Understanding Computation

A General Theory of Computational Processes

Jan van Leeuwen

Jiří Wiedermann

Technical Report UU-CS-2019-012
December 2019

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
Princetonplein 5
3584 CC Utrecht
The Netherlands

Understanding Computation[⋆]

A General Theory of Computational Processes

Jan van Leeuwen[1] Jiří Wiedermann[2]

[1] Dept. of Information and Computing Sciences, Utrecht University,
Princetonplein 5, 3584 CC Utrecht, the Netherlands
J.vanLeeuwen1@uu.nl

[2] Institute of Computer Science, Academy of Sciences of the Czech Republic,
Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic
jiri.wiedermann@cs.cas.cz

"All machinery can be regarded as continuous"
A.M. Turing [42], 1948

Abstract. The notion of computation is well understood, and well formalized, in the classical context of digital information processing. However, the paradigm of computation is increasingly used in the characterization of processes in sciences like physics and biology as well. This seems to recognize computation more broadly as an 'elementary mechanism of nature', not limited to the digital domain. What *is* computation, if it is observed in this broader sense? Without computers as the single defining mechanism, classical abstractions like Turing machines and rewrite systems seem to be unsuited to capture its newer meanings. Following our epistemic approach, we develop a new framework for understanding computation and, as a consequence, for understanding computational processes, based on observed trajectories (curves) of computational activity in suitable metric spaces. With computational processes rather than all sorts of different models of computation as the core abstraction, one can model and characterize many aspects of computation as a mathematical object, from composition to computational and structural complexity. Discrete computations appear as projections of continuous ones, clarifying a complex issue from Turing's 1948 report. We present both the philosophy and a first outline of the implied, machine-independent, general mathematical theory of computation.

Keywords: computation, computational processes, dynamical systems, epistemic theory, metric spaces, philosophy of computing, topology.

Contents:

[⋆] Version dated: December 3, 2019. This work was partially supported by ICS CAS fund RVO 67985807, CAS Programme Strategy 21, and the Karel Čapek Center for Values in Science and Technology (Prague, Czech Republic).

1 Introduction

Computation is traditionally understood to refer to the calculatory actions of a human computer and, by extension, to anything that can essentially be reduced to this. This view was well-formalized by Turing and in many later studies, and it is well-recognized since. Computation is now routinely associated with the action of electronic computers. However, the language of computation is increasingly being used to describe the functioning of processes in physics and biology as well, and these processes may neither be discrete nor obviously algorithmic by our common understanding. As computers are no longer the single defining mechanism, classical abstractions like Turing machines or rewrite systems seem less and less appropriate to define computation in the general way it is now understood. The philosophical question arises whether a more encompassing theory could be given. What is it that computations do?

1.1 Re-investigating computation

The question to re-investigate the core notions of computation has emerged in all kinds of contexts before, especially in established domains of computing. A good example is modern 'scientific computation' for which a real of algebraic model of computation seems far more suitable (cf. Blum et al. [7]). Also, from a programming perspective, the logic-based model of *abstract state machines* (cf. Gurevich [20]) seems to give a much more viable view of how a computation proceeds and, thus, can be specified. Additional phenomena that arise in practice like interaction (reactivity), unboundedness of operation and program evolution have all been added to the regular Turing machine model, to obtain models that are deemed more natural for studying modern computations [47].

Matching models As a general guideline, Aho [1] suggested that computation must be defined in conjunction with a precise model of computation which matches the problem one wants to study. For example, the computational phenomena that one wants to study may be captured by some hybrid model, whose meaning and operations can be defined beyond doubt. Depending on the nature of the model, this might focus attention on the operation and complexities of the chosen model rather than on the nature of the computation it achieves. Examples of how this can be done differently can be seen in e.g. Blum's machine-independent theory of discrete computation [6] or in Valiant's general approach to parallel computation [43].

A good impression of the many different approaches to the notion of computation as found today, is given in the contributions to the *Ubiquity* Symposium in 2010, see Denning [15]. These witness the gradual transition in thinking about the notion, from 'computation by machines' to 'computation by processes', allowing for a much greater flexibility in matching it to our newer understandings of it. In our present analysis, this still lacks uniformity and focus on the core notion of computation.

