Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 141 (2005) 181–198

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The Role of Agent Interaction in Models of Computing: Panelist Reviews

Peter Wegner

Brown University, USA

Farhad Arbab

CWI & Leiden University, Netherlands

Dina Goldin

Univ. of Connecticut & Brown University, USA

Peter McBurney

Univ. of Liverpool, UK

Michael Luck

Univ. of Southampton, UK

Dave Robertson

Univ. of Edinburgh, UK

1 Introduction

I am pleased that panelists agreed to contribute short reviews relating to the panel on "The Role of Agent Interaction in Models of Computation", at the Workshop on Foundations of Interactive Computation", held in Edinburgh in April 2005. The panelists were asked, prior to the panel, to address the following questions:

doi:10.1016/j.entcs.2005.05.022

- What are the challenges in formalizing models of interactive computation?
- How does interaction impact the notion of solving computational problems?
- How do interactive systems relate to open multi-agent systems?
- How can practice and experience on multiagent systems influence the theory of computation?
- How do models of interaction contribute to the study of AI, networking, and other applications?
- How does interaction influence interdisciplinary applications?
- Where do concurrency theory and agent coordination/cooperation fit in?

The presentations at the panel and the questions and comments of the audience greatly contributed to our understanding of the role of interaction in better modeling the computing discipline, which was a primary goal of the overall workshop. The authors and titles of the panelist reviews included in this article are as follows:

- *Farhad Arbab*, Impact of Interaction on Constructive Models.
- *Dina Goldin*, The Turing Thesis Myth.
- *Peter McBurney*, Agents and Interaction (co-authored with Michael Luck).
- *David Robertson*, Interactive Cooperation.

We hope that these panel reviews will contribute to the readers understanding that interaction can contribute to the evolution of future models of computation.

Peter Wegner, Panel Chair

2 Impact of Interaction on Constructive Models of Computation

The first question posed to the members of this panel asks: "how do interactive models change our notion of a computational problem?" The second question inquires "how do models of interaction contribute to the study of AI, networking, and other applications?" Software Engineering deals with construction of complex software systems. I address these two questions in the context of Software Engineering: how models of interaction can contribute to the study of complex systems, and how interactive models can change our notion of computational problems that we construct those systems for.

Various models of computation exist to serve different purposes. Turing Machines (TMs), for instance, capture the essence of algorithmic computing as a sequence of mechanical operations that, if terminates, transforms its given input into an output. TMs were devised to explore the expressiveness of this notion of computing, and its limits. TMs are not (meant to be) useful for the actual construction of computing systems, hardware or software. Examples of constructive models of computation include the so-called von Neumann model, functional programming, logic programming, imperative programming, and object oriented programming.

Every system that performs a complex computation is not necessarily complex. Quite simple systems can contain complex algorithms. The challenge in engineering of complex computing systems has little to do with complex algorithms; it involves coping with the complex behavior that arises out of composing even a non-trivial number of (even simple) constituent parts into a coherent functioning whole. The complexity of a system arises out of the multitudes of combinations of ways in which the functioning of its constituent parts can mutually affect and interfere with one another. Various forms of concurrency often arise in the study and construction of such systems.

Concurrent TMs do not add expressiveness over what a single universal TM offers: whatever algorithmic computation a set of concurrent TMs can perform, can also be performed on a single TM. In spite of this expressive equivalence, models of computation that have proven effective for construction of sequential programs are notoriously inadequate for construction of concurrent systems. Calculi such as CSP [14], CCS [21], the π-calculus [22,25], process algebras [6,7,29], and the actor model [1] are among the various models of computation specifically aimed at the complexities that arise in the construction of concurrent systems.

Wegner's proposal of Interaction Machines [30,32] and the claim that they model more than the algorithmic notion of computing captured by TMs have drawn considerable attention on interaction as a new paradigm in computing. However, Interaction Machines, as well as most subsequent work on interaction, e.g., by Goldin, et al. [11], and van Leeuwen and Wiedermann [27,28], focus on expressiveness issues. As such, one may regard them as the "TM level" work for the new paradigm of interaction. Wegner and Goldin have proposed interaction as a framework for modeling of complex systems [33].

