Towards a Unified Theory of Implementation [Draft]

Nick Wiggershaus

Abstract. Various conceptual approaches to the notion of implementation can currently be traced in the foundations of computing. This paper articulates a *unified* theory of implementation based on agential and use-based notions of physical computation. Two hitherto largely independently treated notions of implementation are juxtaposed: type-(A), dealing with the notion of correctness through the ascription of teleological functions to 'computational artifacts'; and type-(B), concerned with bridging the abstract/concrete dichotomy for providing an account of concrete computation. While little research has been conducted on their relationship, my analysis shows that their scope overlaps at the abstract-physical interface. Considering recent developments in the respective discourses, I show that both accounts may mutually enrich each other considerably through unification. Specifically, I submit that (A) and (B) can be unified by appealing to the conceptual machinery of use-based accounts of computation informed by the literature on material models and scientific representation. On this view, agents use putative computational systems as epistemic tools by *imputing* mathematical functions and *ascribing* teleological functions.

Keywords Implementation, Physical Computation, Scientific Representation, Correctness, Philosophy of Computer Science

1. Introduction

It sounds like a cliché, but the implementation of computation is ubiquitous. Not only are we surrounded by everyday devices such as laptops and smartphones that run our software, but computation is also at the core of foundational questions in computer science, robotics, AI, and cognitive science. Despite its ubiquity in computer science and adjacent fields, the notion of implementation is typically left informal. It is often associated with the realization, instantiation, or concretization of a plan or idea, *relating* two objects or domains with one another.[1] Considering the rapid developments in theory, technology, and areas of application of computing, various philosophical studies conceptually reconstructed what constitutes the implementation of computation in their respective fields. Confronted with a plurality of approaches and theories of implementation, one may wonder – are there common assumptions and systematic overlaps? Accordingly, this paper concerns the relationship between the arguably two most prominent clusters of implementations that have emerged over the last few decades. For tractability, I refer to these views as *type-(A) implementation* (with '(A)' for ascription/artifact) and *type-(B) implementation* (with '(B)' for bridging).

Type-(A) implementation emerged from the concerns about the verification and correctness of so-called computational artifacts like computer programs.[2] Much of the corresponding discourse

[1] Overall, there are various (pre-theoretic) understandings of 'implementation', even occurring in the domains of art, language or other affairs (Rappaport 2005).

[2] After the fiercely held *verification debate* in the late 1980s and 1990s in the *communications of the ACM*, it was apparent that the field would benefit from a philosophical underpinning of the notion of verification and correctness. For a collection of some of the key contributions to verification see Colburn et al. (1993). For a critical assessment of the debate see McKenzie (2004, 197-218).

is couched in terms of *function ascription* (in the teleological sense) and pertains to the relation between different abstract levels or structures. Type-(B) implementation, on the other hand, concerns the nature of computation *qua* physical process in material systems (including both natural and artificial ones). This notion is paramount to determine if systems like brains or even the whole universe compute. Virtually all (B)-accounts share the idea that sequences of formal/abstract computational states map to the evolution of a physical real-world system. One of the main differences between type-(A) and (B) are their fundamentally different underlying equivalence relations. According to the former, implementation is a relation that links abstract objects to other abstract objects or structures; the latter connects abstract objects with physical systems and thus has similarities to notorious problems of applied mathematics and scientific representation. In other words, neither type-(A), nor type-(B) discourse has fully embraced each other's domains of application. Yet to address fundamental questions such as

> How do (human) agents use and manipulate physical systems for computation? Is there a principled difference between mere computation and program execution? To what extent does type-(A) implementation apply to non-designed systems? How can we judge correct physical execution and usage of programs from incorrect ones? What kinds of things are programs – are they abstract or concrete?

one requires the combined insights of both type-(A) and type(B) implementation. That's why future research in computing would benefit from a theory of implementation incorporating the main insights of the two notions of implementation. I take this as a motivation to engage in such a project.

Subsequently, the paper unfolds like so: The main goal of this paper is to shed light on these questions by juxtaposing type-(A) and (B) implementation. For so doing, I first take a closer look at type-(A) implementation in section two. Section three then introduces type-(B) implementation by scrutinizing candidate solutions to the so-called *Problem of Implementation* and their relation to the philosophy of applied mathematics. Thereafter, in section four, I juxtapose both types of implementations by focusing on their respective most salient features. Whilst at first sight applying to different domains of computing and having different purposes, I claim that they are in fact conceptually compatible. In section five, I suggest that the unification of the previously introduced notions of implementation is possible by relying on the conceptual machinery of material models and scientific representation. Based on these insights, the resulting synthesis suggests a view according to which putative computational systems are *epistemic tools*, i.e., material artifacts used by agents for computation. When using material artifacts (akin to material scientific models) for computation, agents impute mathematical functions and ascribe teleological functions to engage in a form of object-based reasoning. I shall refer to this view as, a *unified theory of agential implementation* (UTAI). Lastly, I conclude.

2 Type-(A) Implementation

2.1 A brief introduction to Implementation in Computer Science

The *Oxford Dictionary of Computer Science* provides a useful characterization of implementation to start with

> **Implementation:** "[t]he activity of proceeding from a given design of a system to a working version (known also as an implementation) of that system, or the specific way in which some part of a system is made to fulfil its function.".

The *relation* between design and its working version may be applied to various so-called *Levels of Abstraction* (LoA) Floridi (2008), Primiero (2019). As such, application in computation is wide-ranging and comprises examples such as 'the implementation of an algorithm in a high-level programming language', or 'the implementation of machine code instructions in a real-world computer'. Such 'level-talk' is frequent in computer science – a concrete textbook example of various implementation stages of a program is pictured in Fig. 1. Instances like these may be generalized and accordingly culminate in a view as depicted under label (b) in Fig. 1: a (stored-program digital) computing system is typically composed of various LoA forming a computational hierarchy.[3]

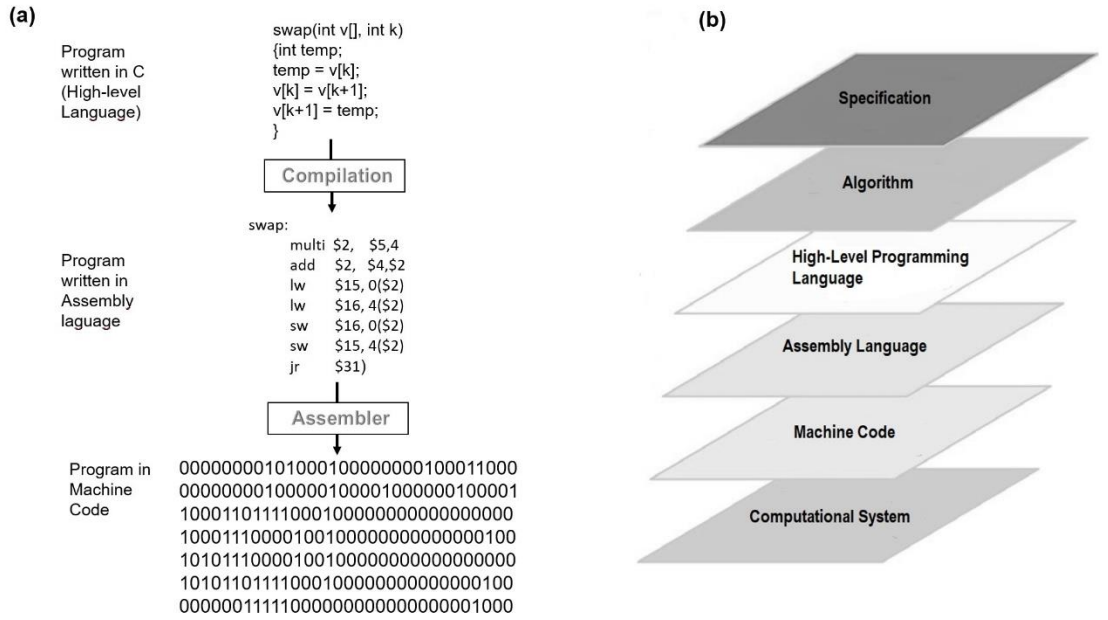

Fig. 1: Different depictions of the computational hierarchy. **(a)** A concrete instance of the different LoA and stages of implementation of a program written in C (example adapted from Patterson and Hennessy (2014, 15)). **(b)** Generalized image of typical LoA of a computer program.

At the bottom, one finds the *physical* computing system. The computing system is constituted of various material components and their specific arrangement. If set up and configured correctly, the system may execute a predetermined series of concrete computations. At the top of the hierarchy, one may find the most *abstract* level, often thought to be the (formal) specification of a program. 'Abstract', in this context, refers to the degree to which language features are divorced from any specific hardware details (Scott 2009, 111). In this sense, higher LoA entail fewer details about the underlying computational system. In the same vein, the process of abstraction can,

[3] See Hennessy and Patterson (2014, Ch. 2) for a fully worked-out textbook example. See also Scott (2009).

roughly put, thus be understood as the inverse of the operation of implementation. At the same time, the computational objects corresponding to the different LoA may be abstract in a second sense though. Viz., they may be referred to as abstract – as opposed to being concrete, material, or physical – since as strings of symbols they have no causal relations acting upon them. This second sense of abstraction becomes relevant at the abstract-physical interface.

Importantly, what follows from this brief discussion is this: Whereas higher LoA are related by what might be called symbolic implementation, the last implementation stage is qualitatively different. For instance, the program written in C in Fig. 1 is a particular symbol structure that is translated (i.e., compiled) into assembly language.[4] When descending the computational hierarchy down to machine code, implementation is the relation that obtains between different abstract strings of *symbols*. Ultimately, at the abstract-physical interface, a different kind of relation is required – one that relates abstracta with concrete states of the putative computational system. In what follows, I take a closer look at the philosophical characterizations of these implementation relations.

