Computers as interactive machines: Can we build an explanatory abstraction?

Alice Martin, Mathieu Magnaudet, Stéphane Conversy

Computers as interactive machines: Can we build an explanatory abstraction?

Alice Martin, Mathieu Magnaudet, Stéphane Conversy
ENAC Université de Toulouse
firstname.name@enac.fr

PREPRINT. *Minds and Machines*. Accepted: 5 January 2023

In this paper, we address the question of what current computers are from the point of view of human-computer interaction. In the early days of computing, the Turing machine (TM) has been the cornerstone of the understanding of computers. The TM defines what can be computed and how computation can be carried out. However, in the last decades, computers have evolved and increasingly become interactive systems, reacting in real-time to external events in an ongoing loop. We argue that the TM does not provide a mechanistic explanation for interactive computing. The reason is that the fundamental phenomena relevant to interactive computing are out of the scope of classical computability theory. Part of the explanatory power of the TM relies on what we propose to call an execution model. An execution model belongs to a level of abstraction where it is possible to describe both the functional architecture and the execution in mechanistic terms. An updated execution model is warranted to provide the minimal mechanistic description for interactive computation as a counterpart of what the TM could explain regarding Church-Turing computation. It would support an explanation of the ubiquitous computing devices we know - those interacting with humans, e.g., through digital interfaces. We show that such a model is not available within interactive models of computation and that relevant abstractions and concerns are available in computer engineering but need to be identified and gathered. To fill this void, we propose to reflect on the level of abstractionrequired to support the mechanistic description of an interactive execution and propose some preliminary requirements.

Keywords : Philosophy of computing · Computers · Turing Machine · Interactive computing · Model of computation · Mechanism · Human-computer interaction

Introduction

Both computing-minded philosophers (Piccinini, 2008a; Rapaport, 2018) and philosophy-minded computer scientists (Smith, 2002) have asked what computers are. However, philosophical answers have paid little or no attention to an important property of current computers: their interactive nature. Every usable computing device, from the smartwatch on our wrist to the complex computing systems embedded in the cockpit of an aircraft relies on a complex entanglement of interactions between incoming external events and computational processes. From an epistemological point of view, these devices raise new challenges as they exhibit some properties that seem beyond the scope of the classical theory of computation.

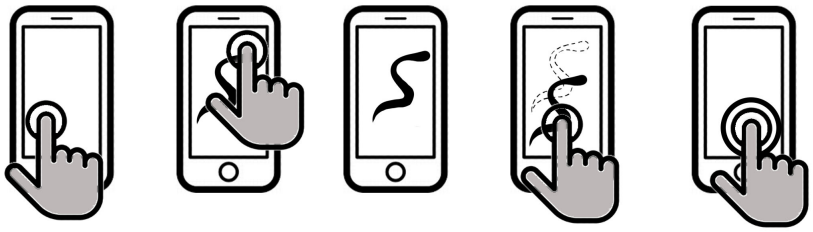

Figure 1: Example of an interactive system: a drawing app

To understand what makes interactive computers special, we focus on a subset of interactive devices, namely those designed and studied by the Human-computer Interaction (HCI) community (Myers and Rosson, 1992; Beaudouin-Lafon, 2006; Myers et al., 2008; Hornbaek and Oulasvirta, 2017; Basman et al., 2018). Consider this simple example illustrated in Figure 1): a drawing application on a smartphone. When one touches the screen and moves a finger over it, the application draws a line whose thickness depends on the pressure applied to the screen. If the user releases the pressure, the drawing remains and becomes an object on the screen to be interacted with. The drawing is erased if the user does a double tap over the screen.

As simple as it is, this example reveals some interesting phenomena:

1. A physical event triggers the drawing. Thus, there is a causal link between physical and computational processes.

2. One property of the drawing (the thickness of the line) is entirely dependent on the pressure of the finger, making the computational process responsive to the structure of the physical event.

3. During the execution, the drawing becomes a new object on the user interface and can be interacted with. In other words, the production of outputs is dynamic (there is no need to wait for the termination of the execution), and outputs can affect future inputs, in a feedback loop.

4. The double-tap behavior involves measuring the time that has elapsed between two taps.

None of these phenomena can be explained in the classical epistemic framework, where a computer is understood on a formal level through the theory of computability, or on a concrete level through a specific model of computer architecture. On the formal side, we are left with automata theory and formal language recognition. On the concrete side, a classical framework such as the Turing Machine describes a specific computer architecture model and its functioning: when one talks about a computing system performing Turing-Church computations, the model involves a binary alphabet, a memory unit, and a processing unit whose execution stops when the output has been reached. When trying to describe current interactive computers, we argue that some relevant phenomena for an explanation are simply out of the scope of the classical framework.

Faced with this discrepancy between the classical framework and the actual functioning of computers, some philosophers have encouraged a reevaluation of the epistemic stance toward computers. Philosophical accounts have delineated typologies instead to distinguish

types of computers in terms of functionalities (Piccinini, 2008a) or have deconstructed the idea that computability theory is per se a complete story of computing phenomena (Smith, 2002). Others have developed the idea that transcending Turing computability is required to deepen our understanding of computation for mathematical (Goldin and Wegner, 2008; Dodig-Crnkovic, 2011) and epistemic (MacLennan, 2003) reasons. In any case, the debate asks whether the classical notion of computation can capture relevant phenomena to (mathematically) formalize and (epistemically) understand current computing systems. However, to the best of our knowledge, no mechanistic description of interactive computing has been proposed in philosophy.

To get a clearer epistemic view of interactive computers, we propose introducing the concept of an execution model. What we call an execution model explains what a computer does by describing in mechanistic terms how instruction is carried out on some functional architecture. It is posited at a level of abstraction between the purely formal model of computation and the physical-mechanical model of the hardware.

Computing in the early days could find its execution model in the Turing machine (TM), although the TM was not initially concerned with modeling an actual computer (computers did not even exist in 1937). But the singularity of the TM and its explanatory power relied at least upon its ability to be a twofold abstraction. Not only could the TM support the formalization of the set of computable functions (what can be computed), but it also provided the intuition of the mechanism (Bozşahin, 2018) that could achieve such a computation (how it can be executed). This specific feature had already been commented on by Gödel, when comparing the TM to other equivalent formalism (Shagrir, 2006). To refer to the twofold nature of the TM, we use in this paper a distinction between a *model of computation* and an *execution model*. On the contrary, other models of computation, such as the lambda-calculus, do not carry any reference to execution and do not allow any mechanistic description of a computational behavior.

In this paper, we argue that the execution model provided by the Turing machine in the early days of computing cannot account for the properties of interactive computing today, in particular for the computing devices we interact with as users. We add that there is currently no candidate among interactive models of computation for an updated execution model. We are therefore left with an explanatory gap: if one wants to have an explanatory story about the existing computing systems, one needs to provide the right execution model.

This paper is organized as follows: Section 1 conceptualizes the difference between models of computation and execution models. Section 2 reminds how the understanding of digital computers has been initially related to Universal Turing machines. We argue such a view cannot hold when addressing the issue of interactive systems. The section then presents an overview of existing accounts of interactive computing and shows why we do not find the execution model we are looking for among these accounts either. Section 3 proposes requirements regarding an execution model for interactive computing systems and introduces a first sketch of such a model. Section 4 specifies the explanatory role of an execution model and the level of abstraction at which it is situated.

1 Execution models vs. models of computation

Computability theory has provided a framework for building models of computation. As has been proved at length, different formalizations of the concept of computation are equivalent, the most famous being Church's lambda calculus (Church, 1940) and the TM (Turing, 1937). If formally equivalent, a distinction can be made between the lambda calculus

and the TM, as already pointed out in the literature since Gödel (Shagrir, 2006). The TM does also provide an execution model. At least intuitively, a computation formalized as the reduction of a function does not suggest an implementing mechanism in the same sense that the TM does. It would be worth introducing a nuance here, namely that the lambda-calculus inspired later the Lisp language [1]. In any case, the TM and Lisp are abstractions that are able to describe an execution.

We argue that the kind of distinction made initially between the lambda calculus and the TM (although the LISP machine could potentially weaken the distinction) is a distinction between a *model of computation* and an *execution model*. We think such a distinction applies to a *model of interactive computation* and a *model of interactive execution* and is relevant to describe interactive computing systems.

1.1 The current purposes of models of computation

Current models of computation in computer science play today a different role than in the 1930s mathematical realm. In computer science, what makes a model of computation valuable is related to the formal properties it expresses. Once those formal properties are at hand, they allow further procedures to be acted upon them, especially system verification and certification.

In the end, models of computation serve as tools to support and verify the design of a system. These models belong to a particular level of abstraction: they intend to model something other than the system as a whole and the way it works. They focus on verifiable properties, upon which proofs that guarantee the outputs of the system are built. As an example, let us consider the design of commands for an airplane. The designer needs to write a program that describes the behavior of the commands. It is up to the designer to decide which properties matter the most and must be expressed in the model. Those properties to be checked can be, e.g., bounded values for the range of inputs the system can take (to guarantee the "flight envelope", maintaining the correct speed to avoid stall), the absence of infinite loops, or that of memory overflows. The rest can be abstracted as irrelevant to the specific verification task.

1.2 Purpose of execution models

We call an *execution model* the mechanistic description of a computing execution based on some functional architecture. Such a model supports an explanation of the behavior of a computing system in mechanistic terms. In other words, it explains how computation is carried out by defining the components of the system, their properties, and relationships.

