Computing and Interaction

Farhad Arbab[1,2]

[1] Center for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands
[2] Leiden University, Leiden, The Netherlands

Summary. This chapter offers a rough sketch of the landscape of computing with the specific aim of identifying and interrelating well-established topics such as computability and concurrency to newer areas such as interaction and composition of behavior.

1 Introduction

The size, speed, capacity, and price of computers have all dramatically changed in the last half-century. Still more dramatic are the subtle changes in society's perception of what computers can, should, and are expected to do. Clearly, this change of perception would not have been possible without the technological advances that reduced the size and price of computers, while increasing their speed and capacity. Nevertheless, the social impact of this change of perception and its feedback influence on the advancement of computer science and technology, are too significant to be regarded as mere by-products of those technological advances.

The term *computer* today has a very different meaning than it did in the early part of the twentieth century. Even after such novelties as mechanical and electromechanical calculators had become commonplace in the 1960s, the arithmetic involved in engineering calculations and book-keeping was a time consuming and labor intensive endeavor for businesses and government agencies alike. Analogous to *typist pools* that lingered on until much later, enterprises from engineering and accountant firms to banks and insurance companies employed armies of people to process, record, and extract the large volumes of essentially numerical data that were relevant for their business. Since in the early part of the twentieth century, *computer* was the term that designated these professionals, the machine that could clearly magnify their effectiveness and held the promise of replacing them altogether became known as the *electronic computer*[1].

The social perception of what computers are (to be used for) has evolved through three phases:

1. computers as fast number crunchers;
2. computers as symbol manipulators;
3. computers as mediators and facilitators of interaction.

Two specific transformations marked the above phase transitions. The advent of fast, large main memory and mass-storage devices suitable to store and access the significantly more voluminous amounts of data required for non-numerical symbol manipulation made symbolic computation possible. The watershed that set forth the second transition was the availability of affordable personal computers and digital telecommunication that together fueled the explosion of the Internet.

In spite of the fact that from the beginning, symbol manipulation was as much an inherent ability of electronic computers the juggling of numbers, the perception that computers are really tools for performing fast numerical computations was prevalent. Problems such as information retrieval that did not involve a respectable amount of number crunching were either rejected outright as non-problems, or were considered as problems not worthy of attempts to apply computers and computing to. Subscribers to such views were not all naive outsiders, many an insider considered such areas as business and management, databases, and graphics, to be not only on the fringes of computer applications, but also on the fringes of legitimacy. As late as 1970, James E. Thornton, vice president of Advanced Design Laboratory of Control Data Corporation, who was personally responsible for most of the detailed design of the landmark CDC 6600 computer system, wrote [1]:

> There is, of course, a class of problems which is essentially *noncomputational* but which requires a massive and sophisticated storage system. Such uses as inventory control, production control, and the general category of information retrieval would qualify. *Frankly, these do not need a computer.* There are, however, legitimate justifications for a large computer system as a "partner" with the computational usage. [*Emphasis added.*]

[1] As of the date of this writing, on the etymology of the word "computer" the free encyclopedia Wikipedia (http://en.wikipedia.org/) says: "The word was originally used to describe a person who performed arithmetic calculations and this usage is still valid. The OED2 lists the year 1897 as the first year the word was used to refer to a mechanical calculating device. By 1946 several qualifiers were introduced by the OED2 to differentiate between the different types of machine. These qualifiers included *analogue*, *digital* and *electronic*." According to the free English dictionary Wiktionary (http://en.wiktionary.org), however, the usage of the word "computer" as "a person employed to perform computations" is obsolete.

Of course, by that time many people were not only convinced that legitimate computational applications need not involve heavy number crunching, but were already actively working to bring about the changes that turned fringe activities such as databases and graphics into the core of computing, and reshaped it both as *science* as well as by expanding its domain of applications. Nevertheless, Thornton's statement at the time represented the views of a non-negligible minority that has only gradually diminished since. While the numerical applications of computing have steadily grown in number, size, and significance, its non-numerical applications have simply grown even faster and vaster.