2.2 Philosophical Characterizations

Perhaps symptomatic for a more general tendency of computer science, type-(A) implementation mostly focuses on the relation of upper LoA. Here, one central aim is to determine the *correctness* of the various implementation stages. To meet the normative notion of correctness, two conditions are generally called for. On the one hand, a *formal specification* for the program and on the other hand a formal definition of the *semantics* of the programming language. A program is then demonstrated to be correct if there is a formal proof that the semantics of a program is consistent with the program's specification. Arguably, the first philosophical account in this respect is due to Rappaport (1999). He describes implementation as semantic interpretation (1999, 2005),

> **Implementation as semantic interpretation:** An object is an implementation of some syntactic domain *A* in medium *M iff* it is a semantic interpretation of a model of *A*,

i.e., a relation between semantics and syntax of different LoA. Rappaport claims that any correspondence between two domains where one is used to grasp the other is 'semantic correspondence'.

Rappaport's approach is supposed to allow for both mere 'translations' of one programming language into another (symbolic implementation) and even for the qualitatively different case at the abstract-physical interface (i.e., the relation between the bottom layers of the computational hierarchy). A program written in a high-level programming language may not be immediately implemented in a physical system then, but the so-called 'correspondence continuum' is supposed to ensure their connection. The program must go through a series of translation processes, where each time a level that previously acted as a semantic domain then turns into a syntactic for another level below. At last, the 'implementation cascade' bottoms out at the physical level, which then provides the semantic domain for all the previous levels.

[4] Alternatively, programs can also be translated through an interpreter, such that the source code is directly executed (line by line), without previously having been compiled into machine code.

However, concerns were voiced about the way Rappaport employs his notion of semantics as a given, raising questions about whether an independent semantic account is required.[5] While the notion of semantic interpretation adequately describes *that* implementation requires semantics, it lacks the rigor to describe *how* these semantic features come about. The semantic approach does not explain how the physical level obtains its semantic capacities to act as the bedrock for the entire computational hierarchy. Therefore, two improved accounts were suggested.

First, to account for an independent and external account of semantics, Turner turned to the technical-artifact-literature and adopted the notion of *function ascription* (2012, 2014, 2018). Originally, the conception of technical artifacts and their functions is supposed to cover intentionally produced everyday objects like screwdrivers, coffee-makers, and trains (see, for instance, Kroes 2012). They are said to have a 'dual nature': Next to their respective causal/structural properties, this class of artifacts bears normative or teleological features. The function of a coffee maker is to brew coffee; a broken or malfunctioning coffee maker does not work correctly. Only when 'the how' (the structural properties) realizes 'the what' (functional properties) in the right way, can one claim to have a properly working coffee machine. By thus transposing the core insights of the technical-artifacts framework to computational entities, the conception of *computational artifacts* was born (Turner 2018). Accordingly, computational artifacts like programs exhibit a *function-structure duality*, with

> **Implementation as function-structure relation:** The relation between specification (function) and the structure of the (computational-)artifact.

Importantly, artifact function here is an intentional notion, derived from the use plan formulated by designers. The functions are bestowed to artifacts based on the intentions and desires of human agents or an epistemic community.[6] In the computing context specifically, it is the intentional notion of specifications of programmers that provide criteria for correctness and malfunction.

Second, following these developments, Primiero (2020) addressed issues with both the implementation as semantic interpretation and implementation as function-structure relation. The problem with both is that they merely provide an account of implementation for any two neighboring levels and not the *entire* computational hierarchy. For instance, to eventually reach the bottom of the hierarchy (the physical computing system), Turner's version of the computational artifact approach relies on repeatedly flipping the function-structure pair; the process must be repeated for every level in the computational hierarchy. While the structure-function relation may ensure correctness between any two LoA, Primiero argues that the view fails to establish the desired transitivity of correctness throughout the entire computational hierarchy (i.e., between more than just two LoA). The result is an impoverished characterization of miscomputation.

For this reason, Primiero advanced a notion of implementation that considers multiple LoA of the computational hierarchy, where each LoA is constituted by an epistemological construct and ontological domain,

---

[5]For a more detailed summary of these arguments see Primiero (2019, 207f) and (Turner&Angius 2020).

[6] The role and nature of functions are a vexing issue. Many theories of function emerged in the context of biological traits, but have subsequently inspired accounts of artifact functions. Unlike organisms though, artifacts are purposefully created and hence need to account for human intentions. The question is how to balance intentional and causal (non-intentional) features. For a short survey of the debate see e.g., (Preston 2018, §2.3); for arguably the most detailed account, consult Houke's and Vermaas' (2010) ICE-theory (incorporating elements from intentional, causal-role, and evolutionist function theories).

Implementation as the relation of LoA: An implementation *I* is a relation of instantiation between pairs composed by an epistemological construct *E* and an ontological domain *O* of a computational artefact**.**

The idea of the *EO*-pairs here is congruent to the function-structure relation, as the epistemological levels provide "the structure to understand the behaviour of the ontology" (Primiero 2020, 194). Yet, this view of implementation enables a more fine-grained notion of correctness because it differentiates between different layers/EO-pairs of the computational hierarchy. Accordingly, one may e.g., define notions such as *functional correctness* or *procedural correctness*, and a corresponding detailed taxonomy of miscomputation (cf. Fresco&Primiero (2013), Primiero et al. (2015), Primiero (2019, 211-12)).

3. Type-(B) implementation

3.1 The Bridging Problem

The second notion of implementation is, fundamentally different from the previous one. The crucial difference of the type-(B) discourse is that we are no longer exclusively in the logico-mathematical realm. Instead, we are dealing with the relation between mathematical objects (of computability theory) and the *physical* world – a relation raising notorious questions in the philosophy of science and applied mathematics (Wigner 1960, Steiner 1998). The characterization of physical computation is a special instance of such a relation. I first introduce the general problems associated with the relation of mathematical- and physical objects, before shedding light on the particular issues of computation (in the next subsection).

Following Contessa, I refer to the general issue as the *bridging problem*, "the problem of how to bridge the gap between [abstract] models and the world" (2010, 516). Put differently, we need to explain the nature and origin of the relation between two ontologically different categories – viz., how mathematical objects relate to the physical.

Typically, such a relation is construed as a mapping relating mathematical and physical structures. However, one of the main issues with this 'mapping view'[7] is that ordinary functions only obtain between the domains of set-theoretic structures – yet physical objects are not set-theoretic structures. Thus, without further qualifications, maintaining that morphisms obtain between (abstract) mathematical structures and physical objects is not a well-informed option (van Fraassen 2008, 237f). A solution to the bridging problem hence requires overcoming a second obstacle. In what follows, I refer to the issue as

Structure Generation Problem: Explaining how material systems may offer set-theoretical structures.

The seemingly intuitive idea that physical systems somehow just bear or instantiate a unique structure *S* is highly contentious because of *Newman's Objection*. According to the latter, the mapping between a set-theoretic structure and a physical object might be trivialized, since the latter fails to display a unique or privileged structure. Defining a set-theoretic structure as an ordered tuple: $S = \langle \mathrm{D}, R_1, R_2, \ldots \rangle$, where *D* is the domain of the objects x_i and R_i are the relations on *D*, the claim

[7] Sometimes when thinking of models and representations in terms of such a formal relation between structures (e.g., a mapping), the issue is also referred to as mapping view. (Pincock 2004, Batterman 2010, Bueno & Colyvan 2011).

goes that one can always gerrymander D and R_i in such a way that they match an arbitrary structure S. For given there are enough n basic objects $x_1,\ldots, x_n$ in the system (such that the cardinality of the corresponding domain is sufficiently large), then "[…] a system of relations between its members can be found having any assigned structure compatible with the cardinal number of [S]", (Newman 1928, 140), where S is an arbitrary structure.

So, in order to generate a suited structure, one needs to engage in two tasks. First, one must specify the domain D of objects x_i; second one must specify what the relations R_i between these objects are. It is wide consensus that both tasks are all but straightforward though. The following quote exemplifies the issues. Bueno and Colyvan, for instance, remind us about the challenges when trying to determine the elements in a domain D:[8]

> "Put simply, the world does not come equipped with a set of objects (or nodes or positions) and sets of relations on those. These are either constructs of our theories of the world or identified by our theories of the world. Even if there is some privileged way of carving up the world into objects and relations […], such a carving, it would seem, is delivered by our theories, not by the world itself. What we require for the mapping account to get started is something like a pre-theoretic structure of the world (or at least a pre-modeling structure of the world)." (Bueno & Colyvan 2011, 347).

3.2 The Problem of Implementation

When it comes to physical computation, we are confronted with a special instance of the bridging problem, the so-called *Problem of Implementation*. In virtually all cases, physical computation is characterized in terms of the mathematical theory of computation and a "mathematics first" attitude (Curtis-Trudel 2022), according to which some computational formalism of computability theory is the starting point for the definition of physical computation. Simply put, the problem of implementation needs to describe how a computational formalism hooks on to the world.

Let me first have a brief look at computational formalisms. Computational formalisms may be defined in a large variety of ways. The computer science literature features at least two main kinds of computational formalisms (Turner 2018, 190):[9]

1. *Programming languages*, like C, Python, etc.
2. *Machine Models*, like Turing Machines (TM), Finite State Machines (FSM), etc.

