An Interactive Viewpoint on the Role of UML

Dina Goldin — Univ. of Massachusetts / Boston

David Keil — Univ. of Massachusetts / Boston

Peter Wegner — Brown University

Table of Contents

Abstract

The role of the Unified Modeling Language (UML) is to model interactive systems, whose behaviors emerge from the interaction of their components with each other and with the environment.
Unlike traditional (algorithmic) computation, interactive computation involves infinite and dynamic (late binding) input/output streams. Algorithmic tools and models do not suffice to express the behavior of today's interactive systems, which are capable of self-reconfiguring and adapting to their environment.
Whereas procedural languages may express precise designs of closed processes, UML provides support for the inherently open-ended preliminary steps of system analysis and specification, which are becoming increasingly complex. Interactive systems require dynamic models where interaction has first-class status, and where the environment is modeled explicitly, as actors whose roles constrain the input patterns.
UML's interaction-based approach to system modeling fits well with the encapsulation-based OO approach to implementation. By coupling these approaches, the software engineering process can provide a complete solution to system design and implementation. A theoretical framework for modeling interactive computing can strengthen the foundations of UML and guide its evolution.

1. Introduction

The *Unified Modeling Language* (UML) emerged in response to a need for a notation (a visual language) that can express the behaviors of today's *interactive* computing systems and that can guide in constructing them. In the UML framework, software design entails building an object-oriented representation of a system, as well as of its *environment*, e.g. its users (modeled as *actors*).
Interactive systems such as modeled with UML represent a new paradigm in computation that inherently cannot be modeled using traditional, or *algorithmic*, tools. At the heart of the new computing paradigm is the notion that a system's job is not to *transform* a single *static* input to an output, but rather *to provide an ongoing service* [We2]. The service-providing nature of present-day systems was specifically noted by the Object Management Group (OMG) in defining the UML standard [OMG1].
When a system is viewed as a service provider, the interaction between the system and its environment

becomes an integral part of the computing process. UML presents a uniform domain-independent framework for modeling the different interactions present in today's systems: interactions among objects or software components, interactions between users and applications, interactions over networks (including the Internet), and interactions among embedded devices.

Programs that work non-interactively, transforming a given input to an output by a series of steps, represent the traditional, or *algorithmic* paradigm of computation (Figure 1). Theoretical tools for modeling algorithmic computation include *Turing Machines* [HU], recursive function theory, and the lambda calculus, which all define the same set of *computable* functions [Ba]. Algorithmic computation is present throughout interactive systems modeled by UML, at the low implementation level, but the entire system cannot be expressed by such computation. Models of interactive computation are more recent; they include the Calculus of Concurrent Systems (CCS) [Mi], input/output automata [LMWF], and Sequential Interaction Machines [WG2, WG3] that maintain persistent state information between interaction steps.

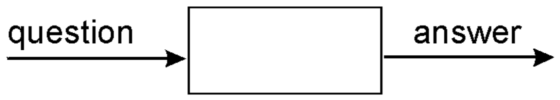

Figure 1. Algorithmic computation

A software system modeled by UML is a *computing entity.* System components and objects are also computing entities:

> **Computing entity**: a finitely specifiable software system, component, or object that is being modeled by UML. Entities may contain sub-entities, or be a part of a larger entity.

Once a system is implemented, its *external behavior* emerges out of its interactions with its environment:

> **Environment of an entity**: the producer of *inputs* for the computing entity, and the consumer of its *outputs*; it is outside the entity, interacting with it via inputs and outputs.

> **Actor**: an active role player in a given entity's environment; for a system, actors are usually system users, but they may be other systems, software or hardware.

> *The desired behavior of a computing entity is determined by the service that it provides to its environment, whether it is the user or other computing entities.*

A system's internal behavior, which affects the external behavior indirectly by affecting the values of outputs, emerges out of the internal interactions among its components. In this sense, systems and their components are like organizations and their subunits (departments, teams, or individuals), which function through the interactions among these subunits, and between the organization and the external world.

Given the similarity between software systems and business organizations, it is not surprising that UML is also applied to model organizations in the context of business process reengineering [Ja2]. However, the present work's focus is on software systems. We discuss how:

- through use case diagrams and other interaction-based diagrams, UML models a system whose behavior is defined by its observable external interaction with its environment (actors);
- subsystems, and objects internal to a system, have their own interactions with their environments, modeled in UML with state, activity, and interaction diagrams;
- UML models external occurrences with events, which may trigger actions;
- concurrency and parallelism can give rise to *multi-actor interaction*, which is a richer kind of behavior than sequential interaction and is modeled in UML in various ways;
- UML reflects a paradigm shift from algorithms toward interaction;
- interaction machines provide a theoretical foundation for formalizing interactive computation.