Verifying formal properties is different from investigating why the system behaves the way it does. There are two different tasks. The former task belongs to applied mathematics. It describes abstract computations through formal models by focusing on specific properties. The latter is left to the epistemologist and is the question the philosopher

[1] Steve Russell, a student of McCarthy, showed in 1958 that the eval function of Lisp could serve as a concrete abstract machine and be directly implemented: "But in late 1958, Steve Russell, one of McCarthy's grad students, looked at this definition of eval and realized that if he translated it into machine language, the result would be a Lisp interpreter. This was a big surprise at the time. Here is what McCarthy said about it later: Steve Russell said, look, why don't I program this eval, and I said to him, ho, ho, you're confusing theory with practice, this eval is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the eval in my paper into [IBM] 704 machine code, fixing bugs, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today."(Graham, 2004), page 185.

is interested in when asking what a computer is. It requires something other than task-oriented formalizations of properties abstracted away from any physical mechanism. What the epistemologist needs to make sense of (the overall behavior) belongs to another level of abstraction.

Computations and their models belong to a level of abstraction independent of implementation details. Computations, as already coined, are "medium-independent" (Klein, 2020). On the contrary, to have a model of some execution belongs to a lower level of abstraction, where minimal references to the devices that allow the execution are made. Still, there is no need to dig into fine-grained implementation details to make sense of computing behavior in mechanistic terms. The TM model for classical computation exemplifies the required level of abstraction.

1.3 The Turing Machine: the execution model for classical computation

We argue that the TM also provides an execution model. Some would object that the Turing Machine does not describe a mechanism, in the sense that Turing's description of his abstract machine provides no hint about how the tape header works, for example. In other words, the TM could be considered non-mechanistic, arguing that it does not provide a complete causal blueprint. Nevertheless, this does not prevent the TM from being a mechanistic abstraction. In a 1950 paper, Turing defines it as a writing mechanism (Turing, 1950) (more comments on that aspect can be found in (Lassègue and Longo, 2012)). The TM is not a full-blown description mechanism but a mechanistic sketch.

The way the tape head works is abstracted away. But the TM still presents minimal components of a functional architecture: the tape, the tape head, and the state register. Each of the components has some properties. The tape is of infinite length and divided into cells; the tape head can read three types of symbols (0, 1, "blank"), erase them, and replace them with another symbol (0 or 1). The state register contains n number of states. The relationships between the components and how computation can be executed on such a functional architecture are described following an instruction table. The TM's instruction table is a basis for describing in mechanistic terms the transition from one cell to another. It contains a transition instruction for each state, defining the action to be executed (reading/writing action), a move to the right or the left of the cell, and a new state to be entered. Similar concerns are developed by Bozşahin (Bozşahin, 2018):

> The best theory to date for computability, that of Turing (1936), is an abstract mathematical object in the form of an automaton, and even in the abstract form it is physically realizable because it has primitives which are not reduced to other operations, and whose terms are quite simple and clear: move left, move right, change state, read, and write.

As already pointed out, the physical implementation of a Turing Machine is not straightforward. It requires the addition of physical devices, e.g., a clock, to control the tape head. The fact that implementing the TM abstraction requires additional care about implementation details does not make the TM abstraction non-mechanistic. A complete blueprint is not necessary to build a mechanistic abstraction. Flight mechanics would not tell how to build an airplane from A to Z, and still be an accurate mechanistic description of how a plane comes to fly. The point is that just because a physically implemented TM does not match the TM abstraction, it does not prevent the TM abstraction and its realization from being mechanistically equivalent. Given the mechanistic description of the abstract TM, one can make sense of the execution behavior of a physical TM.

On the contrary, other models of computation, like lambda calculus, do not reference a functional architecture and a mechanistic description of the execution. The singularity of the TM, among other formalisms, has already been pointed out:

> Showing the step-by-step progress of a function's concrete state in excruciating detail was uncommon. We were used to the global views of Frege and the lambda calculus of Church. It led to an understanding of quite complex tasks, and crucially, at the same time showing transparently that it happens without a concomitant increase or complication in the internal mechanism. (Bozşahin, 2018)

2 Understanding interactive computers qua models of computations: to what extent?

For historical and conceptual reasons, the Universal Turing machine (UTM)[2] has been the cornerstone of most work on the epistemology of computers (Mol, 2018). It is not surprising since the UTM is not only a model of computation but also describes a mechanism and suggests minimal components for functional architecture. In this section, we briefly remind readers of the connection between the TM and the epistemology of computers. Then, we consider the limitations of the Turing machine to explain the essential properties of interactive computing systems. We add that the minimal mechanistic description provided by the TM for Church-Turing computation needs a clear-cut and identified counterpart when explaining interactive computation. That does not mean that no abstraction in computer science can support such an explanation: it means only that there is still work left to identify the relevant abstractions and ingredients to build a minimal mechanistic description of interactive computation. We think such a minimal description dedicated to interaction is meaningful to the epistemology of computing.

2.1 The Universal Turing Machine as a model for the digital computer

Mechanistic explanations have become the scientific standard to account for phenomena Machamer et al. (2000); Craver (2001); Glennan (2002). Digital computers are no exception, and the straightforward way to address what they do is to provide a mechanistic description of the relevant computing mechanisms at stake. There are many levels of abstraction to describe what a digital computer is. For example, one could focus on the description of what is going on in terms of voltage firing from the point of view of an electronic engineer. Or one could focus on the transformations in memory registers or describe the program running on the machine. The complexity of every layer makes it difficult to articulate a comprehensive explanation of how the system works, paying tribute to every single layer.

Therefore, from an epistemic point of view, understanding computers is challenging: what is the proper explanatory focus? It is a well-known issue for any mechanistic explanation: "it is sometimes possible to decompose a system at high or too low a level and miss the level at which interactions transpire that are crucial to account for the phenomenon in question."(Bechtel, 1994). Engineers can flesh out the way each layer functions and how the layers are related to each other (Nisan and Schocken, 2021; Lee, 2018). However, they admit it is impossible to provide a detailed overview of a computing system within

[2] A Universal Turing Machine is a special Turing Machine that can simulate any other Turing machine, hence its name.

a single abstraction. Nevertheless, this does not mean that epistemologists have to give up the search for a single explanatory abstraction. It only shows that any explanatory abstraction for a computing system is necessarily a trade-off: it cannot be exhaustive in detail and understandable by a human.

Traditionally, philosophers have been describing the computing mechanisms in a computer in terms of the manipulation of digits. According to Piccinini (Piccinini, 2007, 2008b), for example, a computing system is mechanistically described through its capacity to generate output strings of digits from input strings of digits and (possibly) internal states following a general rule.

Such an approach finds tools and models for a mechanistic explanation within the frame of computability theory. State automata, and among them the UTM, match the targeted level of explanation: it is the very job of a state automaton to model the transitions from one state to another, following rules step by step. Therefore, it is not surprising that the epistemology of digital computers, computing mechanisms, and computability theory have become related to each other in the philosophy of computing. That situation is also a reminder of the peculiar status of the Turing machine in computability theory, among other models of computation. Therefore, philosophers have tended to focus on the Turing machine among all models of computation. Today, the UTM is still influential in philosophy to think of the modern computer (Mol, 2018).

However, computer scientists have already argued that such a view is mistaken (MacLennan, 2003; Lee, 2020): "(...) computers can do things that a universal Turing machine cannot. Many applications, including Wikipedia and Google search, are designed never to terminate and are interactive" ((Lee, 2020), chapter 8). Of course, other models have been introduced since to account for the new properties of digital computers. However, as we will see in the coming subsections, they are mere models of computation and cannot serve as execution models.

Since the time that the UTM first served as a model to understand what computers are, computing practices have evolved. A legitimate question is then whether the UTM pays justice to current practices and whether new formalisms dedicated to interaction are able to support an explanation.

2.2 Interactive computing systems properties and the limits of the UTM model

A review of the literature shows that interactive computing systems have posed several challenges to computer science researchers. They have led to the introduction of new formalisms to model new properties. Among those properties, some have been particularly commented on in explicit theories of interaction presented in the following subsection: (i) the arrival of external input data and ongoing execution, (ii) concurrency and synchronization between processes, and (iii) time.

That does not mean these three properties alone embrace what interaction is. In particular, interaction cannot be reduced to concurrency. In any case, the mentioned formalisms do not target an account of the mechanisms supporting interactive execution. Other aspects like dynamicity (the possibility to have components added or deleted during execution) (Attie and Lynch, 2003; Navarre et al., 2009; Arbach et al., 2015), or the importance of feedback loops where outputs can affect future inputs as exemplified in cyber-physical systems (Lee and Varaiya, 2003), have less been commented on within theories of interaction but should play a role in our account.

2.2.1 External input data and ongoing execution

As said before, computers are increasingly interactive. They are no longer transformational systems producing a final output after a finite execution. Instead, they continuously react in time to external events that modify the course of execution.

In a UTM, all data is given before the execution starts and there is no means neither to change the course of execution once the machine is launched by providing new symbols that were not initially provided, nor to have outputs affecting future inputs. Turing had already thought about formalizing the arrival of external input data provided during an ongoing execution: he introduced such an abstraction in his Ph.D. dissertation (Turing, 1939) and coined it the "oracle machine". This concept was expanded later by Post (Post, 1948)[3].

When he invented this concept, interaction was not a concern for Turing. The purpose of the oracle machine was to address the issue of undecidability in computability theory. Nevertheless, Turing's work has later inspired some derived formalism on "extended" Turing machines, such as the "Reactive Turing machine" (Andersen et al., 1997; Baeten et al., 2013; Luttik and Yang, 2016) or the "Persistent Turing machines" (Goldin, 2000). This later work can be seen as a way to account for interaction in the classical framework. However, they fail to account for an essential aspect of interaction, namely the *inversion of control*. With an oracle machine, the machine drives the execution and takes the initiative of requesting an answer from an external Oracle. Conversely, truly interactive systems are systems that react to incoming events: the occurrence of external events drives the execution.

Moreover, we still need the mechanistic description of how such an external action on the tape is made possible. The oracle machines cannot explain it, as their job is not to explain how the oracle interacts with the tape: "they cannot be asked to justify the causal/physical chain of their steps" (Bozşahin, 2018). For an interactive computing device to get external data, one needs to account for the capacity of the system to wait, stop, and resume some processes upon data arrival. Data arrival would be useless if it could not trigger something within the system.

