Abstract Platforms of Computation

Matthew C. Spencer[1] and **Etienne B. Roesch** and **Slawomir J. Nasuto**[2]
Thomas Tanay and **J. Mark Bishop**[3]

Abstract. Computational formalisms have been pushing the boundaries of the field of computing for the last 80 years and much debate has surrounded what computing entails; what it is, and what it is not. This paper seeks to explore the boundaries of the ideas of computation and provide a framework for enabling a constructive discussion of computational ideas. First, a review of computing is given, ranging from Turing Machines to interactive computing. Then, a variety of natural physical systems are considered for their computational qualities. From this exploration, a framework is presented under which all dynamical systems can be considered as instances of the class of abstract computational platforms. An abstract computational platform is defined by both its intrinsic dynamics and how it allows computation that is meaningful to an external agent through the configuration of constraints upon those dynamics. It is asserted that a platform's computational expressiveness is directly related to the freedom with which constraints can be placed. Finally, the requirements for a formal constraint description language are considered and it is proposed that Abstract State Machines may provide a reasonable basis for such a language.

1 INTRODUCTION

Over the last 80 years, computing has developed considerably, but there is still much debate about what "computing" is. When Alan Turing broached the field in the 1930's, he provided a very precise understanding of computing machines and computable problems. However, as variations on Turing's original mechanisms have been explored, the original definitions have become less appropriate and modern computational praxis now bares little resemblance to the original conception. The gap between practice and theory in computer science has been noted in other works [8, 6, 5, 16] which have strived to provide broad reviews of the field and suggest modern approaches to the discourse of computing. This discourse will be further explored here with an aim to provide a unified framework for defining computation and positioning popular computing formalisms.

There appear to be two central issues within the computing community: computational expressiveness and the scope of what can be called computation. The former issue is a discussion about the scope of problems that can be addressed with different computational formalisms, beginning with classical Turing Machines [23] and discussing other, more advanced concepts [12, 24, 6, 15, 16, 9]. The latter issue accepts the varying scopes of computational expressiveness and the proliferation of computational paradigms and explores the boundaries of what computation might be [5]. Both of these issues will be explored here to help establish the depth and breadth of the proposed framework.

Finally, if there were a unified framework of computation, a formal abstract language for describing programs would be valuable. Modern programming languages are largely platform dependent but a number of abstract formalisms [19, 11, 14, 10] might provide possibilities for platform-agnostic program descriptions. Such program descriptions will be considered here.

The following paper will first review computational paradigms, starting with Turing Machines. This discussion will address notions of super-Turing Machine expressiveness with a specific discussion on modern views on interaction. Interactive computing discusses concurrent partially coupled systems which, when shifted to the continuous time domain, begin to resemble natural physical systems. To differentiate computation from physical processes, a constructive view on the boundaries of computation will be explored. From these previous ideas, a framework of abstract computational layers will be proposed. Notionally, this framework will encompass the breadth of computational paradigms. Finally, the requirements of an abstract program description language for this framework will be discussed.

2 CLASSICAL COMPUTING

2.1 Functions and algorithms

A class of functions, called "effectively computable functions", contains those functions that can be worked out through finite sequences of simple mechanistic operations. Prior to the 1930s, such sequences (known as algorithms), lacked a formal definition and it was poorly understood which functions were effective and which were not. In the 1930s, both Alonzo Church and Alan Turing approached this problem whilst addressing the *Entsheidungsproblem*. Aiming to introduce a formal description of algorithms, Church produced the λ-calculus and went on to show the unsolvability of the *Entsheidungsproblem* [3]. Simultaneously, Turing approached this same problem, but from a different angle. Taking seriously the notion of sequences of mechanical operations, Turing formalised algorithms with his "abstract machines" (which are now commonly known as Turing Machines (TMs)) and went on to provide what is generally accepted to be a more convincing proof of Church's result [23]. Ultimately, Turing Machines and the λ-calculus are equivalent formalisms for representing the class of effectively computable functions, a point which is embodied in the Church-Turing Thesis (CTT).

When initially conceptualized, the CTT described this class of effective functions and the algorithms used to solve them. These algorithms necessarily had the very particular property that they processed mechanically, from some input to some output through a finite sequence of simple, ordered operations. By "mechanistic pro-

[1] email: m.c.spencer@reading.ac.uk
[2] University of Reading, UK
[3] Goldsmiths, University of London, UK

cessing", it was understood that the sequences were carried out deterministically, and that no insight or agency could interfere in the processing. By "simple operations", it was understood that each operation in the algorithm's description was trivially performed by a human mathematician. It was on this basis that Turing conceived of his machines.

2.2 Turing machines

Turing's *abstract machine* involved a tape (a bidirectional, arbitrary length, linear string of symbols), a read/write head pointing to a location on the tape, a set of possible configurations in which the machine could exist, and a table of instructions that could move the head, read and write the tape, and change the configuration of the machine. The machine would proceed mechanically, looking up an instruction in the table using the current configuration and the input symbol on the tape. Then, this instruction would be executed, potentially resulting in modifying the symbol on the tape, changing the machine's configuration, or shifting the tape one step to the left or right. The machine then proceeded to look-up the next operation from the table and so on until the machine would finally halt. Between the instructions in the table and the set of available configurations, each machine could embody a single algorithm. To execute the algorithm, the input was provided on the tape and the machine processed until halting. Once the machine had terminated, the tape would contain the output of the algorithm. While termination was not an absolute necessity (specifically in the case of computing numbers to arbitrary precision), the full answer could not be known unless the machine halted.