In the following, I use the term 'model of computation' *Mc* for both. Models of computation are logico-mathematical formalisms that enable us to encode an abstract sequence of computations through a programming language, a machine table, a transition function, and so on. For instance, formally, the concept of a Turing Machine can be characterized as a quadruple $TM = (Q, \sum, m, \delta)$, where Q is a finite set of states q; $\sum$ is a finite set of symbols; m is the initial state $m \in Q$; δ is a transition function that determines the next move δ: $(Q \times \sum) \rightarrow (\sum \times \{L, R\} \times Q)$. TM's transition function δ maps from computational states to computational states (De Mol 2021). Put differently,

[8] And even if one had successfully determined what constitutes a domain's objects, there remains the pressing problem of the right set of relations R_i among them. As Psillos aptly puts it, "[…] the structure of a domain is a relative notion. It depends on, and varies with, the properties and relations that characterize the domain. A domain has no inherent structure unless some properties and relations are imposed on it. Or, two classes A and B may be structured by relations R and R' respectively in such a way that they are isomorphic, but they R may be structured by relations Q and Q' in such a way that they are not isomorphic." (Psillos 2006, 562)

[9] The description of computational formalisms is inspired by a similar presentation in Rescorla (2013).

transition functions like δ, or corresponding notions in theoretically equivalent models of computation, allow for the encoding of a sequence of computations.

Now, the basic problem underlying virtually every characterization of physical computation is then providing a characterization of how *Mc* and the computational state transitions described by the corresponding transition function δ latch onto physical systems. Accordingly, formal abstract computational state transitions need to 'mirror' the evolution of the physical states of a material system. Often the situation is depicted in a diagram as seen in Fig. 2[10]:

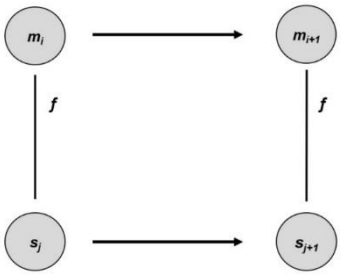

Fig. 2: A typical depiction of the core idea underlying physical computation.

Subsequently, many scholars working on physical computation agree that there are at least two main issues, albeit related, concerning implementation: Not only do they want to demarcate those systems which seemingly compute (e.g., laptops and brains) from those which don't (rocks), they'd also like to determine which computation is executed rather than another. Closely following suit with (Sprevak 2018 and Ritchie&Piccinini 2018), the problem of implementation concerns:

> **COMP** The conditions under which a physical system is computing.
>
> **IDENT** The conditions that specify that a computational system implements one computation rather than another.

On this view, the implementation of a specific computation is constituted by two features. While COMP determines *that* a given physical system is computing, IDENT concerns the question of *what* is computed. Both COMP and IDENT are intertwined in a way that makes it difficult to understand the latter without at least some preliminaries of the former.

Virtually all potential answers attempt to solve the problem of implementation by adhering to an equivalence relation between mathematical structure and physical system in terms of a mapping relation. This thought is reflected in the so-called *simple mapping account* (SMA)[11] and was, among others, articulated by Putnam (1987). The main idea is based on a simple mapping between abstract formalism M_C and a somehow given physical structure S_C. Accordingly, the SMA postulates that a physical computing system S_C implements a computation *iff:*

[10] To the best of my knowledge, one of the first instances in the philosophical literature of this diagram can be found in Cummins (1989).

[11] The term was introduced by Godfrey-Smith (2009) and made prominent by Piccinini (2015); note the similarity to the 'mapping view' in applied mathematics and scientific representation introduced in the previous section.

Simple Mapping Account (SMA)

1. There is a function f mapping the states s_j of S_C to states of M_C, such that
2. Under f, the physical evolution/state transitions $s_j \rightarrow s_{j+1}$ are morphic to the formal state transitions $m_i \rightarrow m_{i+1}$ of M_C (specified by δ), where $f(s_j) = m_i$.

The approach is elegant and appears to straightforwardly capture what's pictured in Fig. 2; the SMA has basically become the key ingredient of all approaches to solving the problem of implementation that were to follow.

3.3 Accounts Beyond the Simple Mapping Account

However, it is widely agreed that the SMA bears two major problems. First, the SMA is charged with trivializing the notion of concrete computation. As pointed out by Copeland (1996), the SMA essentially suffers from Newman's Objection. Given any kind of physical system S_C, one may carve out the physical states of S_C in whichever arbitrary way such that they are morphic to M_C. In other words, structure is too cheap to come by – any arbitrary computational description (like a *hello world* program written in C) with a sequence of computational states $m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_i$ can be mapped on an arbitrary evolution of physical state transitions (of, say, a rock). So, according to the SMA every macroscopic object realizes all kinds of computations, a position known as unlimited pancomputationalism – an unsatisfying answer to COMP and IDENT.

Moreover, there is the issue of computational identity IDENT – the question of which of the multitude of computational profiles that simultaneously apply to a system is implemented.[12] The claim goes: Even if there was somehow a unique computational structure to begin with, structure alone would fail to deliver an account of basic computational individuation, as one needs extra ingredients to discern which computation is carried out. Frequently, the case is exemplified with a system S_C implementing logical duals, like a logic gate with the following behavior:

Input_A	Input_B	Output
5V	5V	5V
5V	0V	0V
0V	5V	0V
0V	0V	0V

Table 1: Logic gate

Under the assignment 0V → F, 5V → T, the truth table (Table 1) of the logic gate corresponds to an AND-gate. However, by flipping the labels (0V → T, 5V → F) the same device implements an OR-gate. Now, the issue is that the same system appears to simultaneously implement multiple computations (conjunction and disjunction) at once. The phenomenon generalizes to many other gates and computational systems. Fresco et al (2021) recently called such physical systems *multiply specifiable* if they possess at least two logically non-equivalent labeling schemes when using the same labels (e.g., "F" and "T"). Consequently, the question arises, which of the two labeling schemes is the preferred one?

[12] To the best of my knowledge, the problem was first mentioned in Shagrir (2001).

In response to the triviality arguments and multiple specifiability most physical computation/type-(B) implementation accounts have been trying to impose constraints in the form of extra conditions onto the SMA:

Extended Mapping Account (EMA)

1. There is a function f mapping the states s_j of S_C to states of M_C, such that
2. Under f, the physical evolution/state transitions $s_j \rightarrow s_{j+1}$ are morphic to the formal state transitions $m_i \rightarrow m_{i+1}$ of M_C (specified by δ), where $f(s_j) = m_i$.
3. f is constrained by extra conditions.

Since the formulation of the SMA, a plethora of potential candidate constraints/conditions and corresponding accounts have emerged from the discourse of physical computation. A variety of different EMAs recognized that the material implication of physical state transitions (SMA's second condition) is too permissive, allowing for too many computational structures. Subsequently, many proponents tried to spell out the conditions that allow one to delimit or single out a privileged structure. Accordingly, some argued that putative computing systems need to meet the condition/constraint of having *counterfactual* state transitions (Copeland (1996)), i.e., roughly put if the system S_C had been in a physical state that maps onto m_i, it would have evolved into a state that maps onto m_{i+1}. Others formulated similar requirements in terms of a suited *causal* structure (Chalmers (1996), Scheutz (1999)) or *dispositional theories* (Klein 2008). Counterfactual constraints ensure that the mapping f applies to *all* of M_C's potential execution traces and not just, as previously the SMA, a single particular one $\delta(m_1, x)$ (where x is some input). As such, counterfactual constraints are an important feature of type-(B) implementation and I will discuss their relation to type-(A) implementation further down below.[13]

4 Juxtaposing (A) and (B)

Let me step back here for a moment and assess what we have learned so far. The previous two sections familiarized us with type-(A) and type-(B) implementation. The takeaway is that two largely separated discourses aim to characterize implementation:

> **Type-(A) implementation:** Enriched with semantics, computer scientists may use type-(A) implementation to evaluate the correctness of computational artifacts by normative requirements (i.e., the specification) of the stakeholders (programmers, users) involved. In this context, normative judgments pivot on the notion of teleological function ascription. However, since this implementation relation is formulated in terms of the formal and deductive methods of symbolic implementation it falls short of adequately responding to the bridging problem/problem of implementation and may only apply to upper LoA.

[13] One further prominent refinement of the SMA is the *mechanistic* account of computation (e.g., Milkowski (2013), Fresco (2014), Piccinini (2007, 2015)), according to which computation must be implemented in specific computational mechanisms. In so far as mechanisms have a causal structure or are said to have counterfactual state transitions, the mechanistic account can be interpreted as a refined version of the previous EMAs. Yet other approaches focused on answering the problems associated with IDENT (cf. Lee (2020) for an overview). According to so-called *semantic* theories of computation, content is essential to computational states; in other words, computation is always about something. Shagrir (2001, 2022), for instance, maintains that such semantic elements determine a privileged labeling scheme of whichever computational system at hand.

> **Type-(B) implementation:** Directed at the formal characterization of physical computation, type-(B) implementation concerns one implementational stage only – the abstract-physical interface. The theoretical framework underpinning virtually all characterizations of concrete computation is the idea that there is a mapping $f: S_C \rightarrow M_C$ that bridges the gap between abstract computational formalism (e.g., symbolic machine code) and the dynamic evolution of the physical states of the putative physical computing system.

Now, at first glance, type-(A) and (B) seemingly just address different issues. However, such segregation potentially becomes problematic for computational phenomena that entail multiple implementation stages and hence may require both theories of implementation. Arguably the most prominent examples in this respect are computational artifacts like computer programs whose implementation stages spread over the entire computational hierarchy (cf. Fig. 1).

