Simon. *The Sciences of the Artificial*. MIT Press, 1969.
- [49] G. Theraulaz and E. Bonabeau. A Brief History of Stigmergy. *Artificial Life* 5, pp. 97-116, 1999.
- [50] A. Turing. On Computable Numbers with an Application to the Entscheidungsproblem. *Proc. London Math Soc.* 2 (42), pp. 230-265, 1936.
- [51] A. Turing. Systems of Logic Based on Ordinals. *Proc. London Math. Soc.*, 1939.
- [52] A. Turing. Computing Machinery and Intelligence. *Mind*, 1950.
- [53] F. W. Vaandrager. Expressiveness results for process algebras. *Technical Report CS-R9301*, Centrum voor Wiskunde en Informatica, Amsterdam, 1993.
- [54] P. Wegner. Why Interaction is More Powerful than Algorithms. *Comm. of the ACM*, May 1997.
- [55] P. Wegner and D. Goldin. Interaction as a Framework for Modeling. In Chen, et al (Eds.) *Conceptual Modeling: Current Issues and Future Directions*, LNCS 1565, Apr. 1999.

- [56] P. Wegner, D. Goldin. Coinductive Models of Finite Computing Agents. *Electronic Notes in Theoretical Computer Science*, March 1999.
- [57] P. Wegner and D. Goldin. Mathematical Models of Interactive Computing, *Brown Univ. CS Tech. Rep. 99-13*, 1999.
- [58] P. Wegner, D. Goldin. Computation Beyond Turing Machines. *Comm. ACM*, Apr. 2003.
- [59] G. Wiederhold. Mediation in Information Systems. *Computing Surveys*, June 1995.
- [60] T. Winograd. Towards a Human-Centered Interaction Architecture. *In HCI in the New Millennium*, John Carroll, Ed., Addison Wesley, 2000.
- [61] F. Zambonelli and H. Parunak. Signs of a Revolution in Computer Science and Software Engineering. *Proc. ESAW02*, 2002.