The real world contains unpredictability and non-computable functions. This places interaction at the center of the design of computing systems that must interface with the real world. But an interaction-centric perspective serves as an effective framework that simplifies the study, specification, and construction of all kinds of complex systems. In systems that deal with the subsets of the real world that exclude unpredictability and non-computable functions, or in closed computing subsystems that do not interact with the real world at all, everything is, at least in principle, knowable and/or computable. Good engineering practice suggests that it makes perfect sense to pretend that even in these situations, the context in which each piece of a computing system must function is unknowable and may be subject to unpredictable change. Pretending that the environment is unknowable is tantamount to assuming that every component/subsystem interacts with the real world. It leads to a "defensive" design perspective that yields more robust components/subsystems which contain fewer implicit assumptions about their environment and can therefore be reused in a wider variety of contexts. But, what sorts of models can one use to actually construct systems, exploiting interaction as a computing paradigm?

Of course, interactive systems consist of concurrent parts, and it is only natural that people resort to the wealth of knowledge, models, languages, and tools that have emerged over decades of experience with concurrency, in construction of interactive computing systems. However, one should not misconstrue the lack of better tools and familiarity of existing ones as evidence for their adequacy. The fact that we currently apply languages and tools based on various concurrent object oriented models, the actor model, and various process algebras, etc., simply means that they comprise the best in our available arsenal, but it does not mean that they necessarily embody the most appropriate models for tackling interaction in practice. Even for no reason other than to properly evaluate the adequacy of our existing tools, we need to cast them aside, if only temporarily, and address, with a fresh mind, a few fundamental questions regarding the hallmark properties of *constructive models of interaction*. After all, if interaction identifies a distinctive shift within (or out of) concurrency, of a magnitude deserving recognition as a new paradigm, then it surely must have at least some non-trivial practical implications on what characterizes a suitable model or tool for construction of systems exploiting that distinction.

The most striking hallmark of *interaction* is that it is a phenomenon that involves two or more actors. This is in contrast to *action*, which is what a single actor manifests. A model of interaction must allow us to directly specify, represent, construct, compose, decompose, analyze, and reason about that which transpires among two or more engaged actors, without the necessity to be specific about their actions. Only then can we have an explicit model of an interaction (protocol) and directly study its properties independently of the details of its engaged actors.

The actor model and various concurrent object oriented models tackle concurrency by focusing on construction of things that interact, not on interaction. The actors (or agents, objects, etc.) in these models constitute the primary concern. By explicitly performing actions such as sending or receiving a message or invoking a remote method, these actors collectively fulfill their tacit obligations under an implicit interaction protocol. These models have no means to express an interaction protocol explicitly as a tangible piece of specification or code. Instead, an interaction can be derived only by considering and inferring the possible combinations of the actions of all involved actors together. Significant properties of interactions in these models, e.g., rendezvous, synchronous, or various forms of asynchronous communication, are often fixed by the model itself and cannot be altered. Where such a model allows a choice, it is only the interacting parties that can choose from the alternatives. For instance, the target of a message is determined by its sender; what the receiver must do is indicated by the sender (i.e., which method is invoked), but determined by the receiver; if the model allows a choice on synchronization, then the actions of the interacting parties determine which form is used; etc. All this means that given a set of actors, their interaction is fixed; the same actors cannot be composed in a different interaction; an interaction protocol is neither tangible nor explicit; it cannot be specified, studied, or reused independently of its actors; and the only way to change an interaction is by modifying the actions of its involved actors.

Making interaction explicit in construction of a computing system allows building a system by choosing (1) an explicit interaction protocol, (2) a compatible set of actors, and (3) composing them. Models based on process algebras are (somewhat) better suited for this purpose. In the π-calculus, for instance, communication is not targeted. This allows a third-party process to pick and compose two other processes in a variety of ways, thus influencing the protocol by which they interact. Nevertheless, process algebras are models for constructing *processes* out of a set of atomic ones, which represent primitive actions. They offer operators for composing (atomic) processes into more complex ones. Interaction and communication protocols ensue only as ancillary consequences of the unfolding of the collective behavior of the processes involved in a concurrent system and have no explicit constructs to directly express them. The compositionality offered by process algebras convolutes composed interaction protocols: to learn how a process r that is a parallel composition of processes p and q interacts with its environment, one must unravel the actions of p and q and consider all of their possible combinations.