We are still at the tail-end of the second transition (from symbolic computation to interaction) and trying come to terms with its full implications on computer science and technology. This involves revisiting some established areas, such as concurrency and software composition, from a new perspective, and leads to a specific field of study concerned with theories and models for *coordination* of interactive concurrent computations. Pragmatic concerns in software engineering have often driven the advancement of computer science. The transition from symbolic computation to interaction involves, among others, coarse-grain reuse in component based software and (third-party) composition of the behavior of services while their actual software cannot be composed.

Already, a growing number of vendors offer an increasing number of useful computations and services packaged in various forms as specialized hardware and/or software. Together with advanced communication networks, this sets the stage to realize all sorts of new complex applications, from embedded systems with demanding timing requirements to geographically distributed, always-on, dynamically evolving cooperation networks of mobile autonomous agents. Tackling the architectures of complex systems whose organization and composition must dynamically change, e.g., to accommodate mobility, or evolve and be reconfigured to adapt to short- as well as long-term changes in their environment, presents new challenges in software engineering.

Two key concepts emerge as core concerns: (1) interaction, and (2) compositionality. While researchers have worked on both individually in the past, we propose that their combination deserves still more serious systematic study because it offers insight into new approaches to coordination of cooperating interacting components that comprise such complex systems.

2 Computing

The formal notions of *computing* and *computability* were introduced by Alonzo Church (1903–1995), in terms of λ-calculus, and Alan Turing (1912–1954), in terms of Turing machines. Both Church and Turing were inspired by David Hilbert's (1862–1943) challenge proposed in his 1900 lecture delivered before the International Congress of Mathematics at Paris, to define a solid foundation for (mechanical) effective methods of finding mathematical truth.

Hilbert's program consisted of finding a set of axioms as the unassailable foundation of mathematics, such that only mathematical truths could be derived from them by the application of any (truth preserving) mechanical operation, and that all mathematical truths could be derived that way.

But, what exactly is a mechanical operation? This was what Church, Turing, and others were to define. Turing himself also intended for his abstract machine to formalize the workings of the human mind. Ironically, his own reasoning on the famous halting problem can be used to show that Turing machines cannot find all mathematical truths, let alone model the workings of the human mind[2]. Kurt Godel's (1906–1978) incompleteness theorem of 1931, which brought the premature end of Hilbert's program for mathematics, clearly shows the limits of formal systems and mechanical truth derivation methods. By his halting problem, Turing intended to provide a constructive proof of Godel's incompleteness theorem: they both show that there are (even mathematical) truths that cannot be derived mechanically, and interestingly in both cases, the crucial step in the proof is a variation of the diagonalization technique first used by Georg Cantor (1845–1918) to show that the infinity of real numbers between any two numbers is greater than the infinity of natural numbers.

It is far from obvious why Turing's simple abstract machine, or Church's λ-calculus, is a reasonable formalization of what we intuitively mean by any mechanical operation. However, all extensions of the Turing machine that have been considered, are shown to be mathematically equivalent to, and no more powerful than, the basic Turing machine. Turing and Church showed the equivalence of Turing machines and λ-calculus. This, plus the fact that other formalizations, e.g., Emil Post's (1897–1954), have all turned out to be equivalent, has increased the credibility of the conjecture that a Turing machine can actually be made to perform any mechanical operation whatsoever. Indeed, it has become reasonable to mathematically define a mechanical operation as any operation that can be performed by a Turing machine, and to accept the view known as the Church–Turing thesis: that the notion of Turing machines (or λ-calculus, or other equivalents) mathematically defines the concept of an algorithm (or an effective, or recursive, or mechanical procedure).

[2] Intuitively, human beings believe that the human mind can perceive truths beyond mathematics. If so, the working of the human mind is likely beyond the scope of our formal systems. This may be because as Penrose argues [2], what goes on in the human mind is substantially different than what our formal systems express. He proposes that to comprehend the human mind, we require a hitherto lacking, fundamentally important insight into physics, which is also a prerequisite for a unified theory of everything.