2. The behavior of systems

The service-providing role of systems and subsystems modeled by UML is explicitly noted by OMG [OMG1]. In requesting an instance of service, the actor is not concerned with the implementation details or the internal computations of the system*, within the active sub-system boundary.* Nobel prize winner Herbert Simon noted thirty years ago that computers, just like mechanical systems and social organizations, are defined by their behavior, not their internal structure:

> "almost no interesting statement that one can make about an operating computer bears any particular relation to the specific nature of the hardware. A computer is an organization of elementary functional components in which, to a high approximation, only the function performed by those components is relevant to the behavior of the whole system." [Si]

When discussing computers, Simon mostly referred to hardware systems. Computing at that time followed a strictly algorithmic paradigm, with procedural code executed in batch mode. It was hard if not impossible to visualize today's software systems, which UML was created to model, ones that fit Simon's notion of a *complex system* – an interactive system whose behavior emerges out of its interactions. By shifting the focus to the relationships between actors and the entity (computing agent), UML is able to specify software systems that fit Simon's description of a complex system.

The use-case concept models a system and its behavior by specifying the *nouns*, the actors and the entities, rather that the *verbs*, or procedures. This is an *object-oriented* approach to design. A system or subsystem is specified through its external behavior rather than its internal structure; externally determined patterns of interaction are not constrained by any specific algorithm specification. The modeling is at the level of *interfaces*, which specify interactions. The behavior is not modeled explicitly, but emerges as a result of interactions between the system and its environment.

The implementation/interface distinction corresponds to the duality of *internal hidden structure* (configuration and values) versus *external observable behavior* (external interactions). This duality holds both for actors in the environment and the entities in the computing system:

> *The internal structure of an actor is not necessarily known to the entity with which the actor interacts, and vice versa.*

Two systems or interactive entities may be considered equivalent if a *bisimulation* (*observational equivalence*) relation exists between them [Mi1]. This is a relative notion of equivalence, defined in terms of *observable behavior*, first proposed by Moore [Mo] in the context of *finite transducers*, which can be considered the simplest interactive systems [GK]. When two entities have the same observable behavior with respect to their environment, they are *indistinguishable*, or equivalent. By explicitly modeling the external behavior of actors vis-à-vis the computing system, UML has the capability to specify when two systems are equivalent, i.e. when their behavior is indistinguishable.

Equivalent systems can be substituted for each other, allowing for component interchangeability. Providing the notion of equivalence goes a long way towards the formalization of interactive computing. By focusing on the behavior, UML abstracts away the implementation details. On the other hand, *object-oriented* (OO) programming languages specify the *implementation* of interactive systems, components, and objects; their *encapsulation* mechanism is central to this implementation. However, the specification stops at the system's boundaries, without a way to model its interactions with the environment. Thus, they do not have the capability to specify equivalence of computing systems they implement.

The interaction-based UML approach to system modeling fits well with the encapsulation-based OO approach to implementation. By marrying these approaches, the software engineering process can provide a complete solution to system design and implementation.

3. Three views of system models

Object models have many "small" objects with *sequential* interface and interaction protocols. That is to say, the input tokens (*events* and *messages*) are supplied via a single stream which serializes the order in

which the tokens are received by the computing entity. The entity must act on each input before consuming the next one. On the other hand, applications like distributed airline reservation systems have "heavy" components with multiple interfaces and concurrent interaction protocols.
The gap between static structure and dynamic behavior is greater for interactive computing entities such as objects than for algorithms, due to two distinct levels of execution dynamics: the external dynamics of operation execution is entirely separate from that of inner procedural algorithm execution. To account for this, UML provides several views of system modeling: a *static view* for describing static object structure, a *dynamic view* that describes interactions, and a *functional view* that describes transformation behavior of operations [Re, Ja1, Ru]:

Structural modeling	**static view**: describes relations among interactive components (*nouns*); static description of objects, operations, and relations among objects by *class diagrams, object diagram,* and *component diagrams;*
Behavioral modeling	**dynamic view**: describes interactions within the system (*inter-object dynamics*) with *use case*, *sequence, collaboration,* and *state-transition diagrams*;
	functional view: describes behavior of specific functions or methods (*intra-object dynamics*); modeling algorithmic transformation behaviors with *activity diagrams, state diagrams* and *narratives*.