1.2 What computations do

The repeated adjustments of standard models of computation to the developments in computing architectures, have renewed the quest for understanding the core concept of computing that underlies it. The question is: is there an overarching model of computation that emphasizes *computation*, without having to resort to a fixed model of the underlying software or hardware?

In 2013 we developed a very different philosophy of computation, focusing on 'what computations do' (for us) rather than on 'how their underlying mechanism achieves it' [48, 50]. We elaborate on this approach below. Before we go into detail, the question arises how one can abstract from an underlying mechanism here altogether.

Revisiting Turing's 1948 report It has long been recognized that there can be great differences in the functioning of computing machines, presumably with non-trivial consequences for understanding them. In 1948, Turing already described different - what he called - 'varieties of machinery' from this perspective

[42]. In particular, Turing described the apparent dichotomy between *discrete* and *continuous* machinery which is so often seen, as follows ([42], p. 3):

> *We may call a machine ‘discrete when it is natural to describe its possible states as a discrete set, the motion of the machine occurring by jumping from one state to another. The states of ‘continuous machinery on the other hand form a continuous manifold, and the behaviour of the machine is described by a curve on this manifold.* All machinery can be regarded as continuous, *but when it is possible to regard it as discrete it is usually best to do so.*

Interestingly, the description of the dichotomy holds the key to its resolution, if one accepts the view that ‘all machinery can be regarded as continuous’. With the notable exception of [19], this idea seems not to have been followed up so far in the theory of computation. As discrete and continuous machinery are omnipresent in all of the systems we study today, we have to resolve the noted dichotomy in our overarching model of computation if it is to prove its worth.

In view of his 1948 report, one wonders whether Turing would have considered his model of computation from 1936 [41] as a proper basis for understanding the notion of ‘machinery’ which he is now referring to. It is a tantalizing thought that he might not have.

Computational processes In order to understand computation, we have to assume that a spectator can observe the behaviour of a substrate that underlies it. Without requiring any detail, we will assume that the behaviour we want to understand as being computation, is being generated by a computational process (or agent). Effectively this means that *we need to understand what computational processes do as much as what computations do, with as little detail as possible of ‘how’ processes do what they do to cloud the issue.*

The connection of computations to processes rather than machines is well-known, but this has not led beyond the traditional theories of computing before. The framework we develop can be seen as answering a fundamental question of Frailey [17] which calls for a theory of computational processes as a prerequisite for the true understanding of computation. We fully subscribe to Frailey’s motivation for this when he writes:

> *Why should we equate process and computation? If we did so we would overcome the "mathematicians bias" and deal with the complete scope of what computation has come to include. [...] We could develop a theory of processing analogous to our current theory of computation. Such a theory would be a better fit to the reality of what computation truly entails. Whether natural or artificial; mechanical, electrical or biological; described by a formal model or simply occurring naturally and in need of a formal description; processes should be the focus of our attention when we try to understand computation.*

If we adopt this view, the question of understanding ‘what computations do’ reduces to the question of developing an appropriate theory of computational processes. In doing so, one requirement must then be upheld, namely that such a theory does not resort to any special model of the underlying software or hardware.

1.3 Overview

The overarching model we aim for will have to abstract from many features that are manifest in current computing systems. Moreover, we want the framework to be suited for understanding both natural and artefactual systems that we want to include as being computational.

To achieve our aim, we will develop a more general view of computing machinery than Turing's, to bring out the intrinsic meaning of computation in any context. We base our approach on the *epistemic theory to computation* which we developed prior to the present work [48, 50]. We use it to devise a novel theory of computational processes that leads us to understand computation irrespective of how the underlying process achieves it. The theory may be seen as a concretization of the uniform framework for computational machinery as it was alluded to in Turing's 1948 report [42].