3 Interaction

The Church–Turing thesis can simply be considered as a mathematical definition of what computing is in a strictly technical sense; it reflects the notion of computing of functions. Real computers, on the other hand, do much more than mere computing in this restrictive sense. Among other things, they are sources of heat and noise, and have always been revered (and despised) as (dis)tasteful architectural artifacts, or pieces of furniture. More interestingly, computers also interact: they can act as facilitators, mediators, and coordinators that enable the collaboration of other agents. These other agents may in turn be other computers (or computer programs), sensors and actuators that involve their real world environment, or human beings. The role of a computer as an agent that performs computing, in the strict technical sense of the word, should not be confused with its role as a mediator agent that, e.g., empowers its human users to collaborate with one another (including, for instance, word-processing, where a single user engages in self-collaboration over a span of time). The fact that the computer, in this case, may perform some computation in order to enable the collaboration of other agents, is ancillary to the fact that it needs to interact with these agents to enable their collaboration. To emphasize this distinction, Wegner proposes the concept of an interaction machine [3, 4, 5]. Some of the formal aspects of interaction machines are discussed in [6, 7, 8, 9]. Here we focus on the essential difference between interaction machines and Turing machines.

A Turing machine operates as a closed system: it receives its input tape, starts computing, and (hopefully) halts, at which point its output tape contains the result of its computation. In every step of a computation, the symbol written by a Turing machine on its tape depends only on its internal state and the current symbol it reads from the tape. An interaction machine is an extension of a Turing machine that can interact with its environment with new input and output primitive actions. Unlike other extensions of the Turing machine (such as more tapes, more controls, etc.) this one actually changes the essence of the behavior of the machine. This extension makes interaction machines *open systems.*

Consider an interaction machine I operating in an environment described as a dynamical system E. The symbol that I writes on its tape at a given step, not only depends on its internal state and the current symbol it reads from the tape, but can also depend on the input it obtains directly from E. Because the behavior of E cannot be described by a computable function, I cannot be replaced by a Turing machine. The best approximation of I by a Turing machine, T, would require an encoding of the actual input that I obtains from E, which can be known only *after* the start of the computation. The computation that T performs, in this case, is the same as that of I, but I does more than T because it interacts with its environment E. What T does, in a sense, is analogous to predicting yesterday's weather: it is interesting that it can be done (assuming that it can be done), but it doesn't quite pass muster! To emphasize the distinction, we can imagine that the interaction of I with E is not limited to just one input: suppose I also does a direct output to E, followed by another direct input from E. Now, because as a dynamical system, E is non-computable, and the value of the second input from E to I depends on the earlier interaction of E and I, no input tape can encode this "computation" for any Turing machine.

It is the ability of computers (as interaction machines) to interact with the real world, rather than their ability (as mere Turing machines) to carry on ever-more-sophisticated computations, that is having the most dramatic impact on our society. In the traditional models of human–computer interaction, users prepare and consume the information needed and produced by their applications, or select from the alternatives allowed by a rigid structure of computation. In contrast to these models, the emerging models of human–computer interaction remove the barriers between users and their applications. The role of a user is no longer limited to that of an observer or an operator: increasingly, users become active components of their running applications, where they examine, alter, and steer on-going computations. This form of cooperation between humans and computers, and among humans via computers, is a vital necessity in many contemporary applications, where realistic results can be achieved only if human intuition and common-sense is combined with formal reasoning and computation.

For example, *computational steering* allows human experts to intervene and guide an on-going computation with which they interact through visualizations of various scalar, vector, and tensor fields. Construction and manipulation of complex simulation models that use numerical approximation and solutions of partial differential equations, e.g., in computational fluid dynamics and biology, already benefit from such techniques. The applications of computer facilitated collaborative work are among the increasingly important areas of activity in the foreseeable future. They can be regarded as natural extensions of systems where several users simultaneously examine, alter, interact, and steer on-going computations. The promise of *ubiquitous computing* requires the full harnessing of the potential of these combinations. Interaction machines are suitable conceptual models for describing such applications.