Figure 2: The three views

These three views represent projections of the system onto different conceptual dimensions, similarly to database views. Together, they provide a robust modeling framework that can be coupled with a variety of object-oriented programming environments, as pointed out by Jacobson [Ja1]. They reflect the fact that *nouns* (agents, entities) provide a more direct and more expressive model of the real world than *verbs* (procedures, methods) and the further fact that nouns are modeled computationally by the patterns of actions (verbs) that they support. UML supports modeling of nouns and their actions along all three dimensions: the object model expresses relations among nouns, the dynamic model expresses patterns of interaction, and the functional model specifies the effect of individual actions.
The three-view approach to system design clearly indicates the role of algorithms as low-level transformation specifications of primitive elements of interaction patterns. Interactive computation can be broken up into algorithmic steps, but viewing it as nothing more than that would "miss the forest for the trees".
Though there is a resemblance among static, dynamic, and functional models and corresponding levels of modeling for algorithms (see Figure 2), it is only superficial. Consider state diagrams in UML. Like *flow diagrams*, they have a starting state and depict flows of control. But flow diagrams have actions as nodes and control paths as edges, while state diagrams have internal object states as nodes and observable input/output actions as edges. The *state transitions* in state diagrams represent a change in internal action state in response to an external event. State diagrams also resemble deterministic finite automata (DFAs). UML state transitions may involve input or output; thus they are *transducers*, not recognizers or generators. Also, both the number of different transition labels and the number of different states in state diagrams is not finitely bounded, as it is for DFAs.
Similarly, object interaction histories are very different from algorithm execution histories. Algorithm execution histories specify internal instruction sequences, while system interaction histories specify observable events in real or artificial worlds.

An interaction history, which is a trace of interactions during a single (finite) computation, can be viewed as a test case for the system, analogous to instruction execution histories of algorithm computations.

Just as no amount of testing can prove correctness of algorithms, interaction histories can only show the

existence of desirable behaviors and cannot prove absolute correctness.
General patterns of interaction depend both on the system and its environment. Because the behavior of actors in the environment is not completely known, these patterns are not finitely describable. The behavior of entities that interact with open environments that change unpredictably during the process of computation cannot be described by algorithms [We3]. This impossibility result is in spite of the finiteness of the UML model for the system whose behavior we are trying to describe.
The problem of driving from one's work to home provides an illustration. This problem can be solved by combining algorithmic knowledge (a high-level mental map) with interactive visual feedback of road conditions and topography. In a toy world without traffic or bad weather, interactive feedback could be replaced by an algorithmic specification of the problem. In this case, it is possible to provide enough information up-front so that a blindfolded person can know when to turn the steering wheel or to slow down without any feedback. However, in the real world where every pothole and pebble affects the car's trajectory in a chaotic fashion, the complexity of such an algorithmic specification is enormous. The presence of traffic and the effect of weather conditions make the algorithmic specification impossible.

Before UML, the non-algorithmic aspect of interactive system behavior forced the literature on design patterns [GHJV] to resort to informal verbal descriptions of problem and solution structure, by specifying reusable patterns of object and component interaction. Design patterns are behaviorally simple interaction patterns, but interaction patterns are too low level to capture user-level regularities (they are the machine language of design patterns). UML provides a higher level notation for describing design patterns.
The fact that interactive systems cannot be described algorithmically, in a formalism that is equivalent to Turing computability, is an *incompleteness* result. The incompleteness of interactive systems implies that proving correctness is not merely hard but impossible. We must be satisfied showing the existence of desirable behaviors through test cases (sequential interaction histories in the case of UML) and cannot hope to prove the nonexistence of all incorrect behaviors.

4. Modeling interaction in UML

4.1 Use cases: a system and its environment

One of the driving notions behind UML is *use cases*. Use cases encapsulate units of service in a computing entity: a system, a subsystem, or a class. They are descriptions of a category of externally required functionality embodied in transactions or interactions, involving *actors*. A *use case* is an abstraction representing the *input sources* of the system; usually, this means the users, though other sources of input are possible. Use cases restrict or constrain the user, known as *actor*, to a particular *role*, e.g. customer, vendor, line supervisor. The actor interacts with the system by exchanging *messages.* [OMG1]
A use case can be clearly distinguished from a subroutine call or a step in an algorithm. The step is chosen deterministically by the system design, whereas the use case instance is initiated by the external actor. To the system, the structure or state of the actor is unknown.