In addition, the external inputs that arrive during execution are often information about external physical processes. An additional mechanism is necessary to translate a physical magnitude into digitalized data. This operation is called transduction. The importance of transducers has already been mentioned in theories of interactive computing (MacLennan, 2003; Leeuwen and Wiedermann, 2006). By modeling input as a sequence of symbols, the TM or any extended version does not give insight into the variety of dimensions of physical phenomena that can trigger computational processes, such as their spatial organization, their duration or frequency. The very goal of the TM is to provide an answer to a computation, e.g., whether a number is computable or not. Therefore, there are no dimension of the inputs and outputs.

2.2.2 Concurrency and synchronization

During the 1960s, the technological evolution of computers brought about some important new features, notably the interaction of numerous computing processes running in parallel and communicating. But, in a classical UTM, it is the step-by-step execution of a given procedure that is described, and not the synchronization of message passing. Consequently, a concurrency theory emerged from Dick Karp's early work in the 1960s, grew with Petri's work (Petri, 1980) and has now developed into a mature theory of

[3]See, e.g., Soare's work (Soare, 2009, 2013)

reactive systems with diverse network models (for an overview, see (Lee and Sangiovanni-Vincentelli, 1998; Lee and Neuendorffer, 2006)). In the context of modeling concurrency (Milner, 1975, 1982, 1983), the conceptualization of *interactive systems*, as opposed to *computational systems*, emerged in computer science, as formulated by Milner (Milner, 1993, 1999, 2006). "Interaction" here refers to concurrent message passing between agents in distributed systems.

First, it installed the notion of a transition system as the prime mathematical model to represent discrete behavior (Nielsen et al., 1981; Baldan et al., 2001; Glabbeek and Plotkin, 2004; Arbach et al., 2015). Second, it showed that language equivalence was not the correct notion when comparing automata for interactive systems. Instead, it should be replaced by a notion of behavioral equivalence or bisimilarity (Milner, 1999). Third, it yielded many algebraic process calculi facilitating the formal specification and verification of reactive systems.

2.2.3 Time

Time is not a concern in classical automata theory. The Turing machine or any derived abstract machine specifies how to go from the input to the output in a finite number of steps. However, the time it takes to move from one step to another is of no concern. At most, one may be interested in the number of steps from the input to the output. When specifying interactive behavior, a richer notion of time is required: time as duration or physical time, measured in a physical unit. Duration is essential for several reasons.

Many interactive systems interact with humans, and human perception is also sensitive to duration. For example, a written message must be displayed for a minimum duration to be readable by a human.

Similarly, for some interactions between a computer system and a physical environment, time is a crucial element. An autonomous vehicle, for example, must react to changes in its environment in a correct and timely manner, i.e. not too slowly but not too quickly either.

The problem with duration is that it cannot be specified by a reference to steps since we do not know the duration of a single step. The only way to specify a duration is by a reference to a physical process whose duration is known, such as the period of a crystal oscillator. Some reference to a physical clock is thus needed to explain how a timing constraint can occur in a computer. Timing constraints on real-time systems have notoriously posed a challenge, which has been addressed and solved. Diverse timed automata have served as formalization tools (Reisig, 1988; Alur and Henzinger, 1993; Alur and Dill, 1994; Lynch et al., 1996; Segala et al., 1994). Once again, a timed automaton is not an execution model: it does not carry an abstract reference to the mechanism that ensures the timed transition between states.

2.3 Explicit theories of interaction discussing the TM

Faced with the need for new properties to model interactive systems, some explicit theories of interaction discussing the scope of the Turing machine have emerged. Questioning the theoretical bounds of the Turing machine in computer science when faced with the existence of interactive devices has been explored at least since Milner's work on communicating and mobile systems (Milner, 1993, 1999). To our knowledge, there are three areas in which interaction models have been developed. In all of them, the comparison between TMs, oracle machines, and interactive system models is systematically at stake. These

areas are works on concurrency by Milner and his followers, on Reactive Turing machines, and on interaction as a new computing paradigm.

We argue that these formal approaches cannot provide an answer to the epistemologist for two reasons. On the one hand, these models of computation have focused their attention on whether interactive models are reducible to models of classical computation - par excellence, the Turing machine. Proving (or not) that an interactive property can be formalized as a computational property in the classical Turing sense does not answer the question of how an interactive property can exist and be the object of execution. On the other hand, and this is a correlate, these models do not propose a basis for a mechanistic explanation of the very possibility of an interactive computer system.

Milner introduced a distinction between *interactive behavior* and *computational behavior* in computer science. His Turing Award speech (Milner, 1993) summarizes his motivations. Milner was concerned with the logical foundations of computing inherited from Turing. He was preoccupied with the idea that computing practices had evolved since the birth of computing, notably in terms of architecture. Milner's work does not provide us here with what we are looking for. The way he understands interaction is delimited by concerns about concurrency, and a mechanistic account is out of scope.

A more recent literature domain proposes extending the Turing machine to account for interactive computing systems. It may be traced back at least to seminal works on a "Universal reactive machine" (Andersen et al., 1997). In that respect, although pointing at the specificity of interactional behavior, the main framework still relates to Turing's. Baeten (Baeten et al., 2013) is looking for a computational model of interaction, extending the classical TM with a process-theoretical notion of interaction related to Milner's previous work. The aim is to formalize the arrival of input data during execution by extending the original TM with an oracle. Such theory of interaction frames the questions in terms of relationships between models of interactive computation and implications for the Church-Turing thesis (Leeuwen and Wiedermann, 2001). As we explained previously, these formal issues cannot provide us with the level of abstraction we need to build a mechanistic explanation.

Wegner introduced interaction as a new paradigm (Wegner, 1997), based on an empiricist approach (Wegner, 1995), to broaden algorithmic problem-solving (Eberbach et al., 2004; Goldin et al., 2006). The main objection made by other researchers to Wegner's work is that interaction machines can be proven equivalent to TMs. The objections are focused on the defense of the Church-Turing thesis (Prasse and Rittgen, 1998; Cockshott and Michaelson, 2007), assuming that interactive modeling is a way of denying the results of Church's and Turing's work. But focusing the definition of interactive computing around reducibility to the Turing machine's expressiveness is a formal debate. It does not support an explanation of the relevant phenomena that make interactive computing possible.

2.4 The engineering of interactive systems

Interaction has also been addressed outside explicit theories of interactive computation and has been a major concern for engineers and programmers, with the programming of a broad class of "reactive systems". A classical definition can be found in Boussinot's work [4]:

[4]In addition to that definition, a further distinction is introduced in the reactive programming community: reactive systems are sometimes distinguished from interactive systems (Harel and Pnueli, 1985; Mandel and Pouzet, 2005). A distinction is made around the real-time dimension of these systems. An *interactive system* reacts to events in the environment without time constraints, whereas *reactive systems*

> Reactive systems have been defined by Harel and Pnueli as systems that are supposed to maintain an ongoing relationship with their environment. Such systems do not lend themselves naturally to the description of functions and transformations: they cannot be described adequately as computing a function from an initial state to a terminal state. On the contrary, behaviors of reactive systems are better seen as reactions to external stimuli. The role of reactive systems is to react continuously to external inputs by producing outputs. For example, man-machine interface handlers or computer games fall into the category of reactive systems. (Boussinot, 1991).

It has led to a whole new sets of programming languages (Hewitt and Baker, 1978; Caspi et al., 1987; Berry and Gonthier, 1992; Elliott and Hudak, 1997; Agha and Hewitt, 1988; Czaplicki and Chong, 2013; Vonder et al., 2017; Berry and Gonthier, 1992; Berry and Serrano, 2020), and proposals for a new "reactive" programming paradigm (Bainomugisha et al., 2013; Salvaneschi et al., 2015; Bonér et al., 2014).

The literature presents requirements to program and understand interactive systems (Suchman, 1987; Myers, 1994; Ko and Myers, 2004; Victor, 2012; Schmidt and Bansler, 2016; Hornbaek and Oulasvirta, 2017; Basman et al., 2018), studies the specific challenges encountered by programmers (Hill, 1986; Myers, 1991; Myers et al., 1994; Myers, 1994; Myers et al., 2000; Casiez and Roussel, 2011; Bainomugisha et al., 2013; Salvaneschi et al., 2015) and introduces diverse models to help programmers reason about their programs (Dearden and Harrison, 1997; Campos and Harrison, 1997; Navarre et al., 2006, 2009; Canny et al., 2019). However, some HCI researchers note that the appropriate level of abstraction to account for interactive systems is still missing:

> We argue that these models tend to be either: so abstract as to limit their ability to express important interaction concerns for specific systems, and limited in the degree to which they support the construction of software that conforms to the designer's intention; or so specific to an individual system that they provide only limited re-use across development projects and are, therefore, likely to be too expensive to develop except in a few special applications, such as critical safety systems. We argue that it is possible to construct a generic model of a class of interactive systems at an intermediate level of abstraction. (Dearden and Harrison, 1997)

The literature in computability theory shows that researchers have addressed the specifics of interactive computing systems. It is now stated that interactive systems differ from "transformational systems". On the one hand, in order to tackle this issue, new models of computation are proposed to account, e.g., for incoming data during ongoing execution, concurrency, and time. It leaves aside interesting properties that define interactivity, such as the dynamicity of the execution mentioned in our preliminary example. Furthermore, these models are purely formal and say nothing about the mechanisms allowing the realization of these properties. On the other hand, programmers have tools to build interactive systems, but they need the right level of abstraction to fully understand them.

---

react within a time limit set by the environment. For example, the kernel of a general-purpose operating system (OS) is interactive (its response time to events depends on its load and hardware capabilities). In contrast, the autopilot of an aircraft is a reactive system (its response time to events is specified and must be respected). When we talk about interaction from the perspective of HCI, we refer to "interactive system" following that distinction.