Constructive models in the paradigm of interactive computing must offer (1) primitive interactions; and (2) rules of composition for combining (primitive) interactions into more complex interactions, without the need to specify (the actions of) the actors involved. The necessity to explicitly specify an interaction (i.e., what) independently of the actions that its engaged actors perform to manifest it (i.e., how) confers a relational constraint programming style upon such a model: an interaction is an explicit relation that holds among a set of actors and constrains every individual to coordinate their collective behavior, shaping a coherent whole. Such explicitly specified constraints can be composed together in various ways to yield more complex constraints (i.e., interaction protocols), without the need to specify the action sequences of any actors.

Reo [2,5] serves as a good example of a constructive model of interaction. It offers a channel-based exogenous coordination model wherein complex coordinators, called *connectors* are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. Connectors orchestrate the behavior of their engaged actors. Each connector in Reo imposes a specific coordination pattern on its engaged actors, who without knowing anything about their interaction, merely perform I/O operations through that connector.

In fact, every channel in Reo specifies a relational constraint that must hold between the I/O actions performed on its two ends, without saying anything about those actions or who performs them. These constraints specify the relative timing (i.e., synchrony/asynchrony) of (the success of) the I/O actions, and the desired data dependencies between them (e.g., buffering, ordering, selection, conversion, filtering, loss, and/or expiration of data). Reo's compositional operators indeed compose such relations to produce the more complex constraints that constitute the behavior of their resulting connectors. As an explicit, tangible piece of specification or program code, the same connector can be employed to engage entirely different sets of actors to yield entirely different systems. Perhaps more interestingly, the same set of actors can be composed together with different connectors, producing systems with very different emergent behavior. Abstract Behavior Types [3] and Constraint Automata [4] show how Reo's notion of behavior (of channels, connectors, and actors) can be formalized to reason about connectors and their composition.

Whereas process algebras explicitly compose and construct processes making the interaction relations amongst them implicit, Reo explicitly composes and constructs interaction relations and makes processes that engage in those relations implicit. Reo's liberal notion of channels and its fundamental notion of channel/connector composition allow, among other things, explicit construction of connectors that specify interaction protocols involving an expressive mix of synchrony and asynchrony.

Our tools not only influence how we solve problems, they also change our very notion of those problems. A constructive model of computing that uses interaction as a first-class primitive building block espouses the perspective of discovering the inter-dependencies among the principal actors in a computing problem, and directly specifying the relations that constrain their communication. System construction then becomes composing an explicit skeleton of interaction with component actors whose communication this skeleton coordinates.

3 The Turing Thesis Myth

The fields of the theory of computation and concurrency theory have historically had different concerns. The theory of computation views computation as a *closed-box* transformation of inputs to outputs, completely captured by Turing machines (TMs). By contrast, concurrency theory focuses on the *communication* aspect of computing systems, which is not captured by Turing machines – referring both to the communication between computing components in a system, and the communication between the computing system and its environment. As a result of this division of labor, there has been little in common between these fields.

According to the interactive view of computation, communication (input/output) happens *during* the computation, not before or after it. This approach, distinct from either concurrency theory or the theory of computation, represents a paradigm shift that changes our understanding of what computation is and how it is modeled [31]. Interaction machines extend Turing machines with interaction to capture the behavior of concurrent systems, promising to bridge these two fields.

The idea that Turing machines do not capture all computation, and that interaction machines are more expressive, seems to fly in the face of accepted dogma, hindering its acceptance within the theory community. In particular, the Church-Turing thesis is commonly interpreted to imply that Turing machines model all computation. It is a myth that this interpretation of the thesis is equivalent to the original [12]. In fact, the original Church-Turing thesis only refers to the computation of *functions*, and specifically excludes interactive computation.

It is time to recognize that today's computing applications, such as web services, intelligent agents, operating systems, and graphical user interfaces, cannot be modeled by Turing machines; alternative models are needed. Only by facing the fact that the applicability of the Church-Turing Thesis is limited to functions, whereas the general notion of computation is not, can we begin to properly investigate these alternative models.

One such model is *Persistent Turing Machines* (PTMs), originally formalized in [11]. PTMs capture *sequential interaction*, which is a limited form of concurrency; they allow us to formulate the *Sequential Interaction Thesis*, going beyond the expressiveness of Turing machines and of the Church-Turing thesis.

3.1 The interactive paradigm

The question "*what do operating systems compute?*" has been a conundrum for the theoretical community, since they never terminate, and therefore never formally produce an output. Yet it is clear that they *do* compute, and that their computation is both useful and important to capture formally. The computation performed by operating systems is *interactive*, where input and output happen *during* the computation, not before or after it.