> "Since an actor is outside the entity [in a use case], its internal structure is not defined, but only its external view as seen from the entity." [OMG1]

As a result, the actor's decisions driving the course of the computation are seen by the system as arbitrary. To the system, the actor acts as a non-deterministic source of inputs.

> *The system must be designed without having complete knowledge of its environment*

Likewise, to the actor, the system is defined by its behavior rather than by its internal structure. The actor is not presumed to have any information about the implementation of the system. Systems that have the same behavior appear *equivalent*, or interchangeable, to users (see section 2).
The actor may dynamically change during the computation. For example, it might be influenced by the

system's outputs. Thus the system can in no way necessarily anticipate messages from the actor. In a real-world setting, most users will in fact generate system input that is affected by earlier system output. The semantics of the interaction sequence thus may include a bi-directional relation of causality between system input and system output. This is an interactive phenomenon, contradicting a basic premise of algorithmic computation, where input entirely precedes the processing and the output of a computation.

4.2 Collaboration diagrams: internal vs. external interaction

We now turn to *internal interactions* within a computing entity. Here the actors are *internal*; they are computing entities within the modeled system, components or objects. Unlike external actors, internal actors are under the control of the system; their behavior is specified by the system. Internal interaction is *symmetric* (mutual), where the entities serve as each other's actors. From the system's point of view, there is no inherent distinction between actors and "actees" for these interactions. This symmetry is reflected in UML *collaboration and sequence diagrams* that model internal interactions.

UML's support of *collaboration* is an important feature. A collaboration defines a set of *roles* and specifies the interactions of the entities that are playing those roles [OMG1]. Collaboration can be viewed as a *constraint* on behavior, for computing entities as well as human beings; for example, the need for collaboration in the workplace or family constrains our behavior [We2]. The simplest collaboration diagram may simply define a path between interacting objects to denote that instances of the classes exchange messages in the collaboration.

Sequence diagrams focus on time sequencing rather than role playing. They depict the time-ordering of the message passing between interacting entities over their lifetime, focusing on the control aspect. In either collaboration or sequence diagrams, there is no built-in duality of entity vs. environment. For each entity, the others form its environment, but the overall model takes a neutral stance. This is in contrast to the entity-centric *asymmetric* point of view taken in *use cases*, where the actors are not necessarily under the system's control (e.g., they can be human beings, or physical sensors).

Use cases can also be applied to modeling internal interactions; from a computing entity's point of view, interactions with internal agents are no different than with external ones. Use cases specify proper entity behavior in response to the actors, but they make no guarantees about the actors' behavior. Some behaviors such as collaboration cannot be modeled with use cases. It is not enough to hope that the actors will collaborate; the model must explicitly specify it. This can only be done when the model can assume that all parties to the interactions are under the system's control.

5. Event-driven computing

A notion supported by UML, inherited from the Common Object Resource Broker Architecture (CORBA), is that of an *event*. In a single-user GUI-driven system (such as one with the *model/view/controller* architecture), the system receives events from an *event stream* and handles each one in turn. In a system modeled by UML, events can also be internal to the system, triggering changes to components and objects. Events act as *input tokens* for the entities; when the same event affects multiple entities, it acts as input for each of them.

Events decouple control from statement execution to a greater extent than procedure calling; the analogy here is to *asynchronous* vs. *synchronous* computation. *Exceptions* in traditional programming languages are a restricted form of events that cause the normal flow of control to be modified when exceptional actions are required. Event models elevate the exception mechanism to be the primary control structure and generalize it so that occurrence of an event can cause multiple components to be notified of its occurrence.

Streams are distinct from strings or sequences due to their dynamic nature; each element in the stream is not available until the previous one is processed; this is known as a *lazy evaluation* mechanism. Lazy evaluation allows each input token to be produced interactively, after the previous token has been processed. As a result, an interactive input stream between an actor and the system is theoretically infinite: it can always be dynamically extended. For example, the user of a workstation may choose to work a little longer before logging off. This is analogous to interactive hardware devices (transducers), which generally do not enforce a finite restriction on the length of input: they stop only when the power is turned off. The fact that UML can model open-ended event loops means that it can model non-algorithmic behaviors. Algorithms, defined as plans for finite processes, cannot express computation over dynamically supplied infinite input streams; by definition, they have finite input and output. A stream of input or output tokens, defined as *stream* = (*token, stream*), is an element of a *non-well-founded set* defined by *coinductive*, rather than inductive, methods [BM, JR, Ru1, WG3].