Thus, while the TM provided both a model of computation and a mechanistic description serving as a theoretical basis, there is nothing equivalent dedicated to interactive computing systems. We are missing what we propose to call an *execution model* for interaction.

3 Proposal

We argue that a general mechanistic explanation of interaction should account for the following:

- how the internal computing processes can be triggered, paused, and relaunched by external events; in other words, how **causal relationships** between processes as specified by the programmers can hold;
- what allows physical phenomena to interact with computing processes; in other words, what role **transduction** plays;
- how the internal computing processes can be organized in time and respond to delay instructions, which supposes a reference to **physical time**.

3.1 Minimal requirements

3.1.1 Expressing causal relationships

Specifying an interactive behavior of a programmable machine relies on the description of relations between the occurrence of events and the triggering of various processes, possibly computational, within the machine. The causal requirement for interactive devices is notably described by Myers in the following term (Myers et al., 1995):

> Any large, complex application contains thousands of interdependent relationships. For example, a graphical application must deal with the relationships arising from laying out objects, displaying feedback for input operations, and keeping the view consistent with the underlying data they represent. (...) Constraints provide a convenient way to specify relationships and have them automatically maintained at runtime by a constraint solver.

The causal vocabulary is pervasive in the literature when describing an interactive system: (Myers et al., 1994; Jacob et al., 1999; Ko and Myers, 2004; Myers, 2013; Leveson, 2020). The possibility of such causal relationships between processes is the mechanism to explain. The need for such an explanation has been pointed out in the literature, for example, in the "Anatomy of Interaction": "We believe that the major fault of current approaches to programming interactions is that they do not account for **how interactions come to be**" (Basman et al., 2018), or in the Reactive TM community: "In order to mimic site machines, a Turing machine must have **a mechanism that will enable it to model the change of hardware or software by an operating agent**" (Leeuwen and Wiedermann, 2001) (see Subsection 2.3).

The programmer often needs to specify when to stop or restart processes. This is why an interactive abstract machine should have basic mechanisms that allow controlling the life of a process from the occurrence of triggering events. The concept of "event" is understood here in the general sense of something that happens. This can be an event internal to the computer system (such as the end of a computation process) or external (such as a mouse click).

This dimension is absent from the Turing machine: there is no way to describe the launching of the execution in reaction to events. It makes no sense to ask when the tape head starts reading and rewriting a box. For the Turing machine, everything is assumed to be given before the start of the execution; the inputs are already specified. It is the succession of computation steps and the final result on the tape that matter. Similarly, the classical model does not allow to express the pausing or restarting of the automaton. This does not make sense for a computational algorithm. One could object that the Turing machine with oracle models the pausing and the continuation of the calculations. There is, however, a major difficulty with this extension. Indeed, when we talk about interaction, we do not describe an automaton that, at a stage of its execution, would ask an oracle (or any other abstract representation of an external agent) for a new symbol. What we want to specify is that an external process can interrupt or launch a machine process. In other words, interaction presupposes an *inversion of control*: it is no longer a Turing machine that controls the course of its execution but rather the external environment that controls the flow of execution. This requirement has been discussed in the literature. In the Reactive Turing machine community (Leeuwen and Wiedermann, 2001) (See Subsection 2.3), that aspect is also put at the forefront, but without an account of the mechanism allowing the system to take the arrival of new data into account. Remember that in the formal account proposed by the Reactive Turing Community, the oracle is the abstract concept supposed to ensure the arrival of new data. But an oracle is not a mechanism. An oracle machine is an infinite table lookup that formalizes the solving of undecidable problems.

High-level mechanisms, such as event loops, wait continuously for new inputs. In current computer architectures, there are mechanisms for acknowledging the arrival of new data during execution. In many processors, they are interrupt mechanisms, launching, pausing and resuming processes upon arrival of new data — but they can take other forms. When an interrupt is requested, the running process is suspended, some information is saved, and a pre-defined code called a routine is executed. For example, moving the mouse or pressing a key on the keyboard causes an interrupt, which in turn calls a routine. These interrupt handlers allow the reading of the mouse position or the value of the pressed key. What has been read is then copied into memory. At a low-level in memory, the connection between the external input arrivals and the computing system may correspond to changes in specific memory registers. Memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) are two complementary methods of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer.

These mechanisms can take other forms than those found in current architectures. An alternative to interrupts is *polling*: with polling, the CPU steadily checks whether an I/O device requires something to be processed.

Whether it is polling or interrupts, the fact is that in their absence, any form of interaction with arriving data is impossible. Without such a mechanism, it would leave unsaid how the change of symbols on the cell of the tape can involve any change in the system. Thus, at the level of the abstract model, this type of mechanism appears as a necessity.

3.1.2 Referring to transduced data

Interrupt management mechanisms *or any mechanism that would ensure causality management* between external and internal processes are insufficient to ensure the link between the physical world and the computational processes of the machine. An additional mechanism is required to convert physical quantities into digital data: transduction. The

importance of transduction for interaction computing has already been commented on in theoretical computer science (Leeuwen and Wiedermann, 2006), the epistemology of computing (MacLennan, 2003), and HCI (Chatty, 1994; Accot et al., 1997; Navarre et al., 2009; Basman et al., 2018; Canny et al., 2019): "For computers and software to become mediators of human action, they need to be able to respond to the outside world. This is broadly the purpose of interactions as implemented in code: to transduce changes at some locations to changes at other locations" (Basman et al., 2018).

It has also been argued that a low-level model for transduction is crucial for designing fine-grained interactive systems (Chatty, 1994; Accot et al., 1997). Some inputs can be described by a simple Boolean value, indicating the presence or absence of a signal (such as a mouse click). Others can have a more complex structure, such as the continuous evolution of a physical magnitude (light or temperature, for example). A fine-grained interactive system involves the programmer can easily connect and disconnect transduction sources (Basman et al., 2018).

The objective of a Turing machine was to give a result to a calculation. The size and shape of the inputs were of no importance, and the tape presented information in 2D. The Turing machine does not allow accounting for the variety for the variety of physical phenomena that can cause computational processes. However, when the objective is to specify causal relationships between physical and computational processes, it is important to express the structural and temporal properties of physical processes. This means that a transducer is not only the digitization of an analog signal; it must also preserve and transmit the *causal* structure of the physical phenomenon (Accot et al., 1997). We talk about "causal" structure to refer to many transduced physical phenomena, where the programmer needs to consider the periodicity of phenomena. For example, refreshing a frame or adjusting video frame rates requires that the programmer calibrate the reception of data.

In HCI, the challenge of transmitting information is increased with the growing number of input devices and multimodal interaction (Navarre et al., 2006) between users and computers : e.g.,"hands-discrete inputs", "hands-continuous inputs" (Jacob, 1996), other body movements (like head position or direction of gaze), voice, virtual reality inputs (Jacob, 1996).

3.1.3 Expressing measurement of physical time

In the classical theory of automata, physical time is ignored. There is only a notion of logical time reduced to order. Thus, the Turing machine or any derived abstract machine allows specifying a sequence from an input to an output step by step. It is not the physical time but the number of computational steps that is relevant. The physical time it takes to execute a step (i.e., a step in the computation) does not exist for the model (Longo, 1999). In the field of distributed systems, this lack of a physical notion of time has proven to be problematic. The solution has been to reduce the notion of time to a notion of order by using logical clocks or timestamps (Lamport, 1978). However, programming interactive behaviors requires a richer notion of time: a notion of physical time, i.e. duration, measured by a physical unit. A gesture-based (post-WIMP) drawing tool or any interaction technique cannot be implemented without describing timing aspects to represent the quantitative temporal evolution of the interaction technique (Navarre et al., 2009; Canny et al., 2019).

Furthermore, the systems we are talking about interact with humans and involve human perception, which is sensitive to duration. The display of a message, for example, cannot be too brief and must appear long enough to be readable by a human agent. Hence

the need for the programmer to be able to express a duration, here to specify the time after which the process can be interrupted.

The problem with duration is that it cannot be reduced to a number of steps or stages in a computation: we need to know how long a stage takes, and we cannot assume that each computation stage can be executed in an equal time. The only way to specify the duration is to refer to a physical process that can quantify a duration: this is the case of the frequency of an oscillator. At the level of an abstract machine dedicated to interaction, this implies including a reference to a physical clock, which allows specifying a duration. It is true, however, that there are interactive devices that do not require clocks (there are clockless architectures), but we argue that there are rather a subset of interactive devices rather than the general case. Most embedded systems interact with humans and require some notion of delay.

3.2 Components-candidates for a mechanistic sketch

To build a mechanistic interactive execution model, we need to identify the necessary components of the mechanism, their relationships, and their properties. In the following, we define three minimal components derived from the previous requirements to account for the previously mentioned relevant phenomena. We describe, for each component, its properties and relationships with other components.

1. **A source component**. It is enough to say that an execution model must have a set of sources at a higher level (sources referring here both to inputs and outputs). Each causal source is modeled as a set of causal relationships. Sources can be of four different natures: (i) generated by internal software processes (e.g., the assignment of a result in a property), (ii) generated by transduction phenomena (Analog-to-Digital and Digital-to-Analog conversions), (iii) external agents (e.g., user's log-in request) and (iv) generated by the internal expirations of timers. Such a component is made explicit in some views of interaction, supported by toolkits for programmers (Dragicevic and Fekete, 2004; Huot et al., 2004).

   Properties. Causal sources have a structure in time and space that they can transmit to the causality engine. A formal description of a causal source should provide a model for the organization of its parts and characterization of its associated magnitudes. A mouse, for example, can be described as the composition of a 2D signal continuously sending displacement quantities, with a set of occasional valued signals for the buttons.

   A fine-grained theory of interactive devices should provide an ontology of the varieties of causal sources (Chatty, 1994; Jacob, 1996; Accot et al., 1997).