We suggest that computation is to be viewed as an ongoing transformation of inputs to outputs – e.g., operating systems, or control systems. This interactive approach, distinct from either concurrency theory or the theory of computation, represents a paradigm change to our understanding of what is computation, and how it should be modeled. This conceptualization of computation allows, for example, the *entanglement* of inputs and outputs; later inputs to the computation may depend on earlier outputs. Such entanglement is impossible in with the traditional formalization of computation, where all inputs precede computation, and all outputs follow it.

The example of driving home from work [10] represents an empirical proof of the claim that interactive computation is more expressive than function-based computation, i.e. it can solve a greater range of problems. While the Church-Turing Thesis remains true, the mathematical worldview, defining all computational problems as closed-box transformations of predefined input to finite output, no longer reflects the nature of computational problems. Driving home from work, queuing jobs within an operating system, or controlling factory equipment, are all legitimate problems on par with finding common factors or choosing the next move on a given chess board.

3.2 Algorithm as a culprit

The current prevalent misunderstanding of the nature of computation, as expressed by the Church-Turing thesis, can be blamed on the 1960's decision by theorists and educators to place algorithms at the center of the new discipline of computer science (CS). The central role of algorithms in CS was prescribed by the ACM curriculum [8], and clearly reflected in early undergraduate textbooks. However, various textbooks chose to define this term differently.

While some textbooks such as [16] were careful to explicitly restrict algorithms to those that compute functions, and are therefore TM-equivalent, most theory books left the restriction unstated (though implied). Yet other early textbooks, usually those in non-theoretic fields, such as [23], explicitly broadened the notion of algorithms to include problems beyond those that can be solved by TMs. Two of their examples, that can supposedly be solved by an algorithm, are making potato vodka and filling a ditch with sand; driving home from work would fit right in, too.

A recent ACM SIGACT Newsletter acknowledges that of all undergraduate CS subjects, theoretical computer science has changed the least over the decades [26]. While the practical computer scientists have long since followed the lead of [23] and broadened the concept of algorithms beyond the computation of functions[1], theoretical computer science has retained the mathematical worldview that frames computation as function-based, and delimits our notion of a computational problem accordingly. This is true at least at the undergraduate level, despite advanced complexity theoretic work that ventures outside this worldview, such as on-line and distributed algorithms, Arthur-Merlin games, and interactive proofs.

The result of such inconsistent use of the term "algorithm" is a dichotomy, where the computer science community thinks of algorithms as synonymous with the general notion of computation ("what computers do") yet at the same time as being equivalent to Turing machines. This dichotomy expresses iteself in the incorrect interpretation of the Church-Turing thesis that is commonplace in the computing literature:

"A TM can do anything that a computer can do."

We refer to the common belief in the equivalence of this interpretation to the original thesis as the *Turing Thesis myth.*

3.3 Time for new models

The history of extending TMs is at least as old as the theory of computation. All TM extensions that can be found in theory textbooks, such as increasing the number of tapes or changing the alphabet, are algorithmic. In the case of algorithmic extensions, the Church-Turing thesis applies, and it can be taken for granted that the new model is equivalent to the original. However, as a result of the Turing Thesis myth, it is common to assume the equivalence of *any* TM extension to the original, and we no longer expect formal proofs of this.

To capture the contemporary interactive use of computers, the more recent TM extensions have tended to be non-algorithmic, with computation that

[1] Sometimes this was accomplished by revising the term "function" itself, as in imperative languages, where functions may have side effects, etc.

spans multiple inputs and outputs to the underlying TM. Nowadays we consider non-terminating interactive computations of Turing machines, persistent Turing machines, nets of Turing machines, Turing machines with evolvable architecture, etc., exactly for reasons of capturing "all computers", or "all computations". Indeed, the recent proliferation of such models can be viewed as a paradigm shift in the field of models of computation.

Out of habit, researchers have continued to assume that these extensions are equivalent to the original TM. But in the case of such non-algorithmic extensions, Turing's thesis does not apply, and equivalence can no longer be taken for granted. Indeed, it no longer holds, as discussed next.

3.4 Modeling interactive computation

Wegner [31,32] has conjectured that interactive models of computation are more expressive than "algorithmic" ones such as Turing machines. It would therefore be interesting to see what minimal extensions are necessary to Turing machines to capture the salient aspects of interactive computing. Motivated by these goals, [11] has investigated a new way of interpreting Turing-machine computation, one that is both interactive and persistent – *persistent Turing machines* (PTMs).