Interaction machines suggest a new perspective on composition. Traditionally, software composition has focused on *composition of algorithms*, where (the designer of) one algorithm, as part of its own internal logic, decides to engage another algorithm, e.g., through a function call or a method invocation. Composed behavior ensues as a consequence of composing algorithms and its implied flow of control. Interaction machines are self-contained entities that directly neither offer nor engage algorithms. They can be arranged by third parties to engage one another only through their mutual interactions, which involve no flow of control. This leads to *composition of behavior* where the algorithms (embedded in the individual interaction machines) involved in a composed system do not directly engage each other and (their designers) remain oblivious to their composition.

Van Leeuwen and Wiedermann offer a formal treatment of some of the implications of interactive computing and its relationship with the more traditional views of computability in [10]. Goldin et al. [11] propose persistent Turing machines (PTMs) as a stream-based extension to the Turing machine model with persistence and the same notion of interaction as in interaction machines. They investigate the "minimal" changes to the Turing machine model necessary for capturing the extra expressive power conjectured by Wegner for interaction machines over Turing machines, using a general kind of transition system called interactive transition systems (ITSs) as reference. They show an isomorphism that implies every equivalence result over PTMs carries over to ITSs, and vice versa.

Interaction machines have unpredictable input from their external environment, and can directly affect their environment, unpredictably, due to such input. Because of this property, interaction machines may seem too open for formal studies: the unpredictable way that the environment can affect their behavior can make their behavior underspecified, or even ill-defined. But, this view is misleading. Interaction machines are both useful and interesting for formal studies.

On the one hand, the openness of interaction machines and their consequent underspecified behavior is a valuable true-to-life property. Real systems are composed of components that interact with one another, where each is an open system. Typically, the behavior of each of these components is ill-defined, except within the confines of a set of constraints on its interactions with its environment. When a number of such open systems come together as components to comprise a larger system, the topology of their interactions forms a context that constrains their mutual interactions and yields well-defined behavior.

On the other hand, the concept of interaction machines suggests a clear separation of concerns for the formal study of their behavior, both as components in a larger system, as well as in isolation. Just like a Turing machine, the behavior of an interaction machine can be studied as a computation (in the sense of the Church–Turing thesis) between each pair of its successive interactions. More interestingly, one can abstract away from all such computations, regarding them as internal details of individual components, and embark on a formal study of the constraints, contexts, and conditions on the interactions among the components in a system (as well as between the system and its environment) that ensure and preserve well-behavedness.

Consider, for example, constructing a simple system using three black-box components: a clock, a thermometer, and a display. The clock has an output port through which it periodically produces a string of characters that represents the current time. Similarly, the thermometer has an output port through which it periodically produces a string of characters that represents the current temperature. The display has an input port through which it periodically consumes a string of characters and displays it. Our goal is to build a system—similar to what one finds on top of some tall bank buildings—

that alternately displays the current time and current temperature. It is the constraints on the periods and the relative order of exchanges between these three components that together shape the desired alternating behavior in our composed system. It is at least as essential to study and express these intercomponent constraints that define the behavior of a composed system, as it is to study and specify the computation carried out by each of its individual components. It is even more sensible to focus on such protocols and constraints in isolation from intracomponent computation concerns. And this material is the thread that weaves the fabric of *coordination*.

4 Concurrency

The concept of interaction is closely related to concurrency. Concurrency means that different computations in a system overlap in time. The computations in a concurrent system may be interleaved with one another on a single processor or actually run in parallel (i.e., use more than one physical processor at a time). Parallelism introduces extra concerns (over monoprocessor computing) such as interprocessor communication, the links that carry this communication, synchronization, exclusion, consensus, and graceful recovery or termination in case of partial failures. The parallel computations in a system may or may not be geographically distributed. Geographic distribution escalates the significance of the extra concerns in parallel computing by increasing communication link delays, potential for partial failures, and the difficulty of maintaining consistency, which together make schemes based on central control and global views less tenable in practice.