   Relationships. Sources are connected directly to the causality orchestrator (the second component, presented below), which ensures the right reactions are triggered by the input arrival. Signals sent by the expiration of timers are among the possible causal sources, which makes it possible to express duration in interactive computing systems.

2. **A causality orchestrator component**. When programming interactive behaviors, a requirement is that the entire system responds in a deterministic way to the unpredictable arrival of events. That is, the interactive machine must ensure that the order of execution complies with the causal relationships specified by the program. This component is reminiscent of what is labeled as "constraint solver" in

Garnet (Myers et al., 1995). We argue that the role of such a component is hardly modeled by the behavior of the tape head of the TM. In a classic Von Neumann architecture, it is the role of the program counter to ensure the right ordering of a specified sequence of instructions. For an interactive machine, the abstract component that warrants the right causal ordering should be conceptualized as such, and we suggest labeling it as the "causality orchestrator". An instantiation of a causality orchestrator is present in some software toolkits to support the implementation of interactive applications. For example, the Qt toolkit stores signal/slot dependencies between interacting components and implements the cascade of triggers resulting from the occurrence of events. Older, seminal toolkits Williams (1984), such as MacApp Wilson et al. (1990), also implement the distribution of events among object-oriented components. NewtonScript went as far as implementing an event dispatch mechanism according to both the prototype relationships and the containment relationships between graphical components Smith (1995). The operating system (OS) can also be considered a causal orchestrator. The Portable Operating System Interface (POSIX) exemplifies a way to make available to the programmer the managing of the causal engine. POSIX is a family of standards defining both the system- and user-level application programming interfaces (API), along with command line shells and utility interfaces, for software compatibility (portability) with variants of Unix and other operating systems. The core services provided by POSIX could serve as detailed examples of the kind of instructions required for a causal engine: e.g., process creation and control, signals (that send messages to a running program to trigger specific behavior, such as quitting or error handling), I/O Port Interface and Control, semaphores, thread creation, control, cleanup, scheduling, synchronization.

Properties. The causality orchestrator can start, stop, or resume processes, following causal instructions and establishing the right connections between causal sources and connected processes. It should also provide flexibility to allow the composition of new connections, e.g., between new input devices and new interaction techniques.

Relationships. The causality orchestrator receives the structure of the sources. Among sources are time values transmitted by the clock component. The causality engine makes sure a source triggers the wanted processes, which may be computational (e.g., re-calculation of a value) or physical (e.g., launching a new timer).

3. **A clock component**. When one designs an interactive computing system, one needs to specify timing instructions. By "time", we refer to the physical time or an elapse of time, not to order. Therefore, a specific representation of time is required in the model.

   Properties. The clock component should provide both a timer facility, i.e., a component that fires an event when a scheduled duration is expired, and a clock facility, i.e., a component that sends a periodic signal. It is difficult to determine which component is the most fundamental. On the one hand, one can build a clock by establishing a causal relationship between the end of a timer and its restarting. On the other hand, one can build a timer from a clock by establishing a causal relationship between a specified number of clock ticks and the stopping of the timer. However, at the hardware level, the most basic component enabling the measurement of time is an oscillator.

   Relationships. The clock component feeds the source component (the expiration of a timer becomes a source).

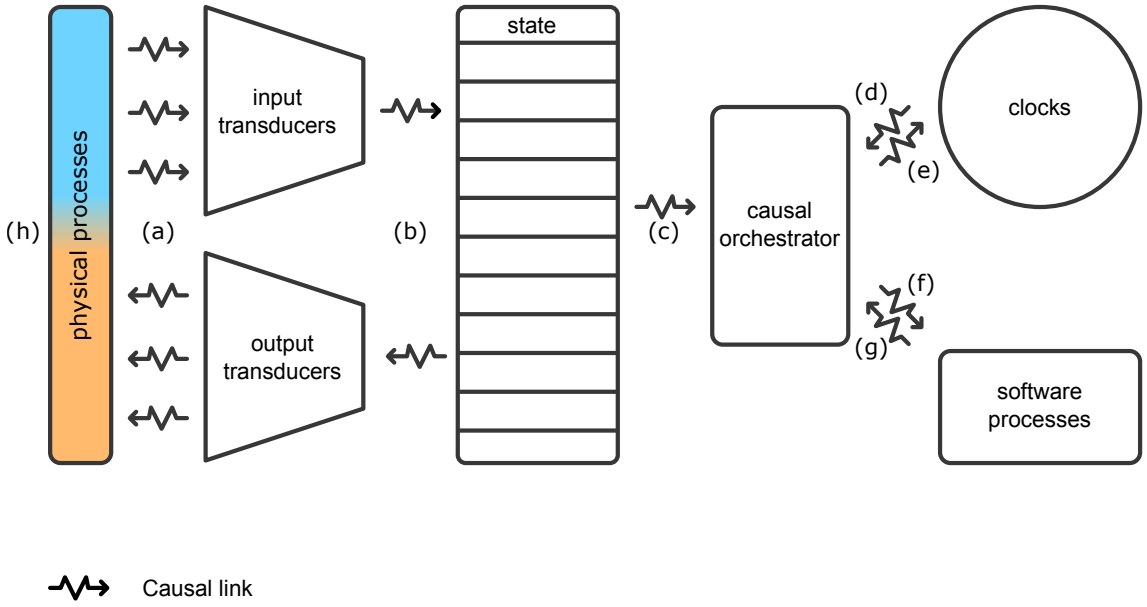

Figure 2: Representing a general interactive execution model

In Figure 2, we propose a representation of our execution model. We add extra comments to detail the links between each element.

First of all, to avoid confusion, the arrows in the figure specify causal links, not dataflows. As such, they specify meta-causal relationships: they are not the causal relationships managed by the causal orchestrator but rather the causal relationships that implement the mechanisms of the machine.

(a) Transducers are physical causal links that ensure the translation of physical processes into digital inputs. **(b)** The arrival of new transduced data changes the state of the machine. **(c)** As commented previously, the machine may react to the reception of new data and trigger the causal orchestrator. The causal orchestrator then ensures that the causal relationships specified by the programmer hold. The causal orchestrator keeps count of the causal relationships specified by the programmer and guarantees their ordered, unique execution **(f)**. In turn, the software processes may trigger other causal links through the causal orchestrator **(g)**. Since some relationships depend on timing constraints, the causal orchestrator activates clocks and timers **(d)** and responds to their ticking **(e)**. As an interactive execution is a feedback loop involving a user, transformational processes are, in turn, transformed **(b)** into physical outputs available to the user through transduction **(a)**. Inputs **(h, orange)** may trigger changes of outputs **(h, blue)**, and outputs possibly affect inputs during the ongoing execution **(h, orange-blue)**, implementing the interaction between the user and the machine.

A few additional comments can be made:

- **(c)**: In some concrete implementations, the machine mechanisms may rely here on interrupts.
- **(c)(e)**: Alternatively to interrupts, another mechanism like polling may ensure the causal link between the reception of data and the causal orchestrator: a clock steadily makes the causal orchestrator poll the state and trigger an execution cycle.
- **(h)**: Outputs and inputs are blended within a single block along a colored spectrum, to represent the fact that inputs are the source of outputs but that outputs in turn

can affect future inputs. Examples include the apparition or deletion of invisible picking zones implementing a transient spatial mode or movement constraints for a phantom haptic device.

3.3 Refining the causality orchestrator with dynamicity concerns

A typology refinement could describe further the dynamicity of causal relationships. Three levels can be identified:

- **Level 0. Causal relationships are static**. Causality is expressed in the following general fashion: "any update of the value of x, updates the value of any entity connected to x". Static here means that the network of connections between entities does not change during execution; thus, the causal path is immutable.

- **Level 1. Causal relationships can be modified at runtime**. This type of dynamicity basically refers to the possibility of switching from one causal path to another one during execution (some are activated, others are deactivated). In the context of a set of causally connected processes, this means that the execution of one process will result in a change in the causal connection between some processes. Some causal links are (de-)activated during execution. For this class of dynamicity, the processes and the topology of their causal links are defined at the beginning of the execution, and the dynamicity comes with the possibility to (de-)activate causal paths during execution.

- **Level 2. Causal relationships can be created or deleted during execution.**

  The third and most powerful type of dynamicity allows a dynamic change in the causal structure, that is, adding or deleting components and causal paths during the execution of the program. Such an evolutive causal structure appears necessary in many applications where one wants to react to the appearance/disappearance of external processes. A sublevel could be introduced within level 2; let us call it **level 2+**. This would allow taking into account a type of dynamicity required when a programmer wants a defined causal relationship to apply to newly created objects. In that case, it is not the causal relationship per se that changes, but only the *relata* that are modified. In other words, it is not the meaning of the arrow in $A \rightarrow B$ that changes (activation, deactivation, adding, deleting) but A and B that change, replaced by a dynamic reference to newly created processes C and D.

4 Positing the concept of execution model among kinds of computational explanations

Thinking about and proposing relevant levels of description for a computational phenomenon and its implementation is far from being a novelty, especially among accounts of concrete computation in cognitive science (Fresco, 2010; Shagrir, 2012). In this section, we position our level of analysis among others that have been classically proposed in the literature to describe computers or computational systems in the broad sense (natural or artificial systems that are assumed to process information like a computer). The question we want to answer (what is a computer today?) implies a well-known epistemological questioning: given that a computational system is both a physical system describable in

physical terms, and the implementation of a given abstraction, to which level(s) of description must we refer to produce a satisfactory explanation of the phenomenon? The difficulty is to agree on a set of criterion of satisfiability of the explanation.

We believe that at least three criteria are proposed in the literature and define at least three types of approaches for a computational explanation, motivating the choice of the relevant level(s) of description: (i) algorithmic satisfiability, (ii) functional satisfiability, (iii) causal satisfiability. Our execution model aims to satisfy both functional and causal satisfiability.