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 computes *concurrently* with its *environment*, both acting as consumers of each other's outputs and producers of each other's inputs.

In addition to having *dynamic stream semantics*, PTM computations are *persistent* 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. The notions of interaction and persistence in PTMs are formalized in terms of the *persistent stream language* (PSL) of a PTM. Given a PTM, its persistent stream language is 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.

The notion of a *universal PTM* is also defined in [11]. Similarly to a *universal Turing machine*, a universal PTM can simulate the behavior of an arbitrary PTM. The class of *sequential interactive computations* is also introduced:

Sequential Interactive Computation: A sequential interactive computation continuously interacts with its environment by alternately accepting an input string and computing a corresponding output string. Each output-string computation may be both nondeterministic and history-dependent, with the resultant output string depending not only on the current input string, but also on all previous input strings.

Examples of sequential interaction include sequential JAVA objects, static C routines, single-user databases, network protocols, and our original example of driving home from work.

A sequential interactive analogue to the Turing Thesis is provided:

Sequential Interaction Thesis: Any sequential interactive computation can be performed by a persistent Turing machine.

This hypothesis, when combined with other results 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 – proving Wegner's conjecture.

It has been also conjectured [32] that *multi-agent* interaction is more expressive than sequential, or *single-agent* interaction. These conjectures remain to be proven.

3.5 Conclusion

Hoare, Milner and others have long realized that TMs do not model all of computation [34]. However, when their theory of concurrent computation was first developed in the late '70s, it was premature to openly challenge TMs as a complete model of computation. Concurrency theory positions *interaction* as orthogonal to *computation*, rather than a part of it. By separating interaction from computation, the question whether the models for CCS and the π-calculus went beyond Turing machines and algorithms was avoided.

Researchers in other areas of theoretical computer science have also found need for interactive models of computation, such as Input/Output automata for distributed algorithms [20] and Interactive TMs for interactive proofs [13]. However, the issue of the expressiveness of interactive models vis-a-vis TMs was not raised until the mid-1990's, when the model of interaction machines as a more expressive extension of TMs was first proposed by one of the authors [31].

While not part of CS Theory, the field of AI has perhaps gone the furthest in explicitly recognizing the expressiveness gains of moving beyond algorithms. In the early 1990's, Rodney Brooks convincingly argued against the algorithmic approach of "good old-fashioned AI", positioning interaction is a prerequisite for intelligent system behavior [9]. This argument has been adopted by the mainstream AI community, whose leading textbooks recognize that interactive *agents* are a better model of intelligent behaviors than simple input/output functions [24].

In the last three decades, computing technology has shifted from mainframes and microstations to networked embedded and mobile devices, with the corresponding shift in applications from number crunching and data processing to the Internet and ubiquitous computing. We believe it is no longer premature to encompass interaction as part of computation. A paradigm shift is necessary in our notion of computational problem solving so it can better model the services provided by today's computing technology.

4 Agents and Interaction

As the computing landscape moves from a focus on the individual standalone computer system to a situation in which the real power of computers is realized through distributed, open and dynamic systems, we are faced with new technological challenges and new opportunities. In this section, I focus on interactive systems in dynamic and open environments.

The characteristics of dynamic and open environments in which, for example, heterogeneous systems must interact, span organizational boundaries. These systems must operate effectively within rapidly-changing circumstances and with dramatically increasing quantities of available information, suggesting that a revision of traditional computing models and paradigms is required. In particular, the need for some degree of *autonomy*, to enable intelligent components to respond dynamically to changing circumstances while trying to achieve over-arching objectives, is seen by many as fundamental.

While this notion is not intended to suggest an absence of control, some application contexts offer no alternative to autonomous software. In practical developments, Web Services, for example, now offer fundamentally new ways of doing business through a set of standardized tools, and support a service-oriented view of distinct and independent software components interacting to provide valued functionality. In the context of such developments, agent technologies have become the primary weapons in the arsenal aimed at addressing the emergent problems, and managing the inherent complexity.