For the sake of the topic of this paper, the Turing Machine can be thought of as machine providing a certain intrinsic and deterministic dynamics. These dynamics are predicated on the ability to execute a small set of elementary operations, specifically, reading and writing tape symbols, moving the tape, reading the current configuration, and changing the configuration. If any of these operations were non-trivial for the executor, a Turing Machine would not, on its own, provide a satisfactory formal representation of an algorithm. Turing Machines are, by their very conception, abstract entities. They were not designed to be actually implemented, and yet, if each of the elementary operations is appropriately simple, implementation as a physical machine could be done by embedding the TM design in a physical form.

Further to the intrinsic dynamics of TMs, they provide a flexible approach to customizing the constraints on those dynamics through the set of configurations and the writing of the instruction table. It is through this customization that the general class of TMs can instantiate a TM for a specific algorithm. However, there are natural limitations intrinsic to the TM formalism which cannot be overcome through this customization (further discussed in Section 2.3). These limitations were not an oversight, but were necessary for formally representing effectively computable functions. However, modern digital computers do not exclusively describe this class of functions.

Later in his career, Turing considered several extensions to the TM concept. An important extension was that of the Universal Turing Machine (UTM) – a Turing Machine capable of simulating any other Turing Machine. From this idea, universal computers (for computing more than a single function) were devised, ultimately resulting in modern computers. However, since modern computers have a finite memory, it is argued that they are strictly less powerful than UTMs, which may have a tape of arbitrary length. Nonetheless, it has been argued [5] that modern computers can be (and regularly are) used for more than the computation of functions, and are therefore *more* expressive than UTMs. These views will be discussed in Section 3.1.

Other extensions that Turing considered were those that incorporated non-mechanical components into the machine's structure; in other words, including non-trivial operations. Two such components that are popularly discussed are so-called "oracles" and human users. In Oracle Machines, the TM is able to consult an all-knowing oracle, which is capable of delivering a non-trivial answer in finite time, and while such Oracles do not exist in practice, they do serve to describe an abstract computing paradigm of consulting an outside expert. Such consultation is also represented in Turing' Choice Machines, in which the TM might pause occasionally to query a human user for additional input. In both of these cases, the TM architecture has been extended to include non-mechanical components and would not be what Turing considered to be "automatic machines" or "purely mechanical".

2.3 Restrictions of Church-Turing formalisms

While Turing Machines provide a robust and formal description of algorithms, they feature a number restrictions which define the space of their applicability. However, owing to these in-built restrictions, other, more computationally expressive paradigms can be imagined. One of the central restrictions to the TM paradigm is the notion of operating on a finite alphabet of symbols. Since the alphabet is finite, it cannot represent all real numbers or other continuous concepts. This has implications for certain types of scientific computing where arbitrary precision is desirable. This also has implications for considering TMs as capable of intelligence, owing to limited and inflexible nature of the alphabet. Further, TMs receive input and emit output encoded by the same alphabet, reducing the types of functions that can be implemented (such as are found in the broader set of bijective mappings).

To consider other restrictions, one must take a larger view of the TM as a computational machine embedded within an environment containing at least one other agent. This other agent, the user of the TM, is implicit in the TM's definition since the input has to arrive on the tape from somewhere and the output must be requested for some purpose that is not the TM's own. In this wider view, other restrictions become clear, namely the temporal insensitivity and the synchronicity of the input and output. In other words, once the user has provided input to the TM, they must wait some time for the TM to complete its operation before it delivers output; additional input cannot be supplied as the TM operates. Also, the input and output must be specified in the same alphabet. Furthermore, for every input there is guaranteed to be a single output, so concepts including multi-input/multi-output or streaming interaction with the environment cannot be modelled. Finally, a TM lacks memory that persists between inputs, which prevents it from modelling a learning system.

3 HYPER-COMPUTABILITY

It is from the class of effectively computable functions, which the notion of "computability" takes its generally understood meaning. If a function is effectively computable, it is computable and has an associated algorithm for computing it. Likewise, it also has a Turing Machine or λ-calculus representation. Similarly, if it is not computable, none of these other representations apply.

The formal definition of "computable" is generally taken to be "that which is computable by a Turing Machine". Thus, any machine formalism that is more expressive or capable than a Turing Machine is generally called a "hyper-computer" and is capable of "hyper-computation", though perhaps adhering to the term "super-Turing" is clearer. One example is the Zeno-machine, which performs each subsequent operation in half the time of the previous one, appealing to the Zeno paradox to perform an infinite number of operations in a finite span of time. Likewise, most hyper-computers include the notion of infinity into their construction (infinite length alphabet, infinite speed, infinite knowledge, etc) [15] and as such, are unsuitable models for practical computing or investigating artificial intelligence or human cognition [16]. However, two types of hyper-computing are notable for their practical importance: machines that are capable of continuous/analog information manipulation and machines that incorporate real-time interaction with their environments.