What's at stake is this – if we want to advance beyond piecemeal approaches toward a coherent scheme, we require a less fragmented understanding of the implementation of various computational phenomena than today. So, in virtue of having a singular systematic framework, I lay the groundwork for presenting a novel theory of implementation by first juxtaposing the most salient features of (A) and (B). For so doing, I limit my focus on the nature and origin of the different mapping relations; the Problem of Structure Generation; and teleological function-ascription. The following three subsections elucidate these salient features in more detail before presenting a unified approach in sect. §5.

4.1 The Origin and Nature of *f*

The first salient feature concerns the role, nature and origin of the different equivalence relations underlying both theories of implementation. Let me briefly begin with type-(A) first. My previous analysis has shown that type-(A)'s main assumption is that a symbol structure of a higher LoA needs to be correctly translated into a symbol structure corresponding to a lower LoA. What then counts as viable implementation between abstract structures is determined by the *agreed-upon* semantics. Since we are concerned with the relation between abstract symbol structures, we do not run into the pressing problems associated with the bridging problem. To establish a relation between different structures (LoA), the type-(A)-implementationist 'merely' requires some form of semantic interpretation, else it remains unclear how the different structures are supposed to be mapped into one another. While a lot more can be said about the precise characterization of the semantics, what suffices for the present juxtaposition is that the provenience of the mapping *f* essentially hinges on the stipulations and conventions regarding the semantics made by the designers and programmers.

Regarding type-(B), matters are less straightforward – here one deals with a cross-categorical relation. Even though, the EMAs of type-(B) literature have brought forward various kinds of constraints on the implementation relation *f* the metaphysical nature of the mapping relation does typically not take center stage in the discourse of physical computation. Exemplary is a statement by Chalmers, stating that

> "[t]he definition of implementation does not appeal to any specific mapping relation: rather, it quantifies over mapping relations, which can be any function from physical states to formal states.

> I also do not know what it is for a relation to have metaphysical commitments." (Chalmers 2012, 231)

Similarly, Sprevak stresses that it is a "strategic error" to focus on the metaphysical nature of *f* (Sprevak 2018, 176). Remaining silent about the metaphysical nature of the mapping may, at least for the current project, come with the cost of an impoverished or partially incomplete picture of implementation. So, what are the options?

In principle, there are several ways in which the nature of the relation can be couched. One strategy is to utilize solution attempts to the bridging problem from other domains (e.g., the philosophy of applied mathematics). For instance, in the context of computing, Colburn (1999) attempted to map out some of the solution space. He suggested that forms of monism, dualism, or a pre-established harmony from other philosophical areas are seemingly all viable options. However, at a closer look, most of these attempts have been deemed notoriously troublesome for the original formulation of the bridging problem. When examining some of these options in more detail, Curtis-Trudel (2022) ruled out many of the options Colburn had originally thought of. Instead, it was shown that the problem of implementation *necessarily* requires a mapping that bridges the gap between a material system and an abstract computational structure. Following that analysis, only two main options remain: On the one hand, a naturalized (here, mind-independent) version of *f*, and on the other hand, a mind-dependent version.[14]

Note that, in principle, both options are compatible with the assumptions of type-(A) implementation. However, as I will argue, the philosophical plausibility of a naturalized relation can be ruled out on the ground of the insights of the literature on scientific modeling and representation. The crucial commonality between scientific representation and implementation of computation is that both essentially require a mapping that relates mathematical structures to a physical substrate. Accordingly, one may apply the results with respect to the bridging problem from one domain (modeling) to the other (computing). What counts for scientific models, *mutatis mutandis* applies to computing devices. For the sake of successfully appropriating these insights, I provide some of the here necessary background below.

According to what Giere (1999) calls the *representational conception*, scientific models are used by scientists for the purpose of representing some (real-world) system, where the latter is commonly referred to as target systems *T*. Scientists use models and their representational capacities for drawing various sort of conclusions about the target system (e.g., explanation, prediction, confirmation); a practice known as *surrogate reasoning* (Swoyer1991). Accordingly, scientific representation is characterized as the relation *f* between a model *M* and its dedicated target system *T*, such that $f: T \rightarrow M$. Importantly, both models and targets then come in various 'ontological flavors'. On the one hand, one typically distinguishes between (i) material and (ii) theoretical models.[15] On the other hand, target systems are either (a) real-world systems or (b) hypothetical scenarios.[16] Consequently, various modeling scenarios result from the possible combinations of (i)-

[14] If one assumes that the relation $f: S_C \rightarrow M_C$ is naturalizable, (i.e., the mapping is simply reduced to facts about the world) then we may regard computation as a mind-independent physical process.

[15] The term model denotes a heterogenous collection of things – models come in the form of descriptions, as material objects, or as abstract (mathematical) objects. For an extended list of 120 types of models see (Frigg 2022, Ch. 16).

[16] Weisberg, for instance, investigates the case of hypothetical modeling (2013, 121-134), where models may represent nonexistent targets, possibilities, or impossible targets (e.g., models where the targets are perpetual motion machines or multiple sexes populations).

(ii) and (a)-(b). As such, the modeling relation is – like the problem of implementation – a special instance of the bridging problem.[17]

Subsequently, Philosophers like Suárez (2003) and van Fraassen (2008) criticize the naturalized attempts of representation, namely that the mapping *f* between model and target reduces to a factual, mind-independent relation since it flies into the face of the bridging problem. Without answering the Structure Generation Problem it is highly contentious to defend a somehow naturally occurring mapping relation. Given that naturalized accounts about mappings fail to elucidate how and when such a relation comes about without explaining the assumption of some privileged preconceived structure, they are moot.

Faced with the inadequacy of naturalized accounts of representation, philosophers of science nowadays commonly agree that scientific representation is contingent on human *agents establishing* a mapping to the intended target. The upshot of such a use-based approach is that the relation between model *M* and target *T* does not simply obtain 'naturally', i.e., without the decisions, conventions, and stipulations of some scientists. In consequence, van Fraassen for instance, formulated his *Hauptsatz* of scientific representation, stating that "[*t*]*here is no representation except in the sense that some things are used, made, or taken to represent some things as thus or so.*" (van Fraassen 2008, 23). However, to fully appreciate such imputed mappings, we still need a story of what structure they are supposed to map onto. I will deal with this issue in the next section.

4.2 The Structure Generation Problem

The second salient feature concerns the necessary structures for the mappings. As already briefly discussed in the previous subsection, the corresponding privileged computational structures on higher LoA are the result of the chosen computational formalism and the corresponding transition function δ devised by the programmers (Turner 2021). Again, the much more philosophically troubling case awaits us at the abstract-physical interface. For remember, to avoid the bridging problem occurring at the bottom of the computational hierarchy, we need a philosophically plausible mechanism of how to generate a structure $S = \langle D, R_1, R_2, ... \rangle$ for a material computing system to establish the relation $f: M_C \rightarrow S_C$. As we have seen, structure generation should entail two steps (see §3.1): (a) determining the elements (i.e., putative computational vehicles) in a domain *D*, and (b) determining the relations R_i (of those vehicles) in the domain *D*.

In what follows I claim that the already familiar process of *information hiding* (essentially a specific approach to abstraction) from computer science may be equally well employed at the abstract-physical interface. As Colburn and Shute aptly remind us

> "Any science, including computer science, succeeds by constructing formal mathematical models of their subject matter that eliminate inessential details. Inasmuch as such details constitute information, albeit irrelevant information, we might call such elimination *information neglect*. It is our contention, however, that computer science is distinguished from mathematics in the use of a kind of abstraction that computer scientists call *information hiding*. The complexity of behaviour of modern computing devices makes the task of programming them impossible without abstraction tools that hide, but do not neglect, details that are essential in a lower-level processing context but inessential in a software design and programming context." (Colburn & Shute 2007, 176; my italics)

[17] For instance, assuming that M is a theoretical model relying on mathematical structure as a representational vehicle, one needs to specify how the parts of the structure are mapped to the physical make-up of the target system. Put differently, the representational relation f needs to bridge the abstract-concrete dichotomy.

'Information hiding' as abstraction allows us to omit inessential details (e.g., to improve programming practice) when creating a computational structure but enables us to later retrieve the hidden information and fill in the required details at the stage of implementation again. Abstraction as information-hiding is an already familiar tool and largely applied to higher LoA in the computational hierarchy. However, what is paramount about the omission-of-details strategy is that it may also be applied to the structure generation of *physical* objects. As such, I claim, it provides a solution to the structure generation problem (and hence the bridging problem) in the context of computation.

In fact, we already briefly encountered an instance of structure generation and information hiding that is well suited for discussion: the standard case of the logic gate (Table 1). For remember, first, a simple, albeit physical description of the components of the logic gate was provided (in the form of a table). Build into this simple and already highly idealized description was the choice to consider only specific physical magnitudes (in this case, voltages) of specific components ($Input_A$ and $Input_B$); these choices constitute both the elements in our domain *D*. In a second step, further physical details need to be omitted and be replaced with a chosen assignment like 0V → F, 5V → T (i.e., a specific labeling scheme to fix (IDENT)) to determine the relations R_i.