4.1 What an execution model does not satisfy: an algorithmically-focused explanation

Algorithmic satisfiability is typically represented by Marr's framework of analysis [5] (Marr, 2010) where the explanation is broken down into three levels of analysis. In the middle is the level of algorithmic description, the intermediate between the description of the computational level (the computation that one wants to explain) and its physical implementation. In this framework, a good explanation is essentially based on the description of the algorithm, which is given a crucial place to operate the junction between the computational level and the implementation.

An execution model articulates the computational level with an intermediary level between Marr's algorithmic and implementation levels. That intermediary level is twofold: it adds a layer of mechanistic description to a functional architecture. It is less abstract than the algorithmic level in that it tells something about how computation can be executed. But it is more abstract than the implementation level because a functional architecture is not a complete specification of the hardware that implements it. It identifies functions than can be carried out by different hardware pieces depending on processor types or technology maturity.

4.2 What an execution model satisfies

4.2.1 Functional satisfiability through a functional architecture

Another approach consists in looking for a satisfying functional explanation. It focuses on identifying the functionalities of the component parts of the computational phenomenon analyzed as a mechanism, such as Piccinini's computational account (Piccinini, 2007). In that case, the focus switches from an algorithmic analysis to identifying the relevant component parts and the assignment of functions to them. Our execution model satisfies such a functional approach. The components of our execution model are singled out by their function and serve to identify a functional architecture à la Pylyshyn (Pylyshyn, 1986). Regarding the level of analysis, the functional architecture, which is the basis for an execution model, is the description of the blueprint but without reference to concrete pieces of hardware. Functional architecture is distinct from the more concrete notion of "architecture" that can be found in the philosophy of computing (Bozşahin, 2018), in which case "architecture" refers to some real hardware pieces. A transmission system in a car, for example, is a function that we can define as the transmission of energy from the engine to the wheels, but which can be realized by different kinds of hardware. An execution is supported by a functional architecture but not reducible to it. As the label

[5] Marr's work on visual perception, e.g., presented in his book, *Vision: A Computational Approach*, has been influential in analyzing complex information processing systems. See McClamrock's paper (McClamrock, 1990) for a synthesis of Marr's framework and criticisms.

coins it, an execution model is about the description of execution, not merely architecture. With only a functional architecture at hand, one cannot describe the very mechanism by which computation is carried out. It requires an additional description of the interaction between components identified as part of the functional architecture.

4.2.2 Causal satisfiability through a mechanistic description

A third criterion, not incompatible with the functional one, has motivated other types of analysis of computational phenomena and the search for complementary levels of description. This criterion is that of a satisfactory causal explanation. The idea common to these approaches is distinguishing between formal computational models and computational mechanistic explanations, arguing that only the latter can provide a causal description of the explanandum. In several papers, Miłkowski defends such a view about the non-mapping between a model of computation and a mechanistically adequate model (Miłkowski, 2014, 2016). The point is that causal organization at the implementation level cannot be told by a formal model of computation. As he explains through an illuminating example: "to repair an old broken laptop, it is not enough to know that it was (idealizing somewhat) formally equivalent to a universal Turing machine." (Miłkowski, 2016). Thus, it is necessary to define abstract levels of description closer to the implementation level to account for non-calculative phenomena which are also relevant to the production of computational behavior. Similarly, with our execution model, we intend to provide a level of description that accounts for the very possibility of interactive programs.

It is noticeable that the number of layers from the very low-level (e.g. voltage firing, logic gates) to the high level are so numerous and complex (e.g. high level-programming languages, libraries), that it is very much a challenge even for an engineer to build a complete and detailed account of what an execution involves in a system, across all layers (see chapters 3 and 4 in (Lee, 2018), for a detailed view of all these layers of abstractions, their relationships, and what actually matters to an engineer). Each layer requires a high degree of expertise to be understood, to the point where one can wonder whether a complete mechanistic view of a computing system is understandable for a human:

> Once upon a time, every computer specialist had a gestalt understanding of how computers worked. The overall interactions among hardware, software, compilers, and the operating system were simple and transparent enough to produce a coherent picture of the computer's operations. As modern computer technologies have become increasingly more complex, this clarity is all but lost: the most fundamental ideas and techniques in computer science—the very essence of the field—are now hidden under many layers of obscure interfaces and proprietary implementations. An inevitable consequence of this complexity has been specialization, leading to computer science curricula of many courses, each covering a single aspect of the field. (Nisan and Schocken (2021), Preface)

Thus, we do not intend to provide a complete causal explanation of the reaction of the computer system and leave a few black boxes, just as the Turing machine is a form of mechanistic description, but without explaining how the mechanism of activation of the tape head works. As such, we will say that the execution model is a mechanistic schema, as it is defined for example by Machamer, Darden, and Craver (Machamer et al., 2000):

> A mechanism schema is a truncated abstract description of a mechanism that can be filled with descriptions of known component parts and activities. (...)

> Schemata exhibit varying degrees of abstraction, depending on how much detail is included. (...) Degrees of abstraction should not be confused with degrees of generality or scope.

What we call an execution model abstracts away from some details but is still legitimate as a mechanistic abstraction, following Boone and Piccinini (Boone and Piccinini, 2016), if it identifies the components, their properties, and relationships producing the phenomena. And what makes it specific compared to other mechanistic abstractions is that it focuses on what makes interactive behavior possible. We are not interested, for example, in the cooling system that may exist in a real computer, even if it is a component that would appear in a complete causal story.

Conclusion

The paper claims that new computing practices, as exemplified in Human-computer interaction, cannot be explained within the frame of computability theory and require a mechanistic description. We propose our concept of an execution model as a candidate for such a mechanistic description. We argue that a twofold model for computers (carrying both a model of computation and a mechanistic description) has ceased to exist. Therefore, there is no support for a mechanistic explanation of current interactive computers. Historically, the TM was such a twofold model at the time for Church-Turing computation and provided philosophers with a hint about computing behavior during execution. Nowadays, formal accounts of computing properties address increasingly specialized tasks, focusing on formalizing and verifying specific mathematical properties. Formalisms are not looking for an explanatory account of computers qua an abstraction describing the execution in mechanistic terms. It is now an open challenge to flesh out the updated minimal mechanistic description of interactive computation, gathering relevant abstraction from computing engineering and achieving a trade-off to be sufficiently general and understandable. In other words, the challenge is finding the minimal counterpart of the execution model that the TM was for Church-Turing computations. We have proposed and conceptualized a candidate for the kind of abstraction that could support the epistemology of interactive systems. Our notion of "execution model" is on board with the concept of functional architecture à la Pylyshyn but adds two dimensions to it. First, we extend it to describe the architecture of an interactive computer. Second, we sketch the underlying mechanism supporting the function of each component, concerned with execution — not merely architecture. An execution model can provide a mechanistic schema of an interactive execution. Such an abstraction is a trade-off and cannot consist of a complete mechanistic description: a computing system is too multi-layered to fit into a simple abstraction. Such trade-offs between completeness and explanatory power are familiar to programmers who build incomplete but explanatory views of the system they program.

Acknowledgements

We are very grateful to our anonymous reviewers for their objections and comments. This work was supported by the French Programme d'Investissements d'avenir ANR-17-EURE-0005 and by Agence de l'Innovation de Défense.

References

Accot, J., Chatty, S., Maury, S., and Palanque, P. (1997). Formal transducers: Models of devices and building bricks for the design of highly interactive systems. In *Design, Specification and Verification of Interactive Systems' 97*, pages 143–159. Springer. https://doi.org/10.1007/978-3-7091-6878-$3_1$0.

Agha, G. and Hewitt, C. (1988). Concurrent programming using actors: Exploiting large-scale parallelism. In A.H. Bond and L. Gasser, eds., *Readings in Distributed Artificial Intelligence*, pages 398–407. Morgan Kaufmann. https://doi.org/10.1007/3-540-16042-6_2.

Alur, R. and Dill, D.L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8.

Alur, R. and Henzinger, T.A. (1993). Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1), 35–77. https://doi.org/10.1006/inco.1993.1025.

Andersen, H.R., Mørk, S., and Sørensen, M.U. (1997). A universal reactive machine. In W.J. Mazurkiewicz A., ed., *CONCUR '97: Concurrency Theory. CONCUR 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 89–103. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-63141-0_7.

Arbach, Y., Karcher, D., Peters, K., and Nestmann, U. (2015). Dynamic causality in event structures. In V.M. Graf S., ed., *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *Lecture Notes in Computer Science*, pages 83–97. https://doi.org/10.1007/978-3-319-19195-9_6.

Attie, P.C. and Lynch, N.A. (2003). Dynamic input/output automata: a formal model for dynamic systems. Technical Report MIT-CSAIL-TR-2003-006, MIT Computer Science and Artificial Intelligene Laboratory.

Baeten, J.C., Luttik, B., and Tilburg, P.V. (2013). Reactive turing machines. In S.M.T.J. Owe O., ed., *FCT 2011: Fundamentals of Computation Theory*, pages 348–359. Lecture Notes in Computer Science, Springer Berlin Heidelberg. https://doi.org/10.1016/j.ic.2013.08.010.

Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., and Meuter, W.D. (2013). A survey on reactive programming. *ACM Computing Surveys*, 45(4), 1–34. https://doi.org/10.1145/2501654.2501666.

Baldan, P., Corradini, A., and Montanari, U. (2001). Contextual petri nets, asymmetric event structures, and processes. *Information and Computation*, 171(1), 1–49. https://doi.org/10.1006/inco.2001.3060.

Basman, A., Tchernavskij, P., Bates, S., and Beaudouin-Lafon, M. (2018). An anatomy of interaction: Co-occurrences and entanglements. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 188–196. Association for Computing Machinery. https://doi.org/10.1145/3191697.3214328.