6. Concurrency and multi-actor interaction

Once we accept the idea that interactive behaviors are richer than algorithms, we can distinguish two levels of interactive expressiveness, *sequential* vs. *multi-actor*. In a *sequential interactive entity* such as an abstract data type, there is a single input (event) stream, representing non-concurrent interaction with a single actor. A *multi-actor interactive entity* processes multiple concurrent interaction streams, each one representing interaction with a separate autonomous actor. The actors interact with the entity simultaneously and independently (autonomously), without necessarily any awareness of the presence of other actors. Moreover, the number of actors can change dynamically: actors can start and terminate their interaction with the entity without the awareness of other actors.

A multi-actor interactive entity is a composite entity, with multiple sub-entities. Some of its sub-entities interact with these external actors, while others only interact internally. Just as actors can come and go, so can these sub-entities; they can be born or die. Interaction streams, too, can be created, rerouted, or destroyed. As a result, the design of multi-actor interactive entities involves active management of interaction, at a level unseen in sequential interaction.

The need for active management of interaction, and for dynamic reconfiguration of interactive entities, makes multi-actor interaction harder to formalize and to model than sequential interaction. On the other hand, it allows the set of behaviors for multi-actor systems to be strictly richer than for sequential systems. Evidence for greater expressiveness of multi-agent interaction comes from transaction theory (where the class of *non-serializable transactions* is known to be richer than the *serializable transactions*), and from concurrency theory (where *true concurrency* is not believed to be reducible to *interleaving concurrency*) [Pr].

Multi-agent interaction precisely defines concurrent distributed systems: it is precisely the interactive aspect of concurrent and distributed computation that makes it more expressive, and more difficult to formalize [WG3]. When a modeling notation includes restrictions on concurrent behavior that force it to become serializable, its expressiveness is reduced. It is challenging to find clean notations for specifying concurrent behaviors that capture the full expressiveness of these behaviors. UML provides support for concurrency (multi-agent interaction), but it is not surprising that there are weaknesses with this support [MM], since the nature of multi-actor interaction is still poorly understood.

Work is under way to use wireless communication and the Internet to enable richer interaction among embedded sensors and controllers, such as those inside cars and appliances, so the streams of data flowing past each device can be harnessed to the benefit of other devices in the network. This is the theme of a recent issue of the *Communications of the ACM* [EGH]. Although this phenomenon has been labeled “beyond interaction” (where interaction is presumed to be associated with human-computer communication [Te]), communication among embedded devices fits precisely into out notion of interaction, and can in principle be modeled by a language like UML.

7. A paradigm shift

The need for new notations for modeling software and information systems arose from the ever-increasing level of complexity of today’s systems. The rise of the world-wide web compounded our expectations of software systems, providing them with a virtual environment far more complex than heretofore. As the level of complexity increased, so did the level of abstraction. Structured programming’s emphasis on the sequence-branch-loop trio of control structures gave rise to structured flow-charting. Top-down design gave rise to module-hierarchy diagramming. Object-based programming brought forth class diagrams. The entity-relationship diagram came in part out of the linking of database relations. UML developed as these approaches showed their limits at modeling distributed concurrent systems that offer replication and load balancing while trying to ensure security and fault tolerance [OMG2]. What is modeled today are systems that offer tangible user services, not just transform data, and where interaction is pervasive.

The interactive, indeed multi-interactive, aspects of today's computing were highlighted by Cris Kobryn, one of those involved in UML's development. He noted that UML emerged under the impetus of the urgent need to add a superstructure for interprocess communication and distributed system services to the infrastructure supplied by the widely-used Common Object Resource Broker Architecture (CORBA). CORBA's IDL could not specify use cases, collaborations, state transitions, and work flows that can be found in a complex system, such as an interactive software system or a business organization [Ko].

The paradigm shift from algorithms to interaction is intimately related to the evolution of practical computing. The technology shift from batch-oriented, procedural mainframe-based technology of the 1960s to object-oriented, GUI-driven, distributed workstation-based technology of today and to mobile, embedded, pervasive computing agents of tomorrow is fundamentally a shift from algorithms to interaction.

yesterday:	batch-oriented, procedural mainframe-based technology
today:	object-oriented, GUI-driven, distributed workstation-based technology
tomorrow:	mobile, embedded, pervasive, adaptable computing agents

The implicit contract between *computing entities* (such as systems or objects) and their clients (users or other entities) has changed from a sales-like contract to transform inputs to outputs to a marriage-like contract to provides continuing services over time [We2]. Computing agents express persistent services over time (marriage) which cannot be modeled by time-independent input-output actions (sales). The folk wisdom that marriage contracts transcend sales contracts translates to the formal result that interactive systems cannot be modeled by algorithms.