The idea of omitting or subtracting (the physical) details from an *agreed-upon description* of a system to allow the scientists to carve out a *unique* structure $S = \langle D, R_1, R_2, ... \rangle$ can be generalized. In their so-called *extensional abstraction account*, Nguyen and Frigg (2017) formalized the procedure just discussed with the logic gate. In a nutshell, they argue for some form of omission of information when tackling the bridging problem using abstraction. Formally, a mathematical structure S_M only links to a physical target system *T* if, previously a particular description of the given (material) system is, through abstraction, transformed into an *extensional description*. An extensional description does not explicitly refer to physical magnitudes (in the context of computing, e.g., a truth table). Extensional descriptions can be created by hiding information. In order to generate a structure *S* of a physical object, *agents* need to decide on a domain *D* and their respective elements. Thereafter, they need to determine the relations R_i among those elements. Once certain choices about the elements in the domain *D* and their relations R_i are *agreed upon* and *held fixed*, the thus generated extensional description gives rise to a purely set-theoretic structure S_T.

As such, information hiding (e.g., in the form of the extensional abstraction account) enables us to generate a suitable computational structure that subsequently suffices to be related to a theoretical computational structure that $f: S_C \rightarrow M_C$. Importantly, we can, in principle, apply the strategy of coming up with a physical description and omit inessential physical details to *all* kinds of material systems, simple or complex, digital or analog. Scheutz's (1999) work, for instance, sheds light on more complex cases such as resistors, analog circuits, and full-blown electrical circuits.[18]

[18] Whilst Scheutz does not explicitly frame his discussion in terms of the here sketched agential structure generation approach, he engages in the activity of omitting physical details from a previously chosen domain. As such, there is no feature in the information-hiding approach that would limit its scope to what is commonly referred to as 'designed' or 'artificial systems' only. While in the case of the garden-variety digital component it is deemed obvious which physical magnitude is supposed to count as a computational vehicle (typically the flow of electrical charge), we may also bestow previously considered 'natural objects' with the intended proper function to compute. The difference when dealing with such 'natural objects' is that prima facie, there might be no common knowledge or no obvious choice as to which physical features may act as computational vehicles, whether they offer a high degree of reliability and so on. Yet, as shown by the research of the unconventional computing community (see Adamatsky (2021) for an overview), one may use chemical reactions, slime molds, or quantum effects as vehicles for computing.

The upshot is that without the agents' description, dependent on specific decisions and conventions, there would be no privileged computational structure; no agents, no unique structure, no mapping!

4.3 Teleology

The last salient feature concerns teleology. Accordingly, a unified theory of implementation needs to contain ingredients that allow one to check whether a program (or any given sequence of computations) is implemented and executed correctly at every LoA. Section §2.2 already described how type-(A) accounts addressed concerns about correctness via teleological function ascription. However, in contrast, remarkably few studies in the type-(B) camp have been designed to consider the normativity of physical computation. Without such normative features one cannot address the verificationist-debate and account for the intricate correctness criteria concerning computer programs and software systems that are required for their implementation. Piccinini's (2015; 2020) *teleological version* of the mechanistic account (cf. also Coelho Mollo 2018) is one of the few exceptions that allows for miscomputation in the first place.[19]

In keeping with the functional-mechanistic account, physical computation is said to be the transformation of some (medium-independent) computational vehicle in accordance with a *rule*.[20] Rules determine what should be computed. Since they can be violated, miscomputation occurs, if the computational mechanism malfunctions (i.e., violates the rule). Which rule a computational mechanism should follow is determined by the so-called *goal-contribution account* of teleological functions. Accordingly, "[a] teleological function (generalized) is a stable contribution to a goal (either objective or subjective) or organisms by either a trait or an artifact of the organism.", (Piccinini 2015, 116).

In principle, this is a welcome feature, for it stabs in the right direction for accommodating the concerns about correctness in computing at the abstract-physical interface. Unfortunately, most recent work on the mechanistic account appears to have focused on an overly strict notion of 'objectivity' that stands at odds with human practice and use. The reason is that the notion of objectivity can be interpreted more or less strongly (Reiss&Sprenger 2020). In the context of computation, Duwell (2021, 18-21) recently suggested a distinction between s*trong objectivity* and w*eak objectivity*.[21] Following his suggestion, an account of computation adheres to strong objectivity if determining whether a system is computational (cf. COMP-condition) is entirely mind-independent. In contrast, accounts of computation complying with *weak objectivity* state that whether a system is computing is dependent on some specific intersubjective agreement (i.e., mind-dependent features shared by an epistemic community).

Hence, a strongly objective version of the functional mechanistic account of computation needs to provide an adequate explanation of goal-contribution in mind-independent terms. Subsequently, *living organisms* are taken as a starting point, claiming that they share a set of capacities,

---

Here, the scientists play a fundamental role in the creation (in the sense of information hiding) of a suitable computational structure.

[19] Mechanistic accounts come in two main flavors and do not necessarily need to be framed in teleological terms. Here the focus only lies on the teleological notion though.

[20] In line with Eagen (2019), I take it that 'rule-talk' is just a different, (normatively connotated) way to refer to act in accordance with a model of computation.

[21] See Fletcher (2018) for a similar discussion in the context of the Abstraction/Representation-Theory of computing.

namely survival, development, reproduction, and helping (Piccinini 2020, 68). The special functional organization of the organism allows them to pursue these capacities. And "[b]ecause these special invariants must be pursued; I call them biological *goals*." (Piccinini 2020, 69).

Notwithstanding, even when setting aside the pressing worries about whether material systems alone can ground goals, rules, purposes, etc., I take it as palpable that applying this naturalistic conception of living organisms does not translate to computational *practice*. As such, attempts to cash out physical computation in terms of natural teleology seem to be a poor fit with the widely employed notion of correctness in computer science. Not only are programs non-living things (we would commit a category mistake), there is simply no obvious way how the goal-contributions biological features of survival, development, reproduction, and helping are connected to the correctness of programs.

Having arguably anticipated some of these objections, Piccinini's goal contribution account entails the clause that teleological functions can be stable contributions to *subjective* goals too. Whilst proponents of the function-mechanistic account readily admit that such a position is rather uncontentious (Piccinini 2015, 148-9; Coelho-Mollo 2018, 446; see also Dewhurst 2018), its ramifications for use and practice are rarely fully explored. At least for the current undertaking, the upshot is this: In accordance with the insights of type-(A) implementation, we hence may also determine the correctness criteria of computational artifacts in intentional, intersubjective terms. Recently, both Schweizer (2019) and Anderson (2019) explored these options. If human agents indeed bestow artifactual functions to a material computing system according to their desires, beliefs, and intentions, computing practice can no longer be characterized by virtue of strong objectivity, since it is no longer mind-independent. But instead of having to give up on objectivity altogether and open the floodgates to complete arbitrariness, we can still rely on the notion of weak objectivity.

4.4 Taking stock

The result of the juxtaposition of type-(A) and type-(B) implementation is a common convergence among their most salient features: The commonality between all cases is the crucial involvement of (human) agents. Without the agents' decisions, one cannot plausibly account for the origins of (i) the mappings between (ii) the generated structures on different LoA (including the difficult case of the abstract-physical interface), and (iii) the occurrence of normative features determining correctness. Accordingly, a unified theory of implementation ought to entail that mappings are imputed, structures are generated, and teleological functions are ascribed.

5 A Unified Theory of Agential Implementation

Equipped with a detailed analysis of type-(A) and (B) implementation, the remainder of the paper presents a unified theory of implementation. Since the common denominator among these findings is the stipulations and conventions of (human) agents, the novel framework traces different agential involvements in terms of dependency relations. Accordingly, the result is called a *unified theory of agential implementation* (UTAI).

Fortunately, the formulation of UTAI does not need to start completely from scratch. I.e., the introduction of UTAI relies on a particular branch of the previously often dismissed interpretational or *agential* accounts of computation. I will show that a recently emerged subset of interpretational accounts relying on the well-established concepts of scientific representation and modeling facilitates the unification of (A) and (B) considerably. I first briefly illuminate the core assumptions of these specific interpretational accounts of computation (sect. §5.1). Finally, yet importantly, I present a run-through of UTAI (sect. §5.2).

5.1 Further blurred lines between material models and computers

A cluster of literature on scientific representation and modeling-based accounts of computation (type-(B)) emerged in recent scholarship. Like previous accounts of computation (SMA and EMA), these approaches distinguish between physical systems and abstract objects that are connected by an equivalence relation (cf. Fig. 2). The key difference is that the mapping between them is understood in terms of scientific representation – as opposed to some mind-independently occurring morphism. The motivations for this approach are the parallels between the practices of modeling and computing.

On the one hand, scientific representation and modeling also address the bridging problem. As I have already argued in sect. §4.1, explanations that work for modeling thus may be appropriated to computing. On the other hand, material models and concrete computing systems share that agents employ them as a surrogate to reason about something else. Overall, many different physical properties can be used as representational- or computational vehicles, respectively. For instance, when surveying the material variety of such vehicles, Sterret notes that

> "[…] electronic circuits were used as analogues of anything that could be formalized as a solution of certain classes of differential equations, and ever more sophisticated machines were developed to deal with ever larger classes of differential equations and problems. Other examples of analogues used for computation are mechanical analogues such as the geared devices built in the seventeenth century, the soap bubble analogue computers invoking minimization principles that were used to efficiently solve difficult mathematical problems in the twentieth century and biological analogue computers of the twenty-first century such as amoeba-based computing (ABC) analogue models." (Sterret 2017, 858)

Qua models, various physical systems may be employed as representational vehicles for an explanation or prediction of a target system. *Qua* computer, one may use physical systems as surrogates to read off the results of a sequence of computation specified under a model of computation M_C. The difference is that instead of a real-world target system, one then simply reasons about a particular 'hypothetical scenario', where the latter is characterizable by a transition function δ that's compatible with a model of computation M_C.