Beaudouin-Lafon, M. (2006). Human-computer interaction. In D. Goldin, S.A. Smolka, and P. Wegner, eds., *Interactive Computation: The New Paradigm*, pages 227–254. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-34874-$3_1$0.

Bechtel, W. (1994). Levels of description and explanation in cognitive science. *Minds and Machines*, 4, 1–25. https://doi.org/10.1007/BF00974201.

Berry, G. and Gonthier, G. (1992). The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V.

Berry, G. and Serrano, M. (2020). Hiphop.js: (a)synchronous reactive web programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 533–545. PLDI 2020, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3385412.3385984.

Bonér, J., Farley, D., Kuhn, R., and Thompson, M. (2014). The reactive manifesto. *Reactivemanifesto.Org*.

Boone, W. and Piccinini, G. (2016). Mechanistic abstraction. *Philosophy of Science*, 83, 686–697. https://doi.org/10.1086/687855.

Boussinot, F. (1991). Reactive c: An extension of c to program reactive systems. *Software: Practice and Experience*, 21, 401–428. https://doi.org/10.1002/spe.4380210406.

Bozşahin, C. (2018). Computers aren't syntax all the way down or content all the way up. *Minds and Machines*, 28, 543–567. https://doi.org/10.1007/s11023-018-9469-2.

Campos, J.C. and Harrison, M.D. (1997). Formally verifying interactive systems: A review. In T.J. Harrison M.D., ed., *Design, Specification and Verification of Interactive Systems' 97. Eurographics*, pages 109–124. Springer. https://doi.org/10.1007/978-3-7091-6878-3_8.

Canny, A., Navarre, D., Campos, J.C., and Palanque, P. (2019). Model-based testing of post-wimp interactions using object oriented petri-nets. In *Formal Methods. FM 2019 International Workshops*, pages 486–502. Springer International Publishing. https://doi.org/10.1007/978-3-030-54994-$7_3$5.

Casiez, G. and Roussel, N. (2011). No more bricolage! methods and tools to characterize, replicate and compare pointing transfer functions. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 603–614. Association for Computing Machinery. https://doi.org/10.1145/2047196.2047276.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J.A. (1987). Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 178–188. POPL '87, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/41625.41641.

Chatty, S. (1994). Extending a graphical toolkit for two-handed interaction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, pages 195–204. Association for Computing Machinery. https://doi.org/10.1145/192426.192500.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2), 56–68. https://doi.org/10.2307/2266170.

Cockshott, P. and Michaelson, G. (2007). Are there new models of computation? reply to wegner and eberbach. *Computer Journal*, 50(2), 232–247. https://doi.org/10.1093/comjnl/bxl062.

Craver, C.F. (2001). Role functions, mechanisms, and hierarchy. *Philosophy of Science*, 68(1), 53–74. https://doi.org/10.1086/392866.

Czaplicki, E. and Chong, S. (2013). Asynchronous functional reactive programming for guis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 48, pages 411–422. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2462156.2462161.

Dearden, A.M. and Harrison, M.D. (1997). Abstract models for hci. *International Journal of Human Computer Studies*, 46(1), 151–177. https://doi.org/10.1006/ijhc.1996.0087.

Dodig-Crnkovic, G. (2011). Significance of models of computation, from turing model to natural computation. *Minds and Machines*, 21, 301–322. https://doi.org/10.1007/s11023-011-9235-1.

Dragicevic, P. and Fekete, J.D. (2004). Support for input adaptability in the icon toolkit. In *Proceedings of the 6th International Conference on Multimodal Interfaces*, pages 212–219. ICMI '04, Association for Computing Machinery. https://doi.org/10.1145/1027933.1027969.

Eberbach, E., Goldin, D., and Wegner, P. (2004). Turing's ideas and models of computation. In C. Teuscher, ed., *Alan Turing: Life and Legacy of a Great Thinker*, pages 159–194. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-05642-4_7.

Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, volume 32 of *ICFP '97*, pages 263–273. Association for Computing Machinery. https://doi.org/10.1145/258949.258973.

Fresco, N. (2010). Explaining computation without semantics: Keeping it simple. *Minds and Machines*, 20, 165–181. https://doi.org/10.1007/s11023-010-9199-6.

Glabbeek, R.V. and Plotkin, G. (2004). Event structures for resolvable conflict. In *29th International Symposium on Mathematical Foundations of Computer*, volume 3153 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 550–561. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-28629-$5_4$2.

Glennan, S. (2002). Rethinking mechanistic explanation. *Philosophy of Science*, 69(S3), 342–353. https://doi.org/10.1086/341857.

Goldin, D. and Wegner, P. (2008). The interactive nature of computing: Refuting the strong church-turing thesis. *Minds and Machines*, 18, 17–38. https://doi.org/10.1007/s11023-007-9083-1.

Goldin, D., Wegner, P., and Smolka, S.A. (2006). *Interactive Computation: The New Paradigm*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-34874-3.

Goldin, D.Q. (2000). Persistent turing machines as a model of interactive computation. In T.B. Schewe KD., ed., *Foundations of Information and Knowledge Systems. FoIKS 2000*, volume 1762 of *Lecture Notes in Computer Science*, pages 116–135. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-46564-2_8.

Graham, P. (2004). *Hackers and Painters: Essays on the Art of Programming*. O'Reilly Associates, Inc., USA.

Harel, D. and Pnueli, A. (1985). On the development of reactive systems. In K. Apt, ed., *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-82453-1$_1$7.

Hewitt, C. and Baker, H. (1978). Actors and continuous functionals.

Hill, R.D. (1986). Supporting concurrency, communication, and synchronization in human-computer interaction—the sassafras uims. *ACM Trans. Graph.*, 5, 179–210. https://doi.org/10.1145/24054.24055.

Hornbaek, K. and Oulasvirta, A. (2017). What is interaction? In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 5040–5052. Association for Computing Machinery. https://doi.org/10.1145/3025453.3025765.

Huot, S., Dumas, C., Dragicevic, P., Fekete, J.D., and Hégron, G. (2004). The magglite post-wimp toolkit: draw it, connect it and run it. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 257–266. UIST '04, Association for Computing Machinery. https://doi.org/10.1145/1029632.1029677.

Jacob, R.J.K. (1996). Human-computer interaction: Input devices. *ACM Comput. Surv.*, 28(1), 177–179. https://doi.org/10.1145/234313.234387.

Jacob, R.J.K., Deligiannidis, L., and Morrison, S. (1999). A software model and specification language for non-wimp user interfaces. *ACM Trans. Comput.-Hum. Interact.*, 6, 1–46. https://doi.org/10.1145/310641.310642.

Klein, C. (2020). Polychrony and the process view of computation. In *Proceedings of the 2018 biennial meeting of the Philosophy of Science Association. Part II*, volume 87, pages 1140–1149. https://doi.org/10.1086/710613.

Ko, A.J. and Myers, B.A. (2004). Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 151–158. CHI '04, Association for Computing Machinery. https://doi.org/10.1145/985692.985712.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21, 558–565. https://doi.org/10.1145/359545.359563.

Lassègue, J. and Longo, G. (2012). What is turing's comparison between mechanism and writing worth? In D.A.L.B. Cooper S.B., ed., *How the World Computes. CiE 2012.*, volume 7318 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-30870-3$_4$6.

Lee, E.A. (2018). *Plato and The Nerd*. MIT Press. https://doi.org/10.7551/mitpress/11180.001.0001.

Lee, E.A. (2020). *The Coevolution*. MIT Press. https://doi.org/10.7551/mitpress/12307.001.0001.

Lee, E.A. and Neuendorffer, S. (2006). Concurrent models of computation for embedded software. *System-on-Chip: Next Generation Electronics*. https://doi.org/10.1049/PBCS018E$_c$*h*7.

Lee, E.A. and Sangiovanni-Vincentelli, A. (1998). A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. https://doi.org/10.1109/43.736561.

Lee, E.A. and Varaiya, P. (2003). *Structure and Interpretation of Signals and Systems*. Addison-Wesley, first edition edition.

Leeuwen, J.V. and Wiedermann, J. (2001). Beyond the turing limit: Evolving interactive systems. In R.P. Pacholski L., ed., *SOFSEM 2001: Theory and Practice of Informatics*, volume 2234 of *Lecture Notes in Computer Science*, pages 90–109. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45627-9$_8$.

Leeuwen, J.V. and Wiedermann, J. (2006). A theory of interactive computation. In *Interactive Computation: The New Paradigm*, pages 119–142. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-34874-3$_6$.

Leveson, N. (2020). Are you sure your software will not kill anyone? *Commun. ACM*, 63(2), 25–28. https://doi.org/10.1145/3376127.

Longo, G. (1999). *The Difference between Clocks and Turing Machines*, volume 27, pages 211–232. Springer Netherlands, Dordrecht. https://doi.org/10.1007/978-94-015-9620-6$_1$4.

Luttik, B. and Yang, F. (2016). On the executability of interactive computation. In B.L.J.N. Beckmann A., ed., *Pursuit of the Universal. CiE 2016*, volume 9709 of *Lecture Notes in Computer Science*, pages 312–322. Springer, Cham. https://doi.org/10.1007/978-3-319-40189-8$_3$2.

Lynch, N., Segala, R., and Vaandrager, F. (1996). Hybrid i/o automata. *Information and Computation*, 185(1), 105–157. https://doi.org/10.1007/BFb0020971.

Machamer, P., Darden, L., and Craver, C.F. (2000). Thinking about mechanisms. *Philosophy of Science*, 67(1), 1–25. https://doi.org/10.1086/392759.

MacLennan, B. (2003). Transcending turing computability. *Minds and Machines*, 13, 3–22. https://doi.org/10.1023/A:1021397712328.

Mandel, L. and Pouzet, M. (2005). Reactiveml, a reactive extension to ml. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 82–93. PPDP '05, Association for Computing Machinery. https://doi.org/10.1145/1069774.1069782.