Expressiveness, or power, of finite computing agents is defined in terms of the agent's ability to make observational distinctions about its environment. This notion of expressiveness applies equally to people and to computers. People who see are more expressive than otherwise-identical blind people because they can make visual distinctions, while telescopes and microscopes increase expressiveness by allowing people to make still finer distinctions.

The ability of interactive agents to make finer distinctions than algorithmic processes is illustrated by *question answering*. Interactive questioning forces the *answerer* to commit to earlier answers before seeing later questions; it also allows the *questioner* to base later questions on the answers to earlier ones. For example, if an investigator (questioner) is interrogating a suspect (answerer), the investigator can learn more by interactive questioning with follow-up questions than by asking the suspect to fill in a questionnaire, which has the status of a multi-part single question. As a result, it is possible to imagine cases where suspects can fill out any questionnaire without implicating themselves, but where an interactive interrogation can exploit weaknesses in their story to establish their guilt.

The idea that interaction is not expressible by or reducible to algorithms was first proposed by Wegner in 1992 at the closing conference of the fifth-generation computing project in the context of logic programming [We4]. Reactiveness of logic programs, realized by commitment to a course of action, analogous to the suspect's commitment to earlier answers, was shown to be incompatible with logical completeness, realized by backtracking. The fifth-generation project's failure of achieving its maximal objective of reducing computation to first-order logic can be attributed to theoretical impossibility rather than to lack of cleverness of researchers.

Brooks' belief that there is no silver bullet for system specification [Br] can be restated in terms of the impossibility of algorithmic specification of interactive systems. In fact, a proof of irreducibility of interactive specifications to algorithms can actually be interpreted as a proof of the nonexistence of silver bullets. The irreducibility of interaction to algorithms also explains Rentsch's comment that "Everyone is in favor of [object-oriented programming] but no one knows just what it is." [Ren] If "knowing what it is" means reducing object-oriented programming to algorithms then the reduction is bound to fail. But if we enlarge the class of things that "count" as explanations of object-oriented programming to include interactive models such as specified with UML, then we can succeed.

8. Interaction machines: a model of interactive computation

Interactive computational models provide a formal framework for interactive computation, just as Turing Machines (TMs) model algorithmic computation. TMs have a tape, which initially contains a finite input string, and a state transition mechanism that reads a character from the tape, performs a state transition, optionally writes a character on the tape, and repositions the reading head. When the machine halts (goes into a "halting" state), the contents of the tape represents its output.
Sequential interaction machines (SIMs) model objects and software components. Persistent Turing Machines, which are a canonical version of a SIM, extend the Turing machine model by treating each Turing computation as a "macrostep" in an interactive computational process over a dynamically generated stream of input strings. Another essential extension is a persistent internal worktape, so the initial TM configuration (which includes the contents of this tape) changes for every macrostep:

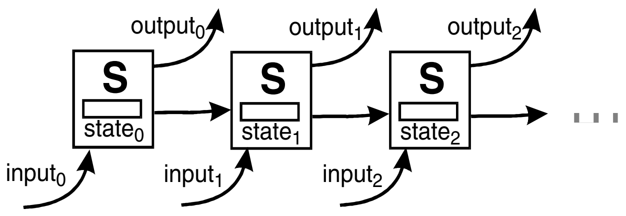

Figure 3. Sequential interaction

SIMS model *sequential interactive entities* such as abstract data types, where there is a single input (event) stream, representing non-concurrent interaction with a single actor. A *multi-actor interactive entity*, whether it is a system or a component, processes multiple concurrent interaction streams, each one representing interaction with a separate autonomous actor. Multi-actor interaction machines (MIMs) are more expressive than SIMs. Multi-actor interaction is more expressive than sequential interaction, explaining why true concurrent distributed computation cannot always be simulated by interleaving computations.
A computing system is said to be *open* if the course of its computation can be affected by external events, and *closed* otherwise. The distinction between closed and open systems is precisely that between algorithmic and interactive computing. Turing machines are closed systems because their rules of engagement require all inputs to be supplied on an input tape before the beginning of the computation. Algorithms are closed because they shut out the world during computation. Interactive computing is open in allowing external agents to observe and influence computation.
Any open system can be closed by constraining its rules of engagement to be independent of external effects. If two interactive (open) subsystems only interact with each other, then the system or subsystem which results from their composition is closed. This closed system may be algorithmic, if both subsystems are finitely specifiable and effective. Two-component systems where each acts as a constraint on the other arise in control theory: one component is the system being controlled while the other is a controller.
Models of interaction provide a unifying conceptual framework for the description and analysis of object and component-based architectures. They are a tool for exploring software design models like UML, models of distributed object and interoperability like CORBA, design patterns, coordination languages like Linda and Gamma, and AI models of planning and control.
The irreducibility of interactive systems to algorithms was noticed by [MP] for reactive systems, by [Mi1] in the context of process models, and by [We3] in the context of interaction machines. This approach in [We3] differs from related work by focusing on models of interaction and notions of expressiveness that are language-independent as well as domain-independent. Subsequent work [WG2, WG3, Go], has led to the development of persistent Turing machines (PTMs) as a canonical model of sequential computation, of an expressiveness hierarchy for sequential computation, and of the result that multi-stream interaction machines (MIMs) are more expressive than sequential interaction machines (SIMs).
Turing's proof that algorithms, Turing Machines (TMs), and the lambda calculus are equally expressive [Tu1], suggested that the question of expressiveness of computing had been settled once and for all