Accordingly, in some cases the distinction between what counts as a scientific model and what counts as a computer overlaps considerably. Take, for instance, Frigg & Nguyen's (2018) example

of the *Philips-Newlyn machine*. The machine uses the flow of water through a specifically designed pipe system for modeling the distribution of commodities in a national Keynesian economy. However, also known as 'MONIAC' (Monetary National Income Analogue Computer) the device can equally well be regarded as a special-purpose liquid-based analog computer. Instead of representing a selected economic scenario, the MONIAC can in principle be used to compute (a small set of) differential equations.

Several scholars turned considerations like these into an approach to physical computation. For instance, in his advancement of *L-machines* (a specific computational formalism), Ladymen (2009) concludes that concrete computation appears to be contingent on (scientific) representation.[22] Care (2010), on the other hand, provides a narrative of a use-centric history of analog computing as modeling. Similarly, but from a philosophical perspective, Papayannopoulos (2020) examined the conceptual commonalities between analog computers and analog models (when developing a notion of analog computation). Arguably the technically most worked out account to date is the Abstraction/Representation (AR) Theory introduced by Horsman, Stepney, Wagner, and Kendon (2014) and subsequently developed further by Horsman (2015, 2017), Horsman Kendon, Stepney, (2017, 2018) and Horsman et al. (2017). In a nutshell, (AR) Theory fully formalizes the relationship of concrete systems and abstract objects as a *representational triple* $\langle m_i, f, s_j \rangle$, where *f* is perceived as scientific representation, offering elaborate 'commuting diagrams' in similar fashion to Fig. 2. Consequently, computers are characterized as 'predicting devices' for computational entities.

Following these developments, Fletcher (2018) (and subsequently Duwell (2021)) critically assessed the (AR)-framework under philosophical considerations and, importantly, suggested a modified, specifically agential version.[23] Simply put, (AR) Theory should be couched in agential terms because most contemporary approaches of scientific representation are too. In sum, this accumulation of research suggests that material models and computers are (fine-tuned) physical objects employed by human agents as *epistemic tools* for their specific context-dependent purposes.

5.2 UTAI and its features

At last, let me introduce the theory that enables the unification of type-(A) and (B) implementation: UTAI. Figure 3 provides a schematic depiction of UTAI and its most important features – the implementation of a model of computation M_C at the abstract-physical interface through the execution of (one of the traces) its corresponding transition function δ. In what follows, the various elements of Fig. 3 are discussed in detail.

---

[22] Ladymen's view should not be conflated with the semantic view of computation (see e.g., Shagrir 2001, Shagrir 2022). While both accounts appeal to the notion of 'representation', they differ concerning which notion of representation is used. Put simply, the here-discussed notions advocate a view according to which physical states need to represent computational states (of an abstract model of computation). The semantic idea defends the claim that computational states require aboutness (e.g., about some affairs in the environment) to determine the computational identity (IDENT).

[23] In a similar vein, Szangolies proposed an account of (AR) Theory in terms of physical models (2020, 272-276). Paralleling Fletcher's analysis, Maroney and Timpson (2018) claim that information processing is contingent on the involvement of conscious agents or an epistemic community. Whilst the authors originally applied their reasoning to 'information processing', they state that it equally well applies to physical computation/implementation (cf. Fletcher (2018) for the same reasoning).

First, the abstract-physical interface is illustrated by the dotted line horizontally running through the diagram. In comparison to Fig. 1 (b), higher LoA and the full computational hierarchy are implied to be represented in the 'abstract realm' (upper half above the dotted line), where one deals with symbolic implementation (type-(A) implementation). In line with the SMA, the equivalence between the state transitions of M_C (specified by a transition function δ) and the evolution of physical states of a putative computing system S_C is pictured through a diagram as, e.g., found in Cummins (1989) or Ladymen (2009) (cf. Fig. 2). The labels (1) – (3) correspond to the assumptions of representation/modeling-based theories of computation discussed in the previous section (sect. §5.1):[24]

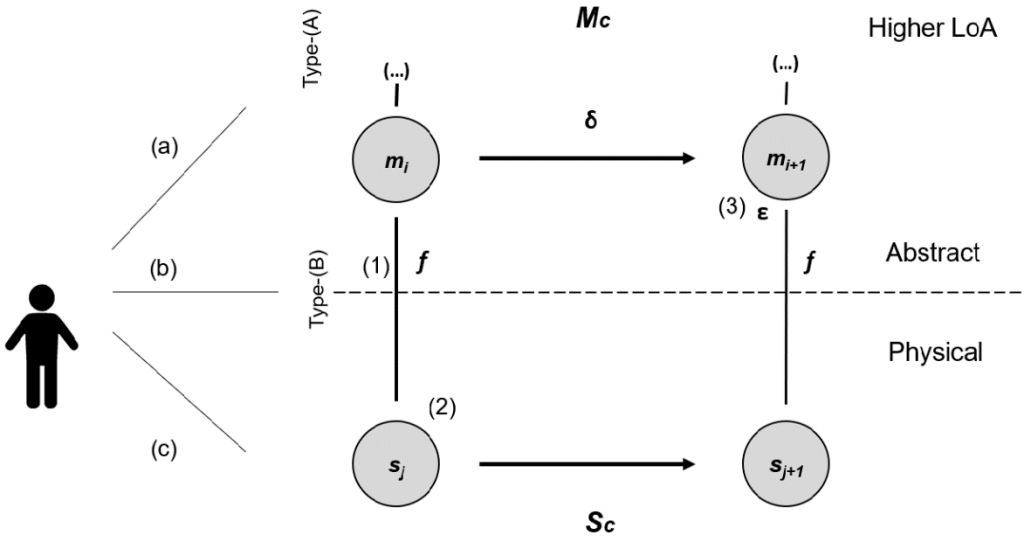

Fig. 3: Schematic depiction of a unified theory of agential implementation (UTAI)

Assumption (1) Implementation is based on a *representation relation f* between the designated *I/O-states* of the putative computing system and the abstract states transitions defined by δ of a computational formalism M_C (cf. Ladymen 2009, Horseman et al. 2014, Fletcher 2018, Papayannopoulos 2020), such that $f: M_C \rightarrow S_C$ holds. Analogous to the case of material scientific models, where certain features of the model act as a representational vehicle to represent (features of) a target system, implementation comes about when certain (selected) features of a putative computing system act as a computational vehicle to compute a function. Note that this development is a departure from the EMA, for two reasons: On the one hand, scientific representation is (nowadays) typically deemed to be directional – as such, the relation can no longer be characterized as a bidirectional morphism. Scientific models represent targets, but targets do not represent their models. On the other hand, scientific representation/ implementation can no longer be deemed a binary relation, but a ternary one, because it is only due to the use/stipulations of some agent that the linkage between model and target arises in the first place.

Assumption (2) The starting state m_i (i.e., the input state) of M_C must be *encoded* in the initial state of the material system s_j. Likewise, the output state m_{i+1} requires decoding it from s_{j+1} by reading it

[24] In as much as features (1)-(3) can already be found in the previous literature, UTAI is indebted to their respective insights.

off or performing some type of measurement. The rightwards-pointing black arrows in the diagram, labeled by δ and S_C respectively, stand for the transition of the computational states m_i and m_{i+1} and the temporal evolution of the computational vehicle/label bearers s_j (both represented by the grey circles).

Whilst maintaining that the act of encoding input-states and decoding output-states after a computation has been argued for before (e.g., Churchland & Sejnowski(1992, 63)), the here advocated view is different because the acts of encoding and decoding are formulated in terms of scientific representation. Importantly then, encoding and decoding are – in line with the notion of scientific representation – directional (no longer isomorphisms) and require to be carried out by some agent or epistemic community.

Assumption (3) Following the suggestion by Horsman and collaborators, I argue that the notion of scientific representation enables us to address *inaccuracy* as a form of miscomputation.[25] Put simply, the idea is that one may introduce an acceptable margin of error ε between the output states of M_C and output states of the physical system. If the chosen computing system would be completely reliable (i.e., run without errors), each computation cycle would yield perfect equivalence between the state transitions of M_C and the physical evolution of the system; there would be no inaccuracy. However, complete reliability is a pipe dream. Real-world systems typically do not perfectly behave in a preconceived idealized manner – as a result, the abstract series of computation and the physical outcome only coincide up to ε.[26] Depending on the chosen value of error interval ε, the very same process could count as a correct computation in one case and as a miscomputation in another.

While I think that (1)-(3) of the previous accounts are largely right, UTAI entails further features and more fine-grained factors. Specifically, UTAI needs to elucidate the involvement of and dependency on human agents concerning the imputation of mappings, the creation of structure, and the ascription of teleological functions. To uncover the intricate interrelations between human agents and the different, UTAI explicitly tracks these dependency relations,[27] denoted by (a)-(c), between human agents (represented by the black mannequin) and various elements in the implementation process. In what is left of the paper, I explain the implications of these dependency relations in detail:

Dependency relation (a) emphasizes the crucial involvement of human agents on higher LoA. It is paramount for at least two reasons: First, following the desires, intentions, and pragmatic concerns of the programmers, a 'computational problem' is formulated. In actual programming practice, this typically leads to a (formal) specification, determining what is supposed to be achieved. The specification acts as the normative yardstick to check correctness. As discussed in the type-(A) implementation literature, the specification may be seen as an ascribed teleological function. It is necessary to provide judgments about correct execution and faulty behavior (miscomputation).

---

[25] Philosophers of science commonly agree that there is at least a second type of misrepresentation, viz., mistargeting. Transposed to computation, this view amounts to the case where users (accidentally) implement the wrong computational formalism. I leave this discussion out for now.