Nevertheless, concurrency in itself does not change the essence of computing. Clearly, interleaving is but one specific regiment for programming a Turing machine. Parallelism, on the other hand, involves multiple Turing machines. Although not obvious at the outset, it turns out that involving multiple Turing machines does not increase their expressiveness: parallel systems are mathematically equivalent to a single Turing machine. This is not so for interactive systems. What distinguishes an interactive system from other concurrent systems is the fact that an interactive system has unpredictable input from an external environment that it does not control.

The theoretical equivalence of (closed) concurrent systems and a Turing machine is of little practical use. It is far more difficult to consider, design, and reason with a set of concurrent activities than it is to do so with individual sequential activities; the whole, in this case, is considerably more (complex) than the sum of its parts.

The study and the application of concurrency in computer science have a long history. The study of deadlocks, the dining philosophers problem, and the definition of semaphores and monitors were all well established by the early 1970s. Theoretical work on concurrency, e.g., CSP [12, 13], CCS [14], process algebra [15], and π-calculus [16], has helped to show the difficulty of dealing with concurrency, especially when the number of concurrent activities becomes large. Most of these models are more effective for describing closed systems. A number of programming languages have been based upon some of these theoretical models, e.g., Occam [17] uses CSP and LOTOS [18] uses CCS. However, it is illuminating to note that the original context for the interest in concurrency was somewhat different than the demands of the applications of today in two respects:

- In the early days of computing, hardware resources were prohibitively expensive and had to be shared among several programs that had nothing to do with each other, except for the fact that they were unlucky enough to have to compete with each other for a share of the same resources. This was *concurrency of competition*. Today, it is quite feasible to allocate tens, hundreds, and thousands of processors to the same task (if only we could do it right). This is *concurrency of cooperation*. The distinction is that whereas it is sufficient to keep independent competing entities from trampling on each other over shared resources, cooperating entities also depend on the (partial) results they produce for each other. Proper passing and sharing of these results require more complex protocols, which become even more complex as the number of cooperating entities and the degree of their cooperation increase.
- It was only in the 1990s that the falling costs of processor and communication hardware dropped below the threshold where having very large numbers of "active entities" in an application makes pragmatic sense. Massively parallel systems with thousands of processors are a reality today. Current trends in processor hardware and operating system kernel support for threads make it possible to efficiently have in the order of hundreds of active entities running in a process on each processor. Thus, it is not unrealistic to think that a single application can be composed of hundreds of thousands of active entities. Compared to classical uses of concurrency, this is a jump of several orders of magnitude in numbers. When a phenomenon is scaled up by several orders of magnitude, originally insignificant details and concerns often add up to the extent that they can no longer be ignored; we have not just a quantitative change (i.e., more of the same thing), but rather a qualitative change (i.e., involving new properties, or even a whole new phenomenon). In our view, grappling with massive concurrency requires a qualitative change in (classical) models of concurrency.

The primary concern in the design of a concurrent application must be its model of cooperation: how the various active entities comprising the application are to cooperate with each other. Eventually, a set of communication primitives must be used to implement whatever model of cooperation application-designers opt for; the concerns for performance may indirectly affect their design.

It is important to realize that the conceptual gap between the system supported communication primitives and a concurrent application must often be filled with a nontrivial model of cooperation. Ideally, one should be able to design and understand a concurrent system by separately understanding its individual active entities, and how they cooperate. Precise description of how this cooperation is to materialize has a shorter history than models, methods, and languages for precise descriptions of individual active entities. Various ad hoc libraries of functions (e.g., PVM [19], MPI [20], and CORBA [21]) have emerged as the so-called *middle-ware* layer of software to fill this conceptual gap by providing higher-level support for developing concurrent (and especially distributed) applications on top of the lower-level communication models offered by operating system platforms.