In the first case, the TM's finite alphabet of discrete symbols prevents it from computing problems that require the infinite precision of real numbers. Specifically, if all numbers must be represented using a finite length string of symbols from a finite length alphabet, the range of numerical values is at most a countable. For instance, while a single symbol, such as π, can represent an irrational number, a finite alphabet of symbols might not have symbols to represent the values of *e* or *i* and, if it does, then the inter-symbol relationships would require a look-up table since infinite precision calculations would take infinitely long to complete. Thus, exact quantities cannot always be defined or manipulated. Since most empirical measurements involve such quantities, digital computers are forced to approximate the real numbers instead. Aside from the purely pragmatic desire to process on real numbers, digital computers restrict precision when modelling continuous dynamical systems, which is problematic when considering the sensitivity of chaotic dynamics. Also as computers are used to investigate intelligence which is arguably embedded in a continuous physical space, their inability to deal with real numbers limits the extent of this field of research. However, while the processing of real numbers is a clearly practical and desirable capability for computers to have, it is unlikely that it will ever be realized on digital computers (in which higher precision entails greater space requirements).

On the other hand, interactive computation has been a mainstay of the software industry for decades; and while it is ubiquitous, it has only been widely recognized as a super-Turing paradigm within the last ten years. By now, it is quite clear that many features of a TM (synchronous input-output, the unified alphabet, and necessity of termination, to name a few) do not describe systems like autonomous robots, word processors, operating systems, or the internet [26, 12, 6, 5, 16]. For each of these systems, the time-sensitive input is streamed to the machine as the machine works, and the machine's output at any given time is potentially a product of its entire history of operation. Similarly, there is no single ultimate output that these machines produce, but rather they are expected to continuously operate, remaining responsive to the environment.

The internet represents another form of interaction, in which the environment does not simply provide operational input to the machine, but may also alter the machine's very construction at any time. Whether or not these alterations enhance or reduce the machine's capability, they may fundamentally change how the machine will be able to respond to future input, including gaining the capacity to operate on a newer or larger alphabet [12].

What makes interactive computing more expressive than TM-equivalent paradigms is that interactive computing can express more than functions. Granted, interactive computing may not be able to express all functions, but the ones that are TM-computable exist as a special case where the input-output relationships of the interactive machine are restricted to the TM definition. Essentially, this generalizes the purpose of the machine from computing functions to generally performing a wider set of tasks [6, 5].

3.1 Interaction as hyper-computation

There are many types of the interaction that can be considered, but not all of them entail a system with super-TM capability. For instance, a TM itself defines interaction between the tape and tape-head. In the weakest sense, "interaction" merely suggests the presence of more than one non-independent entity in a system. However, under specific stronger notions of interaction, the restrictions on the Turing Machine formalism are relaxed and super-TM expressiveness ensues. This stronger form of interaction is minimally defined by the following two characteristics:

Definition 1. Coupling: *Current output of an interactive computer affects its future input.*

Definition 2. Persistence: *Current output of an interactive computer is affected by more than current input.*

Without these two characteristics added to the current capabilities of a TM, the interactive machine would have no more expressiveness than repeated calls to a TM. However, with these two characteristics, the machine is capable of modifying its environment in a meaningful way, partially affecting its own future input and therefore making decisions based on potentially all of the input it has ever received. While the first characteristic could be considered a property of the machine's environment (rather than of the machine itself), it can be argued that this property provides a type of sensorimotor coupling within the machine, thus making the machine not just reactive, but active and proactive as well. It could also be argued that a TM might have this property in the reactive sense, but without Persistence, lacks the capability for a long-term action strategy.

It is easy to show at this point that TMs are a special case of this type of interactive computers where the environment is ambivalent about the machine's output and the machine is completely amnesiac. Thus, interactive computers form a proper superset of Turing Machines and are therefore more expressive [24].

There may remain some question, however, about whether these interactive machines are "automatic" in Turing's sense of the word. The initial TM was automatic in the sense that once the input was finalized on the tape, the machine would deterministically produce the output such that each subsequent configuration of the machine (and tape) was entirely determined by the previous configuration. It has been argued that if the machine had paused to query an outside source for input (from a human or an all-knowing oracle, for instance), then the machine would not have been purely mechanical. It could be argued that since interaction with the environment forms a crucial part of an interactive computer's function, that an interactive computer is not purely mechanical. However, there is no reason to suspect that, given the current configuration of an interactive computer and the value of the current input from the environment, that the subsequent operation of the computer would not be entirely deterministic (until the next input arrived). In fact, given a sequence of input symbols from the environment, S, if S were fed to two identical interactive computers that are initially in the same state, both computers ought to produce identical streams of output. While this example neglects the organic role of the environment (namely neglecting the first principle above), it shows that interactive computers may be automatic machines. The key to this argument lies in the distinction that the environment, while a part of the interaction, is not a part of the machine with which it interacts.

3.2 Models of interactive computing

3.2.1 Persistent Turing Machines

A common model of interactive computing that expresses the properties of Coupling and Persistence is the Persistent Turing Machine (PTM) [7]. The PTM formalisms builds from the standard TM by altering the input-output mechanics and by incorporating internal memory. This is done by giving the PTM three tapes: one for read-only input, one for write-only output, and an internal, read/write tape for working memory. The PTM functions at two time scales: during each macro-step, the environment synchronously provides input on the input tape and consumes output from the output tape, while the PTM produces the appropriate output for the given input over a sequence of micro-steps. Thus, while the above two properties of interaction have been added to the TM construction, the input and output are still completely synchronous. However, because the internal tape is persistent across macro-steps, identical input may not always produce the same output.