[26] Once again, one can appropriate the discussion of scientific representation, where the idea that scientific representation reduces to isomorphism was criticized because it cannot make room for distortions.

[27] 'Dependence relation' here is understood as a relation between different entities, where one entity is dependent on another.

Furthermore, dependency relation (a) allows for illuminating a second crucial involvement of human agents. After agreeing on a specification, practitioners may then devise an algorithm. Typically, the algorithm is then formulated in a suitable computational formalism M_C (e.g., a programming language of your choice). The process described here roughly corresponds to the construction of the various LoA in the computational hierarchy (Fig. 1). The bottom line is that specifications, the algorithms targeted at the specific problem, the ensuing source code, and so on are *all* dependent on human ingenuity. Put differently, computer programs are not discovered; they are created by human agents for diverse practices such as the scientific endeavor, business, entertainment, and many more. Ignoring the agential dependence of computational artifacts bears the danger of unreasonably rendering the implementation of computational artifacts in naturalistic terms.

Dependency relation (b) concerns the mapping *f* that bridges the abstract-physical interface. Implementation may occur when agents come up with a structure-generating description (e.g., through information hiding) and a suitable mapping relating the abstract and concrete realms. As argued at length in sections §4.1 and §4.2, naturalized approaches are ill-suited to properly address the bridging problem. Instead, at least two steps necessitate the interpretational capacities of agents: structure generation and imputation.

Structure generation is contingent on agents because it demands that certain properties/capacities of the system are *selected* and *interpreted* as computational states or vehicles s_j. Structure generation may rely on a slightly tweaked version of the often-used notion of information hiding in computer science. Accordingly, agents may omit physical details that are inessential for the implementation of a series of computations (cf. Scheutz 1999, Colburn and Shute 2007, Nguyen and Frigg 2017).

Furthermore, for eventually bridging the gap between an abstract model of computation M_C and a concrete computing system S_C, a mapping relation between the two is needed. Such a relation calls for the stipulations of human agents. Users impute their chosen computational formalism onto the putative physical computing system. The material system can then be used as an epistemic tool, i.e., as a surrogate to carry out the intended series of computations determined by the previously created program.

Lastly, following these considerations, the implementation relation $f: S_C \rightarrow M_C$ bridging the abstract-physical interface (represented by the horizontal dotted line) is no longer conceivable as a mere binary relation. Instead, *f* is a ternary relation – because it necessarily depends on the stipulations of agents – characterized by a *representational quadruple* $\langle m_i, f, s_j, \text{ Agent}\rangle$.

Dependency relation (c), characterizes the physical interactions of the human agent(s) with the putative computing system. Ideally, a computing system is not only sufficiently reliable for repeated executions but also reconfigurable. Physical reconfiguration is needed for ultimately reprogramming the computing system. The mere imputation of a different model of computation M_C onto the very same unchanged structure is not sufficient for implementation. The underlying physical setup from which the structure was generated has to change too; else it results in a mismatch. So, what we require from a programmable system is that a different starting state would

have led to a different, but corresponding output state. Put differently, to be re-programmed, thus calls for a counterfactual explanation.[28]

The notions of modeling and scientific representation that underpin UTAI allow incorporating the crucial constraint that *counterfactual* claims about the computing system hold.[29] Like in the case of the EMA, the counter-factual condition rules out stipulative fiat, i.e., the completely unconstrained usage of arbitrary systems for computation that would collapse into interpretational pancomputationalism. In other words, while interpretation is a necessary condition, it is not sufficient, because the computational vehicles of the computing system need to behave suitably.

6 Conclusion and Outlook

It was argued that explaining the implementation of computational artifacts like computer programs requires combining hitherto two largely separated theories of implementation. I presented the attractive features of an agential theory of implementations: Chief here is the unification of the two required types of implementations. While type-(A) implementation concerns correctness criteria of (abstract) computational artifacts, type-(B) implementation addresses physical computation. Meeting at the abstract-physical interface their respective insights may mutually enrich each through a unifying framework.

The main result of the paper is having provided a specific way of thinking about such a unified theory of implementation. Specifically, I advanced an agential theory of implementation in terms of scientific models and representation: UTAI (unified theory of agential implementation). Put simply, recent scholarship maintaining that both material models and computers are epistemic tools used by agents to reason about different scenarios was extended. In both modeling and computing, agents engage in a form of object-based reasoning, where artifactual functions are externally attributed and a mapping relation between concrete system and abstract target/program is imputed by agents. My analysis showed that accounts, like UTAI, sketched in agential terms offer the right kind of resources to accommodate the main underlying assumptions of both type-(A) and (B) implementation: stipulated mappings; generated structures; and ascribed teleological functions. In sum, UTAI has the explanatory virtue of facilitating cross-talk between so far rather separated discourses and kinds of literature.

As a final remark, I believe that the scope of agential theories of implementation based on scientific representation like UTAI deserves further research in multiple directions. For instance, it remains to be established how much this account extends beyond human-made artifacts (i.e., biological systems like the brain). Particularly, there remains the open question of how the traditionally naturalistic accounts employed in the context of the computational theory of mind respond to the analysis that they cannot adequately overcome the problems associated with the bridging problem.

[28] N.b., this insight is virtually similar to the ones of the counterfactual/causal/dispositional EMAs).

[29] In the context of scientific modeling, Bokulich (2011, 39) for instance reminds us that
"[…] in order for a model M to explain a given phenomenon P, we require that the counterfactual structure of M be isomorphic in the relevant respects to the counterfactual structure P. That is, the elements of the model can, in a very loose sense, be said to "reproduce" the relevant features of explanandum phenomenon."

Acknowledgements

I would like to thank Liesbeth De Mol, Raymond Turner, Ruward Mulder, and Alice Martin for helpful discussions and/or comments on (earlier versions of) this manuscript. This research was funded by the *PROGRAMme* project (ANR-17-CE38-0003-01).

References

(Adamatzky2021) Adamatzky, Andrew (ed.). 2021. *Handbook of Unconventional Computing*. World Scientific.

(Anderson2019) Anderson, Neal G. 2019. 'Information Processing Artifacts.' *Minds and Machines* 29 (2): 193–225.

(Batterman2010) Batterman, Robert W. 2010. 'On the explanatory role of mathematics in empirical science.' *The British Journal for the Philosophy of Science* 61 (1): 1–25.

(Bokulich2011) Bokulich, Alisa. 2011. 'How Scientific Models Can Explain.' *Synthese* 180 (1): 33–45.

(Bueno and Colyvan2011) Bueno, Otávio, and Mark Colyvan. 2011. 'An inferential conception of the application of mathematics'. *Noûs* 45 (2): 345–374.

(Butterfield, Ngondi, and Kerr2016) Butterfield, Andrew, Gerard Ekembe Ngondi, and Anne Kerr. 2016. *A Dictionary of Computer Science*. Oxford University Press.

(Care 2010) Care, Charles. 2010. *Technology for Modelling. Electrical Analogies, Engineering Practice, and the Development of Analogue Computing*. Springer.

(Chalmers et al.2012) Chalmers, David, et al. 2012. The varieties of computation: A reply. *Journal of Cognitive Science* 13 (3): 211–248.

(Churchland and Sejnowski1992) Churchland, Patricia S, and Terrence J Sejnowski. 1992. *The computational brain*. MIT Press.

(Coelho Mollo2018) Coelho Mollo, Dimitri. 2018. 'Functional individuation, mechanistic implementation: The proper way of seeing the mechanistic view of concrete computation'. *Synthese* 195 (8): 3477–3497.

(Colburn 1999) Colburn, Timothy R. 1999 'Software, Abstraction and Ontology'. *The Monist* 82 (1): 3–19.

(Colburn et al.1993) Colburn, Timothy R., Terry L. Rankin, and James H. Fetzer (eds.). 1993. Program Verification: Fundamental Issues in Computer Science. Springer.

(Contessa2010) Contessa, Gabriele. 2010. 'Empiricist structuralism, metaphysical realism, and the bridging problem'. *Analysis* 70, no. 3.

(Copeland1996) Copeland, B Jack. 1996. 'What is computation?' *Synthese* 108 (3): 335–359.

(Cummins1989) Cummins, Robert. 1989. *Meaning and Mental Representation.* MIT Press.

(Curtis-Trudel2022) Curtis-Trudel, Andre E. 2022. 'Why do we need a theory of implementation?' *The British Journal for the Philosophy of Science* 74 (4).

(De Mol2021) De Mol, Liesbeth. 2021. 'Turing Machines.' In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Winter 2021. Metaphysics Research Lab, Stanford University.

(Dewhurst 2018) Dewhurst, Joe. 2018. 'Computing Mechanisms Without Proper Functions.' *Minds and Machines* 28: 569–588.

(Duwell2021) Duwell, Armond. 2021. *Physics and Computation*. Cambridge University Press.

(Egan2019) Egan, Frances. 2019. 'Defending the Mapping Account of Physical Computation'. *APA Newsletter* 19 (1): 24–25. 2

(Fletcher2018) Fletcher, Samuel C. 2018. 'Computers in Abstraction/Representation Theory'. *Minds and Machines* 28 (3): 445–463.

(Floridi et al.2015) Floridi, Luciano, Nir Fresco, and Giuseppe Primiero. 2015. 'On malfunctioning software.' *Synthese* 192 (4): 1199–1220.

(Fresco2014) Fresco, Nir. 2014. *Physical computation and cognitive science*. Springer.

(Fresco, Copeland, and Wolf2021) Fresco, Nir, B. Jack Copeland, and Marty J. Wolf. 2021. 'The Indeterminacy of Computation'. *Synthese* 199 (5-6): 12753–12775.