(*Church's thesis*), and channeled research to questions of design, performance, and complexity for a fixed notion of computability. But as a result of the irreducibility of interaction to algorithms, we know that software systems cannot be modeled by TMs, and their behavior cannot inherently be expressed by first-order logic. New classes of models are needed to express the technology of interaction, and UML fills that need.

9. Conclusion

UML provides a solution to the challenge posed by Milner in 1975, that functions cannot express meanings of processes [Mi2, We3]. UML models nonalgorithmic computation of interactive systems that operate in open noncomputable environments.

Interactive computational models are common nowadays in software engineering, operating systems, and artificial intelligence. The evolution in AI from logic and search to agent-oriented models is not merely a tactical change but is a strategic paradigm shift from algorithms to more expressive interactive models that fundamentally increases expressive power. The reasoning/interaction dichotomy is precisely that between "good old-fashioned" AI (GOFAI) and "modern" agent-oriented AI. This paradigm shift is evident not only in research (such as the StarLogo system [St]) but also in textbooks that systematically reformulate AI in terms of intelligent agents [RN]. UML might prove useful in the AI area, for intelligent agent design.

With the systems modeled by UML, we see a re-introduction of side effects, once banished ("Goto considered harmful"). Side effects were shunned because they produced nonformalizable behavior. Formalizability, or the ability to formally prove various properties of the system, is an attractive feature of algorithmic computation, and it was felt that any computation with side effects could (and should) be transformed to an equivalent one without. With interactive computation, side effects are inevitable; e.g., invoking a method changes the state of the method's owner. Rather than view interactive computation as "undesirable" due to its nonformalizability, we must take it as a challenge to find new, non-algorithmic, tools and methods for formalizing computation, ones where side effects have a place.

It's interesting to note that Turing's seminal paper [Tu1] was not intended to establish TMs as a comprehensive model for computing but on the contrary to show undecidability and other limitations of TMs. Turing actually mentioned irreducibility to TMs of interactive choice machines (*c-machines*) as a presumably well-known fact [Tu1]. However, he did not proceed any further with a theory for such a machine. This early reference to interactive computation has basically gone ignored by the community that Turing founded.

It is tempting to assume that interactive models of computation such as c-machines have not been studied due to lack of their intellectual merit. However, there is a different explanation: that the theory community has lacked the proper conceptual tools to make progress on these ideas. The claim that interactive computation is more expressive than algorithms opens up a research area that had been considered closed. Much work needs to be done on both the foundations and applications of interactive computing, to provide a systematic foundation for interactive software technology [We1]. UML provides a large step forward with motivating and enabling this work.

10. References

[Ba] H. P. Barendregt, Functional Programming and Lambda Calculus, *In* Jan Van Leeuwen, ed., *Handbook of Theoretical Computer Science*, Vol. B. MIT Press, 1990, pp. 322-360.

[BM] Jon Barwise and Lawrence Moss, *Vicious Circles*, CSLI Lecture Notes #60. CSLI Publications, 1996.

[Br] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering.* AddisonWesley, 1995.

[Da] Martin Davis, ed., *The Undecidable*. Raven, 1965.

[EGH] Deborah Estrin, Ramesh Govindan, and John Heidemann, Guest Editors, Embedding the Internet (special multi-article section), *Communications of the ACM* 43:5 (May 2000), pp. 39ff.

[FELR] R. France, A. Evans, L. Lano, and B. Rumpe, The UML as a formal modeling notation, *Computer Standard and Interfaces* 19, pp. 325-334, 1998.