3.2.2 Interactive Machines

A similar construct has also been investigated by Van Leeuwen and Wiedermann [13], referred to as "Interactive Machines" (IMs). While IMs possess both Coupling and Persistence, it also has a weaker input-output relationship, defined as the "interactiveness" property, which states:

Definition 3. Interactiveness: *Any time an IM receives a non-null input symbol from the environment, it must provide a non-null output symbol to the environment some finite time later, and* vice versa.

Thus, rather than have synchronous input and output, such that input and output can be described in pairs, IMs have asynchronous input and output, such that any number of inputs can be fed to the automaton so long as some output is given some time later. While automata that have interactiveness may have their inputs and outputs interleaved, there is not even the stipulation that there be a one-to-one input-output relationship. This weaker input-output relationship positions PTMs as a special case of IMs. Further, unlike PTMs, IMs do not operate at multiple time scales, though blank symbols are defined for both input and output which may allow the simulation of multiple time scales.

3.2.3 Lineages of automata

Another model of interactive computing, Site Machines [24], represent the notion of a physical machine, such as a desktop computer. Aside from interacting with the machine through its conventional channels of input and output (such as a keyboard and monitor), one might also alter the machine's physical construction (by adding more memory or a new communication interface). These interactions may serve to augment the machine's capabilities, or to diminish them (such as enacting physical harm on the machine's components), but either way they have important ramifications for the future of the machine's conventional operation. This same concept can be extended to the Internet as a whole, where new components are regularly added and removed, not simply changing the computational power, but fundamentally altering the machine architecture.

Site or Internet Machines can, at any time, be conceived as complex, multi-dimensional ω-transducers (automatons that process an infinite input streams into infinite output streams). Then, at specific times when physical modifications occur, these transducers can be replaced by other ones with other capabilities. Thus, Site and Internet machines are appropriately represented as sequences or lineages of ω-transducers, and ultimately, the complexity of the input stream is only limited by the complexity of the most complex transducer. Since the set of situations in which the input stream evolves its alphabet of symbols as time progresses cannot be modelled by a Turing Machine, lineages of automata present a strictly more expressive class of machines [25]. The notion of lineages where preceding transducers physically construct their successors have been explored and are referred to as autopoietic automata [27], though their computational expressiveness derives entirely from the notion of lineages.

4 BOUNDARIES OF COMPUTING

Concurrency is a central notion in interactive computing. In the simplest sense, the environment and the automaton are processing in parallel and their behaviour is mutually coupled through the input and output ports of the automaton. However, as more automata are included within the environment, more parallel, semi-enclosed systems are available for direct or indirect interaction. Further, each automaton itself may be a multi-scale system, with high-level operations delegating to lower-level components for execution details (such as is explicitly described by the λ-calculus). Thus, this entanglement of semi-decoupled interactive dynamical systems begins to resemble discrete-time versions of natural physical processes. In fact, any physical process could be described as an interactive computer, given the Coupling and Persistence properties of interactive computing.

In fact, a school of thought called *pancomputationalism* would argue that all physical processes are intrinsically computing. This idea suggests that the laws of physics are computational rules which are processed continuously as time progresses, shifting the current state of the universe to the next. However, this view devalues the concept of computation and there might be more constructive ways to discuss physical systems as computers. To begin this constructive discussion, it is necessary to understand the boundaries of computing.

In his 1992 book, John Searle paraphrased this notion by saying "For any object there is some description of that object such that under that description the object is a digital computer" [21] and Jack Copeland has referred to this as Searle's Theorem[4], for the sake of argument [4] . Copeland rephrases this statement to clarify it, by saying that any object, e, can be described by mapping its states by some labelling, L, such that the pair $< e, L >$ is a computer. Both Searle and Copeland agree that under this definition, e is not intrinsically computational but that it can be described as such with an appropriate selection of L. However, Copeland argues, not just any L is valid.

In Searle's initial statement, he went on to say that there existed a description of a wall such that the wall was implementing a word processor. Copeland shows that for this to be possible, L must be time-sensitive and defined after the fact. In other words, a labelling could be constructed to map the states of an observed history of the wall to the states of a single observed computation of a word processor, but both sequences would have to be completely observed first and the labelling would have to be applied afterwards. This means that not only could L not be defined *before* the wall "executed" the word processor program, once L was defined, it would only apply to the wall at a specific set of time instances, and would be invalid at

[4] Though Searle himself only states the theorem as an absurd logical extension of some of the contemporary theoretical computational discourse, it serves as a concise statement of the central problem.

any other time or for any other run of the program. While $< e, L >$ could be called a computer implementing a word processor in this case, Copeland argues that L constitutes a "non-standard" description of the wall. Alternatively, he argues that only so-called "honest" descriptions ought to be valid for $< e, L >$ to form a valid computer.

The crux of Copeland's argument lies in the dynamics and semantics of e and L. He argues that for the description of a wall to describe a computer implementing a word processor, L would contain all of the computational power, in other words, all of the dynamics *and* semantics of the system; in fact, the wall could be swapped out for any other object. He argues that an "honest" description of an object requires that the majority (if not all) of the dynamics exist solely within e. Meanwhile, L provides an arbitrary, time-invariant symbolic representation of the states of e while respecting the natural dynamics of e.