With computational processes rather than all sorts of models of computation as the core abstraction, one can model and characterize many aspects of computation, from composition to computational and structural complexity. A very preliminary version of this report was presented in [45]. Here we present both the underlying philosophy and a first outline of the implied, generalized mathematical theory of computation in detail.

Organization of this report In Section 2 we unfold the philosophical considerations that underpin our approach and that eventually lead us to the general view of computation that we have alluded to above. As an immediate follow-up on these considerations, Section 3 provides the key abstractions for defining and studying computations and computational processes in a mathematically precise manner. In Section 4 we give a series of examples that aim to illustrate the extensiveness of our approach. In Section 5 we show that the framework allows us to treat computational processes as mathematical objects, and that it enables us to distinguish and define many realistic features of processes in a natural way, including issues such as re-timing and composition. In Section 6 we show that every computational process has a symbolic 'prototype' that precisely characterizes what the given computations of the process can maximally accomplish when different observers, with different interpretations of what they see, observe them.

In the later sections of this report, we will develop a 'theory of computation' for the general notion of computation that we propose. In Section 7 we consider what it means to 'observe' computations and when computational processes may be regarded as operationally or observationally discrete processes. In Section 8 we close in on Turing's claim and consider the relationship between, what we will call, 'discrete processing systems' and 'continuous computational processes'. In Section 9 we will prove that, under mild assumptions, any 'discrete processing system' as commonly considered in computing can be viewed as the rendering of a general computational process as we define it, with special properties. This clarifies the apparent dichotomy in the view of 'machinery' in Turing's 1948 report [42]).

In Section 10 we define the notion of functional processes, and develop a concept similar to differentiation for computational processes, telling us when a process admits a 'step function' that works as a program for it. In Section 11 we then show how diverse notions of complexity can be neatly expressed in the framework, with a discussion of what properties of computations may be 'knowable'. In Section 12 we give some conclusions and reflect on some open questions our framework leads to.

2 Philosophical considerations

In the remainder of this report we will develop an answer to the question what it is that computations do, with a level of generality that will allow us to focus on the concept of computation proper. We first describe the philosophical background of our approach. We then summarize the main results that will be obtained in this report.

Overall we believe that the philosophy of computing is indispensable for finding new pathways of research into computation. Two ingredients will be crucial: the common characteristic of computational machinery as it was suggested in Turing's 1948 report [42], and the recent *epistemic theory of computation* which we developed earlier [48, 50].

2.1 Cornerstones

If we do not limit computation to the digital domain per se, then how should it be viewed? In [22, 23] it was shown how abstractly representing key processes in a known computational domain can clarify when natural systems may be said to 'compute'. For understanding the core notion of computation, this is not sufficient. We sketch the two philosophical cornerstones we use.

Actors and spectators In most approaches to understanding things, whether they are real or artificial, a Kantian duality arises between understanding a thing as it is (Kant: noumenon) and the thing as it appears to us (Kant: phenomenon). In understanding computation one can recognize a similar duality. We will use the *actor-spectator paradigm* due to Beck [4], to describe our field of concern as follows. On the one hand there is an *actor* which produces something (in our case, presumably through an act of computation), and on the other there is a *spectator* which observes, interprets, and signifies what the actor does. Clearly, it is the interplay between these two that ultimately determines the meaning of what the actor does.

Some further reflections are in order. In stead of one actor one could have many, but we only consider the acting of one (or, in any case, of only a bounded number of them). We do not make any assumptions beforehand how an actor does what it does, whether it follows a program, whether it interacts with its surroundings, and what physical properties it has other than that it takes 'some' time to produce whatever it produces. We will later use the term 'process' for what is here called an actor, and sometimes we use the term 'agent' if we mean a process in the act of producing something.

Also, in stead of one spectator one could have several, but we confine ourselves to having just one. We do not prescribe what spectators should see, but merely assume that they can all 'track' what an actor produces in some quantitative way (e.g. by means of a bounded set of parameters). Also, spectators need not only be signifiers of whatever an actor produces, but they may also influence the (functioning of the) actor, and vice versa. For example, a spectator may limit an actor to be active for only a finite time, or become an actor itself based on whatever an actor around him produces. Here we will view spectators primarily as (passive) 'observers' and often use this term to denote them.