(Fresco and Primiero2013) Fresco, Nir, and Giuseppe Primiero. 2013. 'Miscomputation'. *Philosophy & Technology* 26 (3): 253–272.

(Frigg2022) Frigg, Roman. 2022**.** *Models and Theories: A Philosophical Inquiry*. Routledge

(Frigg and Nguyen2018) Frigg, Roman, and James Nguyen. 2018. 'The turn of the valve: representing with material models'. *European Journal for Philosophy of Science* 8 (2): 205–224.

(Giere1999) Giere, Ronald N. 1999. 'Using models to represent reality'. In *Model-based reasoning in scientific discovery*, 41–57. Springer.

(Godfrey-Smith2009) Godfrey-Smith, Peter. 2009. 'Triviality arguments against functionalism'. *Philosophical Studies* 145 (2): 273–295.

(Horsman et al.2014) Horsman, Clare, Susan Stepney, Rob C Wagner, and Viv Kendon. 2014. 'When does a physical system compute?' *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 470 (2169): 20140182.

(Horsman2015) Horsman, Dominic C. 2015. 'Abstraction/representation theory for heterotic physical computing.' *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 373 (2046).

(Horsman2017) Horsman, Dominic. 2017. 'The representation of computation in physical systems.' in *EPSA15 selected papers*. M. Massimi, J. Romeijn, and G. Schurz (eds.), 191–204. Springer.

(Horsman, Kendon, and Stepney2017) Horsman, Dominic, Vivien Kendon, and Susan Stepney. 2017. 'The Natural Science of Computing.' *Commun. ACM* 60 (8): 31-34 (July).

(Horsman, Kendon, and Stepney2018) Horsman, Dominic, Viv Kendon, and Susan Stepney. 2018. 'Abstraction/Representation Theory and the Natural Science of Computation', in *Physical*

Perspectives on Computation, Computational Perspectives on Physics. Michael E. Cuffaro and Samuel C. Fletcher (eds.), 127–150. Cambridge University Press.

(Horsman et al.2017) Horsman, Dominic, Viv Kendon, Susan Stepney, and J Peter W Young. 2017. 'Abstraction and representation in living organisms: when does a biological system compute?' in *Representation and reality in humans, other living organisms and intelligent machines*. Gordana Dodig-Crnkovic and Raffaela Goivagnoli (eds.), 91–116. Springer.

(Houkes and Vermaas2010) Houkes, Wybo, and Pieter E Vermaas. 2010. *Technical functions: On the use and design of artefacts*. Volume 1. Springer.

(Klein2008) Klein, Colin. 2008. 'Dispositional implementation solves the superfluous structure problem.' *Synthese* 165 (1): 141–153.

(Kroes2012) Kroes, Peter. 2012. *Technical artefacts: Creations of mind and matter: A philosophy of engineering design*. Volume 6. Springer.

(Ladymen2009) Ladymen, James. 'What does it mean to say that a physical system implements a computation?' *Theoretical Computer Science* 410, 376–383.

(Lee2020) Lee, Jonny. 2020. 'Mechanisms, wide functions, and content: Towards a computational pluralism'. *The British Journal for the Philosophy of Science* 72 (1): 221-244.

(MacKenzie2004) MacKenzie, Donald A. 2004. *Mechanizing proof: computing, risk, and trust*. MIT Press.

(Maroney and Timpson2018) Maroney, Owen JE, and Christopher G Timpson. 2018. 'How is There a Physics of Information? On Characterizing Physical Evolution as Information Processing.' In *Physical Perspectives on Computation, Computational Perspectives on Physics*, 103-126. Cambridge University Press

(Miłkowski2013) Miłkowski, Marcin. 2013. *Explaining the computational Mind*. MIT Press.

(Newman1928) Newman, M. H. A. 1928. 'Mr. Russell's Causal Theory of Perception'. *Mind* 37 (146): 26–43.

(Nguyen and Frigg2017) Nguyen, James, and Roman Frigg. 2017. 'Mathematics is not the only language in the book of nature'. *Synthese*, pp. 1–22.

(Papayannopoulos2020) Papayannopoulos, Philippos. 2020. 'Computing and modelling: Analog vs. Analogue'. *Studies in History and Philosophy of Science Part A* 83:103–120.

(Patterson and Hennessy2014) Patterson, David A., and John L. Hennessy. 2014. *Computer Organization and Design - The Hardware /Software Interface* (Revised 5th Edition). The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press.

(Piccinini2007) Piccinini, Gualtiero. 2007. 'Computing Mechanisms.' *Philosophy of Science* 74 (4): 501–526.

(Piccinini2015) Piccinini, Gualtiero. 2015. *Physical computation: A mechanistic account*. Oxford University Press.

(Piccinini2020) Piccinini, Gualtiero. 2020. *Neurocognitive Mechanisms: Explaining Biological Cognition*. Oxford University Press.

(Pincock2004) Pincock, Christopher. 2004. 'A new perspective on the problem of applying mathematics'. *Philosophia Mathematica* 12 (2): 135– 161.

(Preston2018) Preston, Beth. 2018. 'Artifact'. In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Fall 2018. Metaphysics Research Lab, Stanford University.

(Primiero2019) Primiero, Guiseppe. 2019. *On the Foundations of Computing*. Oxford University Press.

(Psillos2006) Psillos, Stathis. 2006. 'The Structure, the Whole Structure, and Nothing but the Structure?' *Philosophy of Science* 73 (5): 560–570.

(Putnam1988) Putnam, Hilary. 1988. *Representation and Reality*. MIT Press.

(Rapaport1999) Rapaport, William J. 1999. 'Implementation is Semantic Interpretation'. *The Monist* 82 (1): 109–130.

(Rapaport2005) Rapaport, William J. 2005. 'Implementation is semantic interpretation: further thoughts'. *Journal of Experimental & Theoretical Artificial Intelligence* 17 (4): 385–417.

(Reiss and Sprenger2020) Reiss, Julian, and Jan Sprenger. 2020. 'Scientific Objectivity'. In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Winter 2020. Metaphysics Research Lab, Stanford University.

(Rescorla2014) Rescorla, Michael. 2014. 'A theory of computational implementation'. *Synthese* 191 (6): 1277–1307.

(Ritchie and Piccinini 2018) Ritchie, J. Brendan, and Gualtiero Piccinini. 2018. 'Computational implementation'. In *The Routledge Handbook of the Computational Mind.* 192-204. Routledge.

(Scheutz1999) Scheutz, Matthias. 1999. 'When physical systems realize functions...' *Minds and Machines* 9 (2): 161–196.

(Schweizer2019) Schweizer, Paul. 2019. 'Computation in physical systems: A normative mapping account'. In *On the cognitive, ethical, and scientific dimensions of artificial intelligence*, 27–47. Springer.

(Scott2009) Scott, Michael L. 2009. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc.

(Shagrir2001) Shagrir, Oron. 2001. 'Content, computation and externalism'. *Mind* 110 (438): 369–400.

(Shagrir2022) Shagrir, Oron. 2022. *The Nature of Physical Computation*. Oxford University Press.

(Sprevak2010) Sprevak, Mark. 2010. 'Computation, individuation, and the received view on representation'. *Studies in History and Philosophy of Science Part A* 41 (3): 260–270.

(Sprevak2018) Sprevak, Mark. 2018. 'Triviality arguments about computational implementation.' In *The Routledge Handbook of the Computational Mind*, 175–191. Routledge.

(Steiner1998) Steiner, Mark. 1998. *The Applicability of Mathematics as a Philosophical Problem*. Harvard University Press.

(Sterrett2017) Sterrett, Susan G. 2017. 'Experimentation on analogue models'. In *Springer handbook of model-based science*, 857–878. Springer.

(Suárez2003) Suárez, Mauricio. 2003. 'Scientific Representation: Against Similarity and Isomorphism.' *International Studies in the Philosophy of Science* 17 (3): 225–244.

(Swoyer1991) Swoyer, Chris. 1991. 'Structural Representation and Surrogative Reasoning'. *Synthese* 87 (3): 449–508.

(Szangolies2020) Szangolies, Jochen. 2020. 'The Abstraction/Representation Account of Computation and Subjective Experience.' *Minds and Machines* 30 (2): 259–299.

(Turner2011) Turner, Raymond. 2011. 'Specification'. *Minds and Machines* 21 (2): 135–152 (May).

(Turner2012) Turner, Raymond. 2012. 'Machines'. In *A Computable Universe.* Hector Zenil (ed.), 63–76. World Scientific.

(Turner2014) Turner, Raymond. 2014. 'Programming Languages as Technical Artifacts'. *Philosophy & Technology* 27 (3): 377–397.

(Turner2018) Turner, Raymond. 2018. *Computational Artifacts*. Springer.

(Turner2021) Turner, Raymond. 2021. 'Computational Abstraction'. *Entropy* 23(2).

(van Fraassen2008) van Fraassen, Bas C. 2008. *Scientific Representation: Paradoxes of Perspective*. Oxford University Press.

(Turner and Angius2020) Turner, Raymond, and Nicola Angius. 2020. 'The Philosophy of Computer Science'. In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Winter 2017. Metaphysics Research Lab, Stanford University.

(Weisberg2013) Weisberg, Michael. 2013. *Simulation and similarity: Using models to understand the world*. Oxford University Press.

(Wigner1960) Wigner, Eugene. 1960. 'The Unreasonable Effectiveness of Mathematics in the Natural Sciences.' *Communications in Pure and Applied Mathematics* 13:1–14.