An element necessary to this discussion has merely been implied thus far: for there to be a semantic description of an object, there must be a subject doing the describing. If the semantics are not an intrinsic part of a physical system, then they have to be bestowed upon that system from somewhere else. This represents an instance of an epistemic cut [17], where syntactic manipulation of symbols by a dynamic process (for which the symbols have no meaning) produces meaningful information or performs a meaningful task for a subject (by whom the symbols have their meaning bestowed). Even a common digital computer is not a discrete symbolic machine, but a severely constrained dynamical system in which states of certain components are given specific meanings (eg. 5V stored in a flip-flop may be interpreted as a binary 1, while 0V is interpreted as a binary 0). While there is a general question about how to bridge epistemic cuts in nature [17], it is clear that in the case of practical computation, the physical system is constrained by human hands and given meaning by human minds, such that when a human symbol is fed to the physical system it will produce another human symbol, while remaining oblivious to the meaning of its action.

In fact, another element of computation that is not discussed by Searle or Copeland in this context is that of programming or algorithms. While a dynamical process can have its states labelled to express a single computation (eg. a single pass through an algorithm), there has been no discussion about how to generalize these dynamics to multiple computations (ie. an algorithm run multiple times with different inputs). For a dynamical system to be useful for computation, it must provide a number of degrees of freedom for placing custom constraints upon the dynamics. Thus, even if a wall could be honestly mapped to a computational process, it would still severely restrict the types of tasks that could be computed. This implies that, while any physical system might be described as a computer, not every physical system is a generalized computer, and there will be limits to that system's computational expressiveness.

5 PHYSICAL SYSTEMS AS COMPUTERS

A wide variety of physical systems can be described as useful computational platforms; systems that provide intrinsic and predictable dynamics and a formalized approach to placing constraints on those dynamics. Several examples of such systems will be explored in this section, to demonstrate the flexibility of this framework for conceiving of computation.

First, one could consider the natural physical system of light and occlusion, whereby opaque objects block light and cast shadows. When appropriately constrained, this system can provide meaningful computation through the shape and location of shadows. For instance, a sundial positions a labelled face and a gnomon such that the shadow cast by the gnomon's edge aligns with the labels on the face to indicate the time of day (Figure 5). In a sense, a sundial represents a program for telling time on the platform of the physical system of occlusion and celestial mechanics.

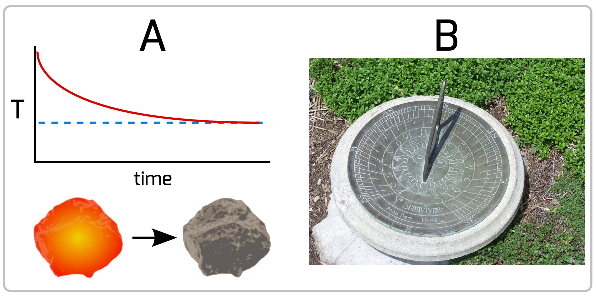

Figure 1. Two examples of natural computation. In A, a heated rock cooling to ambient temperature is used to calculate a geometric curve approaching an asymptote. In B, a sundial is used to compute time [22].

Similarly, non-opaque obstacles can be placed in the path of a ray of light (such as a laser) to reflect, refract, and filter the light. Again, these obstacles will act as constraints on the dynamical system of optics and could be configured to perform meaningful computation. For instance, a series of such obstacles could be arranged such that, depending on the position of the light source, the ray is redirected to one of two target faces, computing a decision problem where the light position and the targets have semantic meaning.

A third example might be dropping a ball in the physical system governed by gravity and collision mechanics. In this system, constraints may be physical obstacles which may be placed at various positions and angles such that a dropped ball may tumble and bounce along a path depended on the position from which it is released (not unlike a pinball machine). If a mapping is devised between the ball's physical positions and some meaningful states of a computational process, the physical obstacles could be configured to represent logical conditions or other predicates.

Finally, the physical system could be that of electrical potential energy, which drives electrons to flow through conductive materials. Constraints can be placed on this system by redirecting or impeding the current, as is done with wires and resistors. The currents can also be manipulated with doped semi-conductors, such as transistors and diodes. Since transistors are the building blocks of transistor-transistor logic (TTL) which yields logic gates (AND gates, OR gates, NOT gates, etc) and ultimately flip-flops, there can be no doubt that the constraints on this system ultimately yield a computational platform.

6 ABSTRACT COMPUTATIONAL PLATFORMS

The above examples have demonstrated a variety of physical systems that can provide varying degrees of general computation. The computational expressiveness of each system relies heavily on the degrees of freedom for setting constraints on the intrinsic dynamics. The key to computing with each of these dynamical systems is to place constraints on the dynamics such that the states of the system can support an honest semantic mapping. For instance, the constraints on electrical potential to produce logic gates are specifically designed to support the semantic mapping of predicate logic. Digital computers are so valuable because of the breadth of what can be expressed in predicate logic, though even predicate logic has its limitations.

Observing the above physical systems, some generalization may be drawn for defining the characteristics of an *abstract computational platform*. First, such a platform must have its own intrinsic dynamics. As in the above examples, these dynamics can be those of the physical world, but they do not have to be. For instance, the dynamics of a Turing Machine are not natural, though they are intrinsic to the platform and are predictable.

Second, such a platform must provide customization of constraints on its dynamics (Figure 2). Essentially, this provides a way to "program" on the platform, redirecting the dynamics to perform some meaningful computation. In other words, it is this property of a computational platform that allows a programmer to map dynamical states to semantic values and place constraints to process the semantics. In the case of a Turing Machine, this is afforded by the design of the machine's configurations and the instruction table.