The two classical approaches to construction of concurrent systems are shared memory and message passing. In the shared memory model, a piece of real, virtual, or conceptual memory is simultaneously made available to more than one entity, which share accessing and modifying its contents through atomic read/write or store/load operations. In the message-passing model, entities communicate and synchronize by explicit exchange of messages.

In the shared memory model, communication is only a side effect of the timing of the memory access operations that its subscribing entities perform, and of the delay patterns induced by the inherent synchronization imposed by their atomicity. Participation of an entity in any specific exchange, and the whole communication protocol, are strongly influenced by ephemeral timing dependencies. These dependencies are equally likely to arise out of errors, (lucky or unfortunate) coincidences, or subtle implicit ordering and data dependencies that emerge from the global semantics of an application. The shared memory model inherently supports indirect, anonymous communication among participating entities whose activities are decoupled from one another in the temporal domain. But communication is not always explicitly obvious in shared memory models.

Communication is the primary concern in message passing models, and the synchronization involved, if any, is only a side effect of what it takes to realize communication. There are indeed many substantially different variants of message passing. Messages can be targeted or untargeted and the exchange of a message may or may not involve a synchronizing rendezvous between its sender and receiver. Object oriented programming ties the semantics of message passing together with method invocation. This further complicates the semantics of message passing by implicating the semantics of the invoked method and the states of the entities involved in its execution. For instance, when an object invokes a method m of another object, o, it expects o to perform something "meaningful" as suggested by the name of the method m. The (future) state of the calling object may depend on the fulfillment of this expectation, which itself involves assumptions about the actual semantics of the method m, as well as the state of the object o.

While each of the variants of shared memory and message passing communication models is useful for construction of concurrent systems, composition of systems involving many active entities raises a number of issues that go beyond concerns for communication of their constituent entities. We address this in the next section.

5 Composition

From houses and bridges to cars, aircraft, and electronic devices, complex systems are routinely constructed by putting simpler pieces together. This holds for software construction as well. We call a software construction compositional (with respect to a set of properties) only if the properties of the resulting system can be defined as a composition of the properties of its constituent parts. For instance, given the memory requirements M_p and M_q of two programs p and q, the memory requirement of a system constructed by composing p and q can be computed as a composition of M_p and M_q (e.g., $M_p + M_q$, $\max(M_p, M_q)$, etc., depending on how they are composed). On the other hand, the deadlock-freedom property of a system composed out of p and q cannot always be derived as a composition of the deadlock-freedom properties of p and q.

According to one trivial interpretation of this definition, all software construction is compositional: every complex piece of software eventually consists of some composition of a set of primitive instructions, and in principle, its properties can always be derived by applying its relevant rules of composition to the properties of those primitives. This is precisely how one formally derives the semantic properties of relatively simple programs from those of their primitive instructions. However, this trivial interpretation of compositionality quickly becomes uninteresting and useless for complex concurrent systems, for the same reason that deriving interesting properties of a complex piece of mechanical machinery from those of its constituent atoms is intractable. With only a smidgen of exaggeration, one can say that attempting to derive the dynamic run-time behavior of such software in this way is as hopelessly misguided as trying to derive the properties of a running internal combustion engine from an atomic particle model of the engine, its fuel, air, and electricity.

To be useful, our definition of compositionality must be augmented with appropriate definitions of "its constituent parts" and "the properties" that we are interested in. Both of these notions are manifestations of abstraction. Instead of considering individual primitive instructions as the constituents of a complex system, we must identify parts of the system such that each part consists of a (large) collection of such primitives whose precise number and composition we wish to abstract away as internal details of that part. The properties of a collection of primitive instructions that are abstracted away as internal details of a part, versus those that are exposed as the properties of the part, play a crucial role in defining the effectiveness of an abstraction and the flexibility of a composition. The more properties we hide, the more effective an abstraction we have, allowing more freedom of choice in selecting the precise collection or sequence of instructions that comprise an implementation of a part. On the other hand, the less properties we expose, the less of an opportunity we leave for individual parts to affect and be affected by the exposed properties of other parts. This, in turn, restricts the possibility of influencing the role that a given part can play in different compositions.