Marr, D. (2010). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. MIT Press. https://doi.org/10.7551/mitpress/9780262514620.001.0001.

McClamrock, R. (1990). Marr's three levels: A re-evaluation. *Minds and Machines*, 1, 185–196. https://doi.org/10.1007/BF00361036.

Miłkowski, M. (2014). Computational mechanisms and models of computation. *Philosophia Scientiae*, 18(3), 215–228. https://doi.org/10.4000/philosophiascientiae.1019.

Miłkowski, M. (2016). A mechanistic account of computational explanation in cognitive science and computational neuroscience. In V.C. Müller, ed., *Computing and Philosophy: Selected Papers from IACAP 2014*, volume 375, pages 191–205. Springer International Publishing. https://doi.org/10.1007/978-3-319-23291-1$_1$3.

Milner, R. (1975). Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, eds., *Logic Colloquium '73*, volume 80, pages 157–173. Elsevier. https://doi.org/10.1016/S0049-237X(08)71948-7.

Milner, R. (1982). Four combinators for concurrency. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 104–110. PODC '82, Association for Computing Machinery. https://doi.org/10.1145/800220.806687.

Milner, R. (1983). Calculi for synchrony and asynchrony. *Theoretical Computer Science*. https://doi.org/10.1016/0304-3975(83)90114-7.

Milner, R. (1993). Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1), 78–89. https://doi.org/10.1145/151233.151240.

Milner, R. (1999). *Communicating and Mobile Systems: the -calculus*. Cambridge University Press.

Milner, R. (2006). *Turing, computing and communication*, pages 1–8. Springer-Verlag. https://doi.org/10.1007/3-540-34874-3_1.

Mol, L.D. (2018). Turing machines. *Stanford Encyclopedia*. URL `https://plato.stanford.edu/entries/turing-machine/#TuriMachModeComp`.
URL: *https://plato.stanford.edu/entries/turing-machine/TuriMachModeComp*

Myers, B. (1994). Challenges of hci design and implementation. *Interactions*, 1(1), 73–83. https://doi.org/10.1145/174800.174808.

Myers, B., Hudson, S.E., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7, 3–28. https://doi.org/10.1145/344949.344959.

Myers, B., Park, S.Y., Nakano, Y., Mueller, G., and Ko, A. (2008). How designers design and program interactive behaviors. In *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, pages 177–184. IEEE. https://doi.org/10.1109/VLHCC.2008.4639081.

Myers, B.A. (1991). Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, UIST 1991*, pages 211–220. UIST '91, Association for Computing Machinery. https://doi.org/10.1145/120782.120805.

Myers, B.A. (2013). Improving program comprehension by answering questions (keynote). In *Proceedings of the 21st IEEE/ACM Conference on Program Comprehension (ICPC)*, pages 1–2. ICPC. https://doi.org/10.1109/ICPC.2013.6613827.

Myers, B.A., Giuse, D., Mickish, A., Zanden, B.V., Kosbie, D., McDaniel, R., Landay, J., Goldberg, M., and Pathasarathy, R. (1994). The garnet user interface development environment. In *Conference Companion on Human Factors in Computing Systems*, pages 25–26. CHI '94, Association for Computing Machinery. https://doi.org/10.1145/259963.260472.

Myers, B.A., Giuse, D.A., Dannenberg, R.B., Zanden, B.V., Kosbie, D.S., Pervin, E., Mickish, A., and Marchal, P. (1995). Garnet comprehensive support for graphical,

highly interactive user interfaces. In R. Baecker, J. Grudin, W. Buxton, and S. Greenberg, eds., *Readings in Human–Computer Interaction*, pages 357–371. Morgan Kaufmann. https://doi.org/10.1016/B978-0-08-051574-8.50037-6.

Myers, B.A. and Rosson, M.B. (1992). Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 195–202. CHI '92, Association for Computing Machinery. https://doi.org/10.1145/142750.142789.

Navarre, D., Palanque, P., Dragicevic, P., and Bastide, R. (2006). An approach integrating two complementary model-based environments for the construction of multimodal interactive applications. *Interacting with Computers*, 18, 910–941. https://doi.org/https://doi.org/10.1016/j.intcom.2006.03.002.

Navarre, D., Palanque, P., Ladry, J.F., and Barboni, E. (2009). Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Transactions on Computer-Human Interaction*, 16, 1–56. https://doi.org/10.1145/1614390.1614393.

Nielsen, M., Plotkin, G., and Winskel, G. (1981). Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13(1), 85–108. https://doi.org/10.1016/0304-3975(81)90112-2.

Nisan, N. and Schocken, S. (2021). *The Elements of Computing Systems. Building a Modern Computer from First Principles*. MIT Press, second edition edition.

Petri, C. (1980). Introduction to general net theory. In W. Brauer, ed., *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-10001-6$_2$1.

Piccinini, G. (2007). Computing mechanisms. *Philosophy of Science*. https://doi.org/10.1086/522851.

Piccinini, G. (2008a). Computers. *Pacific Philosophical Quarterly*, 89(1), 32–73. https://doi.org/10.1111/j.1468-0114.2008.00309.x.

Piccinini, G. (2008b). Some neural networks compute, others don't. *Neural Networks*, 21, 311–321. https://doi.org/10.1016/j.neunet.2007.12.010.

Post, E. (1948). Degrees of recursive unsolvability: preliminary report. *Bull. Amer. Math. Soc.*, 54, 641–642.

Prasse, M. and Rittgen, P. (1998). Why church's thesis still holds. some notes on peter wegner's tracts on interaction and computability. *The Computer Journal*, 41(6), 357–362. https://doi.org/10.1093/comjnl/41.6.357.

Pylyshyn, Z. (1986). *Computation and Cognition. Toward a Foundation for Cognitive Science*. MIT Press.

Rapaport, W.J. (2018). What is a computer? a survey. *Minds and Machines*, 28(3), 385–426. https://doi.org/10.1007/s11023-018-9465-6.

Reisig, W. (1988). Temporal logic and causality in concurrent systems. In F. Vogt, ed., *CONCURRENCY 1988*, volume 335 of *Lecture Notes in Computer Science*, pages 121–139. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-50403-6$_3$7.

Salvaneschi, G., Margara, A., and Tamburrelli, G. (2015). Reactive programming: A walkthrough. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 953–954. IEEE. https://doi.org/10.1109/ICSE.2015.303.

Schmidt, K. and Bansler, J. (2016). Computational artifacts: interactive and collaborative computing as an integral feature of work practice. In B.L.M.P.B.S. De Angeli A., ed., *COOP 2016: Proceedings of the 12th International Conference on the Design of Cooperative Systems, 23-27 May 2016, Trento, Italy*, pages 21–38. Springer, Cham.

Segala, R., Gawlick, R., Søgaard-Andersen, J., and Lynch, N. (1994). Liveness in timed and untimed systems. In S.E. Abiteboul S., ed., *Automata, Languages and Programming. ICALP 1994*, volume 820 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-58201-$0_6$6.

Shagrir, O. (2006). Gödel on turing on computability. In A. Olszewski, J. Wolenski, and R. Janusz, eds., *Church's Thesis After 70 Years*, pages 393–419. De Gruyter, Berlin, Boston. https://doi.org/10.1515/9783110325461.393.

Shagrir, O. (2012). Computation, implementation, cognition. *Minds and Machines*, 22, 137–148. https://doi.org/10.1007/s11023-012-9280-4.

Smith, B.C. (2002). The foundations of computing. In M. Scheutz, ed., *Computationlism: New Directions*. MIT Press.

Smith, W.R. (1995). Using a prototype-based language for user interface: The newton project's experience. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 61–72. OOPSLA '95, Association for Computing Machinery. https://doi.org/10.1145/217838.217844.

Soare, R.I. (2009). Turing oracle machines, online computing, and three displacements in computability theory. *Annals of Pure and Applied Logic*, 160(3), 368–399. https://doi.org/10.1016/j.apal.2009.01.008.

Soare, R.I. (2013). *Interactive computing and relativized computability*, chapter 9, pages 214–271. MIT Press. https://doi.org/10.7551/mitpress/8009.003.0010.

Suchman, L.A. (1987). *Plans and situated actions: The problem of human-machine communication*. Cambridge university press.

Turing, A. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 230–265. https://doi.org/10.1112/plms/s2-42.1.230.

Turing, A. (1939). Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, s2-45(1), 161–228. https://doi.org/10.1112/plms/s2-45.1.161.

Turing, A. (1950). Computing machinery and intelligence. *Minds*, 59, 433–60. URL `http://www.jstor.org/stable/2251299`.
URL: *http://www.jstor.org/stable/2251299*

Victor, B. (2012). Learnable programming–designing a programming system for understanding programs. URL `http://worrydream.com/LearnableProgramming`, accessed: 2021-08-20.
URL: *http://worrydream.com/LearnableProgramming*

Vonder, S.V.D., Koster, J.D., Myter, F., and Meuter, W.D. (2017). Tackling the awkward squad for reactive programming: The actor-reactor model. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, pages 27–33. REBLS 2017, Association for Computing Machinery. https://doi.org/10.1145/3141858.3141863.

Wegner, P. (1995). Interaction as a basis for empirical computer science. *ACM Computing Surveys (CSUR)*, 27(1), 45–48. https://doi.org/10.1145/214037.214092.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5), 80–91. https://doi.org/10.1145/253769.253801.

Williams, G. (1984). Software frameworks. *BYTE Magazine*, 9(13), 124–127.

Wilson, D., Rosenstein, L., and Shafer, D. (1990). *C++ Programming with MacApp*. Macintosh inside out, Addison-Wesley Publishing Company. URL `https://vintageapple.org/macprogramming/pdf/C++_Programming_with_MacApp_1987.pdf`.
URL: *https://vintageapple.org/macprogramming/pdf/C++Programming*$_{w}$*ith*$_{M}$*acApp*$_{1}$987*.pdf*