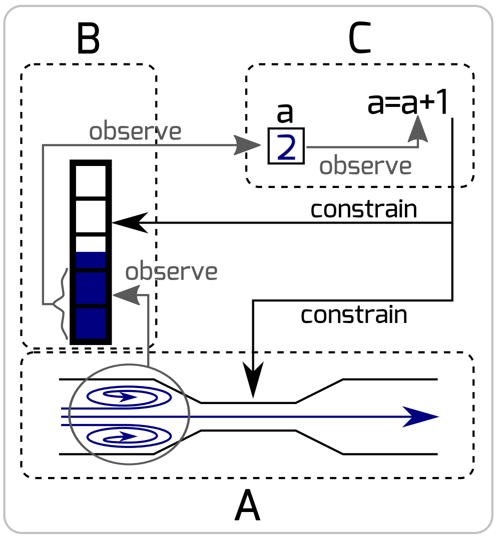

Figure 2. Abstract platforms of computation. In A, the dynamics of some flowing system is constrained to manipulate the quantity of fluid on the left hand side. This quantity is reinterpreted in B, into a real-valued number, and constrained again through discrete quantization. This discrete number is represeted in C as an integer value. Ultimately, a simple equation controls the constraints in A and B to produce the semantic outcome of the equation.

A natural result of these two characteristics is that the dynamics and the degrees of freedom on customization limit the computational expressiveness of a platform. For instance, the light-and-occlusion system might be able to compute more than time of day, but may not be as expressive as a Turing Machine. Thus, while there may exist systems with severely limited computational expressiveness (such as Searle's wall or a heated rock cooling to the ambient temperature), even these systems may serve as platforms for some few computational tasks.

Another result of this definition of a computation platform is that it naturally gives rise to the concept of layers of abstraction (Figure 3). Each computational platform exists as a single layer of abstraction and is capable of supporting computation for another layer (eg. a digital computer supports an operating system which, in turn, supports a web server). Saying that a computational platform is "computing" does not stipulate *what* is being computed. Thus, for every platform, there requires some observer to remap its states and place constraints to establish some semantic meaning of the dynamics before computation can occur. However, this observer need not be a human, but could, instead be another machine, or even another computational layer.

Digital computers represent several, layered, computational platforms before the first line of software is even available. Semiconductors are arranged to constrain the natural dynamics of electricity to emulate logic functions. These logic gates are then arranged as flip-flops which can store one of two stable voltage states. Here, the first layer reinterprets the flow of electrical current (and the state of the electrical potential across the circuit) as logical predicates. The second layer reinterprets the combination of logical predicates into binary values. While this might appear to represent a system with a discrete time step and a discrete state space, it is in fact a constrained continuous system, with each layer of constraints introducing new semantics on the previous layer. Likewise, while a register of 8 flip-flops can store up to 8 binary digits and these digits can have numerical meaning to a human, this semantic meaning is not intrinsic to the register. This same trend continues upwards through the arithmetic-logic unit and the instruction pipeline. Thus, at every layer, the states of the previous layer are remapped with new semantics (eg. voltage $\rightarrow$ binary $\rightarrow$ letters $\rightarrow$ stories).

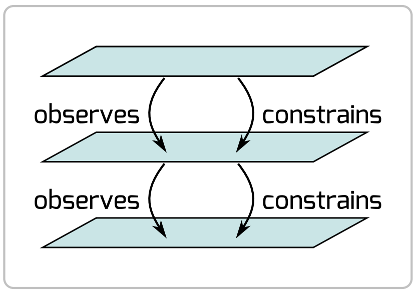

Figure 3. Abstract computational platforms can be layered, each reinterpreting and constraining the states and dynamics of the lower layers. However, the reinterpretation must be a static mapping and the constraints must be afforded by the underlying dynamics, such that not all platforms are suitable for all tasks.

However, computational layers are abstract concepts. While they have an intrinsic dynamics, those dynamics need not be natural or even deterministic. Also, for computational platforms to accommodate layering, a platform must have some notion of what is required for another platform to support it. For instance, a Turing Machine's dynamics can be described as follows:

1. Look up entry in instruction table for current configuration and current tape symbol
2. Change configuration, modify tape symbol, move tape depending on the entry in the table

Also, a Turing Machine provides to the programmer the following operations:

- Read/write tape
- Move tape left/right
- Read configuration
- Change configuration

For a TM program to run, these operations must be guaranteed by the platform. For a platform to support a TM, it would have to provide the ability to change and read the configuration of the machine, and store and manipulate information that can be represented as a linear symbol tape of arbitrary length.

7 FORMAL CONSTRAINT LANGUAGE

Under the definition of computers as layers of abstract computational platforms, programs are described by a collection of specific constraints placed upon the dynamics of the underlying platform. To match the generality of this framework, it would be ideal if there were a formal, platform independent constraint description language (CDL) which could be used to describe constraints on any platform using the elementary operations provided by that platform. Such a language would have to operate at the natural level of abstraction of whichever computational platform it is currently targeting, describing input, output, and all constraints in a natural way for that platform. Further, the language should be flexible to the wide potential diversity in implementations, including interaction rules.