To identify the exposed properties of a part that can and cannot be influenced through its composition with other parts, we distinguish between its behavior versus its semantics. To show the usefulness of this distinction, consider a simple adder as a (software) part (for instance, consider this adder as a process, an agent, an object, a component, etc.). This adder takes two input values, x and y, and produces a result, z, which is the sum of x and y. For this adder to be useful, it must expose its property of how it relates the values x, y, and z, that is $z = x + y$. We call this the *semantics* of the adder because it reflects the meaning of what it does. In addition to this semantics, successful composition of this adder as a part in any larger system requires the knowledge of certain other properties of the adder that must also be exposed. For instance, we need clear answers to the following questions:

- Does the adder consume x and y in a specific order, or does it consume whichever arrives first?
- Does it consume x and y only when both are available?
- Does it consume x and y atomically, or in separate steps that can potentially be interleaved with other events?
- Does it produce z in a separate step, with possible interleaving of other events, or does it compute and produce z atomically together with:
  - the atomic consumption of both x and y, or
  - the consumption of x or y, whichever is consumed last?

The answers to such questions define the (externally observable) *behavior* of the adder, above and beyond its mere semantics. It is clear that even in the simple case of our trivial adder, different alternative answers to the above questions are possible, which means we can have different adders, each with its own different (externally observable) behavior, all sharing (or implementing) the same semantics, i.e., $z = x + y$.

The distinction between behavior and semantics is important in composition of all concurrent systems. However, it becomes essential in concurrent systems where autonomy, anonymity, and reuse of parts comprise a primary concern. Such is the case for a system composed of interacting machines, which we contend serves as the best model for component-based concurrent software. Components are expected to be independent commodities, viable in their binary forms in the (not necessarily commercial) marketplace, developed, offered, deployed, integrated, and maintained, by separate autonomous organizations in mutually unknown and unknowable contexts, over long spans of time. It is impossible to determine the properties of a system composed out of a set of components without explicit knowledge of both (1) the relevant behavioral properties of the components, and (2) the composition scheme's rules that affect those properties.

Traditional schemes for composition of software parts into more complex systems rely on variants of procedure call (including method invocation of object oriented models). Typically, each such scheme specifies much of the extra-semantic properties of the behavior of the composed system by pre-defining aspects of composition such as the (non)atomicity of the call and its return result, synchronization points, permissible concurrency, etc. This limits composition alternatives and restricts the possible behavior that can be obtained by composing a given set of software part to the choices prescribed in that scheme. Moreover, composition through procedure calls requires an intimate familiarity of the caller with the semantics of the called procedure (or method), which creates an asymmetric semantic dependency between the two. This semantic dependency, together with the unavailability of (or stringent restrictions on) the means to control the extra-semantic behavioral properties of a software composition at its composition time, severely limit the range of possible variations that can be composed out of the same set of software parts, which in turn limits the reusability of those software parts.

Component composition is expected to be more flexible than other forms of software composition, such as module interconnections, method invocations, or procedure calls. It is expected to allow the same components to play different roles in different compositions. This flexibility requires the ability to influence the behavior of components at the time of their composition and places the emphasis in composition on interaction. Coordination models and languages [22] address precisely the issues involved in managing the interactions among the constituents of a concurrent system into a coherently coordinated cooperation. However, the different mechanisms that various coordination models offer to manage interaction do not all equally support the increased level of flexibility required in component composition.

In the chapter "Composition of Interacting Computations" in this book, we present a brief overview of coordination models and languages and offer a framework for their classification. We then describe a specific model, called Reo [23], that uniquely uses interaction as its only primitive concept for compositional construction of component coordination protocols.