[Fo] Fowler, Martin, *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

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

[Go] Dina Goldin, Persistent Turing Machines as a Model of Interactive Computation, FoIKS'00, Cottbus, Germany, Feb. 2000.

[GK] Dina Goldin and David Keil, Minimal Sequential Interaction Machines, work in progress, January 2000 (*www.cs.umb.edu/~dqg/papers/minimal.ps*).

[HU] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Ja1] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Revised 4th printing. Addison-Wesley, 1993.

[Ja2] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology.* Addison-Wesley, 1995.

[JR] Bart Jacobs and Jan Rutten, A Tutorial on (Co) Algebras and (Co) Induction, *EATCS Bulletin 62*. 1997.

[Ko] Kobryn, Cris, UML 2001: A Standardization Odyssey, *Communications of the ACM* 42:10 (Oct. 1999), pp. 29-37.

[LMWF] N. Lynch, M. Merritt, W. Weihl, and A. Fekrete, *Atomic Transactions.* Morgan Kaufmann, 1994.

[MM] Michael J. McLaughlin and Alan Moore, Real-Time Extensions to UML, *Dr. Dobb's Journal*, December, 1998.

[Mi1] Robin Milner, Operational and Algebraic Semantics of Concurrent Processes. *In* Van Leeuwen, J., Ed., *Handbook of Theoretical Computer Science.* Elsevier, 1990.

[Mi2] Robin Milner, Processes: A Mathematical Model of Computing Agents. In *Logic Colloquium '73*. North-Holland, 1975.

[Mo] Edward F. Moore, Gedanken-Experiments on Sequential Machines. In C. E. Shannon and J. McCarthy, Eds., *Automata Studies*. Princeton University Press, 1956.

[MP] Zohar Manna and Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[OMG1] Object Management Group, *Unified Modeling Language Specification*. Ver. 1.3, March 2000.

[OMG2] Object Management Group, What is OMG-UML and Why Is It Important?, *omg.org/news/pr97/umlprimer.html*.

[Pr] Vaughan Pratt, Chu Spaces and their Interpretation as Concurrent Objects, in *Computer Science Today: Recent Trends and Developments*, Ed. Jan van Leeuwen, LNCS #1000, 1995.

[Re] Paul Reed, The Unified Modeling Language Takes Shape, *DBMS Magazine*, July 1998.

[Ren] T. Rentsch, Object-Oriented Programming, *SIGPLAN Notices*, 17:12 (September 1982), p. 51.

[RN] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach.* Addison-Wesley, 1994.

[Ru] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1990.

[Ru1] Jan Rutten, A Tutorial on Coalgebras and Coinduction, *EATCS Bulletin 62.* 1997.

[Si] Herbert Simon, *The Sciences of the Artificial,*2nd Edition, MIT Press, 1970.

[St] Starlogo site at MIT Media Lab (*http://starlogo.www.media.mit.edu/people/starlogo*).

[SW] Geri Schneider and Jason Winters, *Applying Use Cases: A Practical Guide.* Addison-Wesley, 1998.

[Te] David Tennenhouse, Proactive Computing, *Communications of the ACM* 43:5 (May 2000), pp. 43-50.

[Tu1] Alan Turing, On Computable Numbers with an Application to the Entscheidungsproblem, *Proc. London Math Soc*., 2:42 (1936), pp. 230-265.

[TW] Putnam Texel and Charles B. Williams, *Use Cases Combined with Booch/OMT/UML: Processes and Products,* Prentice Hall PTR, 1997.

[We1] Peter Wegner, Draft of ECOOP '99 Banquet Speech, *www.cs.brown.edu/people/pw.*

[We2] Wegner, Peter, Why Interaction is More Powerful than Algorithms, *Communications of the ACM* 40:5 (1997).

[We3] Peter Wegner, Interactive Foundations of Computing, *Theoretical Computer Science* 192 (1998), pp. 315-351.

[We4] Peter Wegner, Interactive Software Technology, *CRC Handbook of Computer Science and Engineering*. 1996.

[WG1] Peter Wegner and Dina Goldin, Interaction as a Framework for Modeling, *Lecture Notes in Computer Science* #1565, 1999.

[WG2] Peter Wegner and Dina Goldin, Interaction, Computability, and Church's Thesis, 5/99, accepted for publication in *British Computer Journal*

[WG3] Peter Wegner and Dina Goldin, Coinductive Models of Finite Computing Agents, *Electronic Notes in Theoretical Computer Science* 19 (March 1999)

[ZM] Stanley Zdonik and David Maier, *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1990.