7.1 Abstract State Machines

Seeking a formalism for modern software, Gurevich et al have created Abstract State Machines (ASMs) [8] to be a high-level conceptual model which expresses algorithms and other, non-algorithmic programs at their native level of abstraction. One such ramification of this idea is that data types and operations may all maintain their semantic meanings, delegating the implementation details to a lower layer of abstraction. The abstract layering approach to ASMs naturally fits with the notion of computational layers and thus may provide the basis of an ideal CDL.

Briefly, an ASM defines the state of the system at any time as a structure or group (in the abstract algebra sense) which contains all of the values and potential operations that may be performed on those values. Then, each step of the ASM transforms the state based on which of the operations is valid at each step, performing every valid operation in parallel. To more intuitively grasp the functioning of an ASM, it can be expressed as a set of conditional assignment statements. Each step, every statement is evaluated in parallel and, where the conditions are met, assignments take place to modify the values stored by the system state. To preserve the level of abstraction of the procedure, implementation details are often replaced by semantically named functions, as is regularly done in object-oriented programming, which become the elementary operations of the program. The intuition is that each of these sub-functions might be described by an ASM at their native level of abstraction, recursively expressing more implementation detail.

7.2 Interactive ASMs

While ASMs are not intrinsically interactive, they can include interaction through their abstracted function calls. In the ASM literature, two types of interaction are defined: inter- and intra-step [8, 2]. In inter-step interaction, input is injected into the state of the computer between computational steps. This can be viewed as the environment modifying the state of the computer, an idea that is undesirable in both ASMs and software engineering. The alternative is for the computer to specifically request and then receive input, deliberately inverting the input-output sequence and giving the computer control of the interaction. In interactive ASMs, all interaction (synchronous or asynchronous) is performed through intra-step queries, in which querying the environment (or another agent in the environment) replaces the standard output stream of an interactive automaton and the environment's response replaces the input stream. This querying semantics is reminiscent of Turing's Choice Machines, which could pause to request further input from a user.

The distinction between inter- and intra-step interaction is one largely of perception, as it depends on where one draws the boundary between the environment and the embodiment of the computing agent. For instance, if one considers the sensor buffer of a robot to be external to a program that the robot is running, then the program must deliberately fetch any data that is waiting there. However, the sensor buffer is a physical component of the same robot that is running the program and, by taking this slightly larger view of the system, it might appear that the environment is writing data directly to the robot's computational state. While any instance of inter-step interaction can be inverted by shrinking the program definition in some way to exclude the input ports that are directly coupled with the external environment, this also blurs the line between the automaton and its surroundings. As ASMs directly discuss levels of abstraction, this should not necessarily be a problem, as it can be argued, following the previous example, that the robot's physical body occupies a different abstraction layer from its "mind". Another way of considering the coupling is that the robot's body is a part of the environment and is interacted with by the robot's programming.

Ultimately, since the difference between intra- and inter-step interaction is largely a matter of perspective, and inter-step interaction can be performed by an Interactive Machine, there is no reason to suspect that ASMs cannot be purely mechanical. Thus, the interactiveness of ASMs is as expressive as that of Van Leeuwen and Wiedermann's IMs.

7.3 Continuous Time ASMs

There is, however, one feature of ASMs that bares some consideration for their selection as a formal CDF: the discrete time step. While continuous values might be adequately represented by the symbolic abstraction (in the same way as mathematics represents real numbers), ASMs operate in discrete time steps. However, one could conceive of a continuous time ASM, in which each parallel operation occurs in continuous time, though the properties of such a construction would have to be further explored. If ASMs could be adapted to model constraints on continuous time, concurrent, interactive dynamics, it would likely provide a satisfactory description language for constraints on any computational platform.

8 CONCLUSION

Computation has evolved greatly since the formulation of the Turing Machine in 1936. Many computational paradigms have been envisioned that reach beyond the limitations of the original conception, though, many of these must remain as abstract concepts owing to their inclusion of concepts of infinity. Other paradigms, which are represented ubiquitously in operating systems, word processors, and other embedded or interactive software, also supercede Turing Machine expressiveness. With the repetitive relaxing of the definition of computing, some have come to speculate that all of physics is inherently computational. Aiming to reserve the term "computation" for a special subset of dynamics, this paper has defined computation with a framework of abstract computational platforms.

Computational platforms provide a constructive and unified way to express the computational nature of any natural or abstract dynamical system. It has been argued that any dynamical system can be described as a computer, but every dynamical system has natural restrictions to how constraints can be placed upon its dynamics so as to accomplish meaningful computational tasks. Computation then exists as these constrained dynamics. Under this description a wall could not execute a word processor, but a light-occlusion based system could compute time.

The framework of computational platforms seeks to describe a computational system as a dynamical system, whose states maybe acquire semantic meaning and whose dynamics may be constrained to uphold those semantics. Technically, any dynamical system can be described as such, but the degree to which the constraints can be placed are dependent on each system. With less flexibility on the constraints comes less computational expressiveness.

The final requirement for this approach is to formalize a language for describing constraints for any arbitrary computational platform. It has been shown here that perhaps Abstract State Machines could serve as a useful direction, provided that a continuous time version is feasible. Further insight might be gained by exploring related models of concurrent computation, including the Actor Model [10] and Interaction Nets [11].