Agent-based systems are one of the most vibrant and important areas of research and development to have emerged in information technology in the 1990s. Put at its simplest, an agent is a computer system that is capable of flexible autonomous action in dynamic, unpredictable, typically multi-agent domains. Many observers believe that agents represent the most important new paradigm for software development since object-orientation. The concept of an agent has found currency in a diverse range of sub-disciplines of information technology, including computer networks, software engineering, object-oriented programming, artificial intelligence, human-computer interaction, distributed and concurrent systems, mobile systems, telematics, computer-supported cooperative work, control systems, mining, decision support, information retrieval and management, and electronic commerce. Because of the horizontal nature of agent technology, it is likely that the successful adoption of agent technology in these areas will have a profound, long-term impact both on the competitiveness and viability of IT industries, and also on the way in which future computer systems will be conceptualized and implemented [18].

What is an agent? Agents can be defined to be autonomous, problem-solving computational entities capable of effective operation in dynamic and open environments [35]. Agents are often deployed in environments in which they interact, and maybe cooperate, with other agents (including both people and software) that have possibly-conflicting aims. Such environments are known as multi-agent systems. Agents can be distinguished from objects (in the sense of object-oriented software) in that they are autonomous entities capable of exercising choice over their actions and interactions. Agents cannot, therefore, be directly invoked like objects. However, they may be constructed using object technology. These notions find application in computer systems relation to several distinct aspects, considered below.

Agents as a design metaphor: Agents provide software system designers and developers with a way of structuring an application around autonomous, communicative elements, and they lead to the construction of software tools and infrastructure to support the design metaphor. In this sense, they offer a new and often more appropriate route to the development of complex systems, especially in open and dynamic environments [15]. In order to support this view of systems development, particular tools and techniques need to be introduced. For example, methodologies to guide analysis and design are required; agent architectures are needed for the design of individual components, and supporting infrastructure (including more general technologies, such as Web Services) must be integrated.

Agents as a source of technologies: Agent technologies span a range of specific techniques and algorithms for dealing with interactions with others in dynamic and open environments. These include issues such as balancing reaction and deliberation in individual agent architectures, learning from and about other agents in the environment and user preferences, finding ways to negotiate and cooperate with agents and developing appropriate means of forming and managing coalitions. Moreover, the adoption of agent-based approaches is increasingly influential in other domains. For example, multi-agent systems can provide faster and more effective methods of resource allocation in complex environments, such as the management of utility networks or logistics scheduling, than any human-centered approach. Similarly, the use of agent systems to simulate real-world domains may provide answers to complex physical or social problems which would be otherwise unobtainable, as in the modeling of the impacts of climate change on various biological populations, or modeling the impact of public policy options on social or economic behavior. Agents offer a new and often more appropriate route to the development of complex systems, especially in open and dynamic environments.

Further reading on the ideas of this section can be found in [17,19,36].

5 Interactive Cooperation

Interaction is a key problem in designing large scale systems that depend on shared knowledge (and semantics) in order to support large communities of people with complex activities and goals. Examples of such systems (all currently in their inception) are semantic web systems, some forms of computational grids and large multi-agent systems. Despite the scale of these systems (or perhaps because it) many fundamental problems of interaction remain unresolved. Let us consider seven of these below, with the aid of an abstract, idealised, conceptual model that covers this class of system.

Figure 1 describes the simplest conceptual model I can imagine that can apply. Each component is a process (in the sense of a computation of some sort running on a CPU of some machine) that interacts with its local environment and for which there is a specification of its competence. Processes also have goals: either generated automatically or injected into the system via interaction with humans or other real-world systems. A competence specification is assumed to be some (partial) description of what the process can do, expressed in a standard formal language. Processes are coordinated via an interaction model that may interact with the processes directly or with their specifications. There may be an interaction between local environments as a consequence of other processes not constrained by an interaction model.

To see how this abstract model applies to one of the concrete application domains, we translate it into the more specific architecture for that domain. For example, we can relate it to OWL-S, one of the standard semantic web service specification languages. Translating this conceptual model to an OWL-S view of semantic web services: a specification is the OWL-S service model (describing the outputs of the process conditional on certain inputs);