6 Discussion

The classical notion of computing was forged to formalize and study the algorithmic aspects of computing mathematical functions. Real computers do more than compute mathematical functions; they also interact. Interaction is an increasingly important aspect of the behavior of our modern (hardware and software) computing devices, which often act as agents that engage and communicate with other agents in the real world. Interaction is also the key concern in the composition of complex computing systems out of independent building block components that often run concurrently with one another. The model of interaction machines extends the notion of computing, as what real computing devices do, beyond the classical notion of computing, as algorithmic evaluation of mathematical functions.

Our society increasingly relies on computing devices not only as number crunchers and symbol manipulators, but more importantly, as mediators and facilitators of interaction. Models of computation that incorporate interaction as a primitive concept on a par with that of algorithmic computing form the foundation for study, understanding, and reliable construction of modern computing.

References

1. Thornton, J.: Design of a Computer: The Control Data 6600. Scott, Foresman and Company, 1970.
2. Penrose, R.: The Emperor's New Mind. Oxford University Press, 1990.
3. Wegner, P.: Interaction as a basis for empirical computer science. ACM Computing Surveys **27**, 1995, pp. 45–48.
4. Wegner, P.: Interactive foundations of computing. Theoretical Computer Science **192**, 1998, pp. 315–351.
5. Wegner, P., Goldin, D.: Computation beyond Turing machines. Communications of the ACM **46**, 2003.
6. Wegner, P., Goldin, D.: Coinductive models of finite computing agents. In: Proc. Coalgebraic Methods in Computer Science (CMCS). Volume 19 of Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier, 1999.
7. van Leeuwen, J., Wiedermann, J.: On the power of interactive computing. In van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T., eds.: Proceedings of the 1st International Conference on Theoretical Computer Science — Exploring New Frontiers of Theoretical Informatics, IFIP TCS'2000 (Sendai, Japan, August 17-19, 2000. Volume 1872 of LNCS. Springer-Verlag, Berlin-Heidelberg-New York-Barcelona-Hong Kong-London-Milan-Paris-Singapore-Tokyo, 2000, pp. 619–623.
8. van Leeuwen, J., Wiedermann, J.: Beyond the turing limit: Evolving interactive systems. In Pacholski, L., Ruicka, P., eds.: SOFSEM 2001: Theory and Practice of Informatics: 28th Conference on Current Trends in Theory and Practice of Informatics. Volume 2234 of Lecture Notes in Computer Science. Springer-Verlag, 2001, pp. 90–109.
9. Wegner, P., Goldin, D.: Interaction, computability, and church's thesis. British Computer Journal, 2005 (to appear).
10. van Leeuwen, J., Wiedermann, J.: A Theory of Interactive Computation. In: [24], 2006.
11. Goldin, D., Smolka, S., Attie, P., Sonderegger, E.: Turing machines, transition systems, and interaction. Information and Computation Journal **194**, 2004, pp. 101–128.
12. Hoare, C.: Communicating Sequential Processes. Communications of the ACM **21**, 1978.
13. Hoare, C.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science. Prentice-Hall, 1985.

14. Milner, R.: Communication and Concurrency. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.
15. Bergstra, J., Klop, J.: Process algebra for synchronous communication. Information and Control **60**, 1984, pp. 109–137.
16. Milner, R.: Elements of interaction. Communications of the ACM **36**, 1993, pp. 78–89.
17. INMOS Ltd.: OCCAM 2, Reference Manual. Series in Computer Science. Prentice-Hall, 1988.
18. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networds and ISDN Systems **14**, 1986, pp. 25–59.
19. (PVM) http://www.csm.ornl.gov/pvm.
20. (MPI) http://www-unix.mcs.anl.gov/mpi/.
21. (CORBA) http://www.omg.org.
22. Papadopoulos, G., Arbab, F.: Coordination models and languages. In Zelkowitz, M., ed.: Advances in Computers – The Engineering of Large Systems. Volume 46. Academic Press, 1998, pp. 329–400.
23. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14**, 2004, pp. 329–366.
24. Goldin, D., Smolka, S., Wegner, P., eds.: Interactive Computation: The New Paradigm. Springer-Verlag, 2006 (this volume).