Ultimately, this approach aims to incorporate all conventional computing paradigms, including Turing Machines, Interactive Computing, which have been discussed here, as well as many less conventional paradigms. In particular, this approach seeks to include distributed computing paradigms (for instance, artificial neural networks [20] and Stochastic Diffusion Search [1]), each interacting with the environment and/or each other. Each of these swarms could be seen as a computational platform with intrinsic dynamics. While each swarm agent might be considered as a lower computational layer, constraints can be described on the swarm as a whole by modifying the content or protocol of their communications. By allowing this framework to describe populations of interacting processes, a bridge might be build to constructively discussing the computational capacity of the brain.

The work presented here parallels the framework for computating and information processsing discussed in [18], but with a specific emphasis on incorporating distributed, continuous, interactive, and reflective systems. Further, the work presented here views semantic mappings as crucial for distinguishing computation from other physical processes.

As much as this approach aims to include all dynamical systems under the aegis of computing, it also aims to position these systems such that their computational expressiveness is understood to be limited. While all physical systems could be said to support computation, they are not themselves computational nor are they all capable of expressing the same class of programs. Also, abstract dynamics can be said to support computation, and their computational capacity can be understood alongside their natural counterparts. With such broad applicability, hopefully, the framework of computational platforms will help to constructively focus the future discourse of computing.

9 ACKNOWLEDGEMENTS

The authors would like to convey their gratitude to the John Templeton Foundation for providing the funding for this work and to the anonymous reviewers for their helpful comments.

REFERENCES

[1] J M Bishop, ‘Stochastic Searching Networks’, in *Artificial Neural Networks, 1989., First IEE International Conference on (Conf. Publ. No. 313)*, pp. 329–331, (1989).

[2] Andreas Blass and Y Gurevich, ‘Ordinary interactive small-step algorithms, I’, *ACM Transactions on Computational Logic (TOCL)*, **V**(July), 1–55, (2006).

[3] Alonzo Church, ‘An unsolvable problem of elementary number theory’, *American journal of mathematics*, **58**(2), 345–363, (1936).

[4] BJ Copeland, ‘What is computation?’, *Synthese*, (1996).

[5] Gordana Dodig-Crnkovic, ‘Significance of Models of Computation, from Turing Model to Natural Computation’, *Minds and Machines*, **21**(2), 301–322, (2011).

[6] Dina Goldin and Peter Wegner, ‘The interactive nature of computing: Refuting the strong ChurchTuring thesis’, *Minds and Machines*, 1–26, (2008).

[7] Dina Q. Goldin, Scott A. Smolka, and Peter Wegner, ‘Turing machines, transition systems, and interaction’, *Information and Computation*, **194**(2), 101–128, (2004).

[8] Yuri Gurevich, ‘Interactive algorithms 2005’, in *Mathematical foundations of computer science*, pp. 26–38, (2005).

[9] Yuri Gurevich, ‘Foundational Analyses of Computation’, *How the World Computes*, 264–275, (2012).

[10] C Hewitt, ‘Actor Model of Computation: Scalable Robust Information Systems’, *arXiv preprint arXiv:1008.1459*, 1–32, (2012).

[11] Yves Lafont, ‘Interaction nets’, in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 95–108. ACM, (1989).

[12] Jan Van Leeuwen and Jíří Wiedermann, ‘On the power of interactive computing’, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, **14186**(201), 619–623, (2000).

[13] Jan Van Leeuwen and Jíří Wiedermann, ‘A computational model of interaction in embedded systems’, *Computer Science*, (January), (2001).

[14] Robin Milner, Joachim Parrow, and David Walker, ‘A calculus of mobile processes, i’, *Information and computation*, **100**, 1–40, (1992).

[15] Vincent C. Müller, ‘On the Possibilities of Hypercomputing Supertasks’, *Minds and Machines*, **21**(1), 83–96, (2011).

[16] Aran Nayebi, ‘Plausible hypercomputability’, *arXiv preprint arXiv*, 1–55, (2012).

[17] Howard H. Pattee, ‘The physics of symbols: bridging the epistemic cut’, *Biosystems*, **60**(1-3), 5–21, (2001).

[18] Gualtiero Piccinini and Andrea Scarantino, ‘Information processing, computation, and cognition.’, *Journal of biological physics*, **37**(1), 1–38, (2011).

[19] Wolfgang Reisig, ‘Abstract state machines for the classroom’, *Logics of Specification Languages*, 15–46, (2008).

[20] D.E. Rumelhart and J.L. Mcclelland, ‘Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations’, (January 1986).

[21] John R. Searle, *The Rediscovery of the Mind*, MIT Press, 1992.

[22] SEWilco. Garden sundial mn 2007. Creative Commons Attribution-Share Alike 3.0.

[23] A. Turing, ‘On computable numbers, with an application to the Entscheidungsproblem (1936)’, *B. Jack Copeland*, 58, (2004).

[24] J van Leeuwen and Jíří Wiedermann, ‘On algorithms and interaction’, *Mathematical Foundations of Computer Science 2000*, 99–113, (2000).

[25] Peter Verbaan, J van Leeuwen, and Jíří Wiedermann, ‘Lineages of automata’, *UU-CS*, (2004).

[26] Peter Wegner, ‘Interactive foundations of computing’, *Theoretical computer science*, **3975**(97), (1998).

[27] Jíří Wiedermann, ‘Autopoietic automata: Complexity issues in offspring-producing evolving processes’, *Theoretical Computer Science*, **383**(2-3), 260–269, (2007).