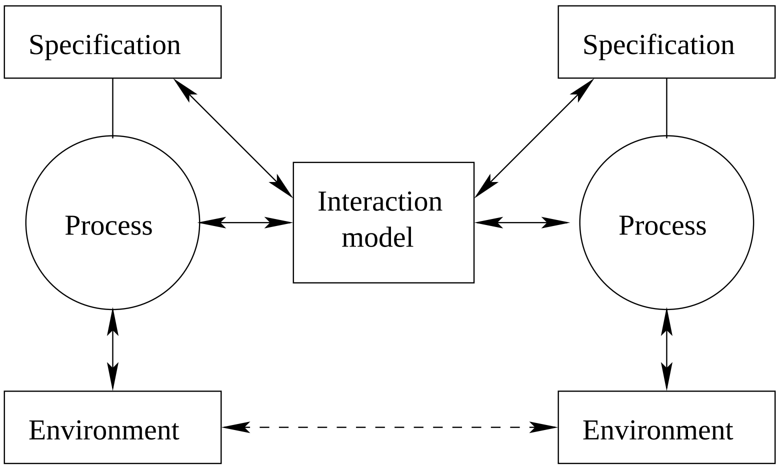

Fig. 1. Conceptual model of process interaction

a process provides the means of enacting that service (via its grounding in a communication language appropriate to the messaging infrastructure); and the interaction model is the OWL-S process model (describing the hierarchical decomposition of processes required to satisfy the specification).

Using the model of Figure 1 as a frame of reference, we can now raise seven questions that stem from vulnerabilities of the model applied in practice. Much of the activity in designing large scale, distributed, systems of the kind described above is stimulated by the search for partial solutions and workarounds for these problems.

The ontology matching problem : Although we assume (for simplicity) a uniform logic, engineers of services will have expressed their ideas in different ways using this logic. We may have a goal and an interaction model result that correspond conceptually but are expressed using different syntax. Conversely, we may have a goal and interaction model result that match but were used to describe different concepts. How can we avoid or detect such situations?

The social norm problem : Our idealised model assumes that models of interaction exist that are appropriate for anticipated goals of the processes in our system. How are these models of interaction determined and how do we now that they describe appropriate social norms?

The coalition formation problem : When we enlist the assistance of some group of processes to interact in some specific way, how do we know we have just those that are essential to the interaction?

The happy ending problem : When a process starts to interact with others using some anticipated model of interaction, how do we ensure that this culminates in obtaining the goal sought by that process through the interaction?

The common knowledge problem : As processes participate in an interaction we require that the knowledge they use to support the interaction remains consistent throughout. It is known to be impossible, in theory, to guarantee this if the processes are asynchronous and if messages between processes may be lost. How do we avoid this theoretical worst case?

The environment scope problem : When a process is participating in an interaction with others it may need to draw on information from its environment. The issue then is how far to explore this environment in order to ensure its information is accurate, particularly if the environment is noisy, uncertain or loosely bounded. How do we ensure that the environment yields information in the right form and of the right quality?

The environmental evolution problem : Interactions take time and during this time the environments of the processes involved may change. It is possible that environmental knowledge which was consistently used by a process at one time during an interaction becomes inconsistent later as the environment alters. How do we avoid the problem of shift in environmental conditions while providing a service?

There are, of course, many views that can be taken of a complex problem and we have explored here only one view. Nevertheless, it is striking how many of these these problems must be viewed as essentially interaction problems, rather than being solved purely through knowledge representation or inference standardisation. It also is interesting historically to observe how few of them are confined to one of the "sub-disciplines" of computing science. Perhaps this is a consequence of the pragmatism required to produce partial solutions to problems that, by their nature, we can control but not domesticate.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] F. Arbab. Reo: A channel-based coordination model for component composition. *Math. Structures in Computer Science*, 14(3):329–366, June 2004.

[3] F. Arbab. Abstract Behavior Types: A foundation model for components and their composition. *Science of Computer Programming*, 55:3–52, March 2005.

[4] F. Arbab, C. Baier, J.J.M.M. Rutten, and M. Sirjani. Modeling component connectors in Reo by Constraint Automata. In *Proc. Int'l Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003), Electronic Notes in Theoretical Computer Science (ENTCS) 97*, pp. 25–46. Elsevier, July 2004.

[5] F. Arbab and J.J.M.M. Rutten. A coinductive calculus of component connectors. In D. Pattinson M. Wirsing and R. Hennicker, eds., *Recent Trends in Algebraic Development Techniques, Proc. 16th Int'l Workshop on Algebraic Development Techniques (WADT 2002), LNCS 2755*, pp. 35–56. Springer-Verlag, 2003.

[6] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[7] J. A. Bergstra and J. W. Klop. Process algebra: specification and verification in bisimulation semantics. In M. Hazewinkel, J. K. Lenstra, and L. G. L. T. Meertens, eds., *Mathematics and Computer Science II*, CWI Monograph 4, pp. 61–94. North-Holland, Amsterdam, 1986.

[8] Curriculum 68: Recommendations for Academic Programs in Computer Science, A Report of the ACM Curriculum Committee on Computer Science. *Comm. ACM* 11(3), Mar. 1968, pp. 151-197.

[9] R. Brooks. Intelligence Without Reason. *MIT AI Lab Technical Report 1293.*

[10] E. Eberbach, D. Goldin, P. Wegner. Turing's Ideas and Models of Computation. In *Alan Turing: Life and Legacy of a Great Thinker*, ed. Christof Teuscher, Springer 2004.

[11] D. Goldin, S. Smolka, P. Attie, and E. Sonderegger. Turing Machines, Transition Systems, and Interaction. *Information and Computation*, 194(2):101–128, Nov. 2004.

[12] D. Goldin and P. Wegner. The Church-Turing Thesis: Breaking the Myth. *LNCS 3526*, Springer 2005, pp.152-168.

[13] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comp.*, 18(1):186–208, 1989.

[14] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1985.

[15] N. R. Jennings. An agent-based approach for building complex software systems. *Comm. ACM*, 44(4):35–41, 2001.

[16] D. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1968.

[17] M. Luck, P. McBurney, and C. Preist. *Agent Technology: Enabling Next Generation Computing. A Roadmap for Agent Based Computing.* AgentLink II, the European Network of Excellence for Agent-Based Computing, Southampton, UK, 2003.

[18] M. Luck, P. McBurney, and C. Preist. A manifesto for agent technology: towards next generation computing. *J. Autonomous Agents and Multi-Agent Systems*, 9(3):203–252, 2004.

[19] M. Luck, P. McBurney, S. Willmott, and O. Shehory. *Agent Technology: Enabling Next Generation Computing. A Roadmap for Agent Based Computing.* AgentLink III, the European Co-ordination Action for Agent-Based Computing, University of Southampton, Southampton, UK, 2005. *In press.*

[20] N. Lynch, M. Tuttle. An Introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, Sep. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[21] R. Milner. *A Calculus of Communicating Systems*, *LNCS 92.* Springer, 1980.

[22] R. Milner. Elements of interaction. *Comm. ACM*, 36(1):78–89, Jan. 1993.

[23] J. K. Rice, J. N. Rice. *Computer Science: Problems, Algorithms, Languages, Information and Computers.* Holt, Rinehart and Winston, 1969.

[24] S. Russell, P. Norveig. *Artificial Intelligence: A Modern Approach.* Addison-Wesley, 1994.

[25] D. Sangiorgi and D. Walker. *The Pi-Calculus - A Theory of Mobile Processes.* Cambridge University Press, 2001.

[26] *SIGACT News*, ACM Press, March 2004, p. 49.

[27] Jan van Leeuwen and Jiří Wiedermann. On the power of interactive computing. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, eds., *Proc. 1st Int'l Conf. on Theoretical Computer Science — Exploring New Frontiers of Theoretical Informatics, IFIP TCS'2000 (Sendai, Japan, August 17-19, 2000)*, *LNCS 1872*, pp. 619–623. Springer-Verlag, 2000.

[28] Jan van Leeuwen and Jiří Wiedermann. Beyond the turing limit: Evolving interactive systems. In L. Pacholski and P. Ruicka, eds., *SOFSEM 2001: 28th Conf. on Current Trends in Theory and Practice of Informatics*, *LNCS 2234*, pp. 90–109. Springer-Verlag, 2001.

[29] Wan Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1999.

[30] P. Wegner. Interaction as a basis for empirical computer science. *ACM Computing Surveys*, 27(1):45–48, March 1995.

[31] P. Wegner. Why Interaction is More Powerful Than Algorithms. *Comm. ACM*, May 1997.

[32] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, Feb. 1998.

[33] P. Wegner and D.Goldin. Interaction as a framework for modeling. *LNCS 1565*, pp. 243–257, 1999.

[34] P. Wegner and D. Goldin. Computation Beyond Turing Machines. *Comm. ACM*, Apr. 2003.

[35] M. J. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[36] F. Zambonelli and H. v. D. Parunak. Signs of a revolution in computer science and software engineering. In P. Petta, R. Tolksdorf, and F. Zambonelli, eds., *Engineering Societies in the Agents World (ESAW 2002)*, Lecture Notes in Artificial Intelligence 2577, pp. 13–28, Berlin, Germany, 2003. Springer.