Epistemic theory Actors can be artificial objects like (programmed) machines or robots, or natural ones like cells or physical systems 'in context'. Indeed, actors can be any system embedded in and potentially interacting with an environment, in which it is capable of producing or generating something. All we need now is a philosophy that tells us when an actor (or process) is producing 'computations'.

Although we may or may not know anything about how an actor produces what it does, we assume that the spectator possesses the necessary means to observe the actor and has a reasoning capability to deal with its observations and possibly conclude from them, in a formal or informal way. Taking this a step further, the *epistemic theory of computation* holds that an observed 'action' is a computation precisely when the action is productive, seen from the perspective of the (knowledge-)theory of the spectator, i.e. when its result is both 'explainable' and 'provable' as an outcome from the premises used by the actor [48–50].

Hence we contend that a spectator, when he is observing a given actor, is observing something he may call a *computation* when the following conditions are met:

– the observed entities over time make sense in a (more or less) well-defined and formal *knowledge domain* he has at his disposal, and
– the overall observed behaviour is that of producing (new) knowledge items in the domain, consistent with the rules of reasoning in the domain.

This complies with the dogma of the epistemic theory of computation, which asserts that *computations are acts of knowledge generation.* The only fact we need to rely on is that, whatever the spectator observes and/or concludes in its frame of reasoning, must be supported by the actions of the actor process. Note that, in this approach, computation is an observer-dependent notion [49].

The epistemic view of computation deviates substantially from the common views of computations as the actions of a specified (or: programmed) computer model, or as transformations of information in some abstract setting. In [48, 50] we have given many examples of the versatility of the epistemic approach, including many examples which are not well-explained or covered by other approaches. We adopt it as our reference view of computation here.

The, observer-dependent, approach is a very flexible one and makes the epistemic philosophy of computation very versatile. For example, it has been used to give computational views of a variety of cognitive processes, and further applications are easily imagined [51].

Reflection In its most general sense, the epistemic theory recognizes that a 'process' is only truly understood once the observable effects it generates over time can be seen as a temporal chain of knowledge that is generated according to the, known or unknown, rules of some underlying knowledge domain. If a process is understood in this way, it is declared 'computational'.

The epistemic theory explains why the range of processes viewed as computational is rapidly expanding, from classical numerical computation in mathematics to all types of processes in other sciences like biology, physics and chemistry nowadays. It is also consistent with the widely-held view that computation is among the fundamental processes in Nature and in the Universe, giving meaning to the proper way in which computation has to be understood as a phenomenon [16].

2.2 What computations are

The given philosophy is still very general but outlines how we aim to approach the specific challenge of this report. Here we outline how the philosophy is applied to obtain a general definition of computational processes and of the computations they produce. We will show later on that many other notions that one may want to define for computational systems, can be cast in this very framework as well.

Computational processes In our general approach to computation, computational processes are merely seen as actors that produce computations. No further assumptions will be made about them, i.e. other than that they must be capable of generating actions over time that are recognized as computations by the criteria of a spectator. In [49] this was restated in formal terms, but here we concretize it further.

In order to develop a theoretical framework, we need to identify *what* it is that the actor produces and the spectator observes over time as being computation. For this, we use the key insight from Turing's 1948 report [42] that all machinery, i.e. all computational processes, can be regarded as continuous, in the following sense:

> *The states of 'continuous machinery [. . .] form a continuous manifold, and the behaviour of the machine is described by a curve on this manifold.*

If we adopt this, the far-reaching consequence is that computational processes are simply actors capable of producing activities over time that can be seen as curves on a suitable manifold, with the additional requirement that these curves capture the generation of knowledge according to the spectator. The actor's production of these activities is then called *computation*.

We are left with the problem to define computational processes as a notion. As it is impossible to specify uniformly *how* processes do what they do, we confine ourselves to defining *what* they do. We formulate it in very general terms, to be made more precise later.

> *Thesis.* A computational process consists of the set (family) of all curves determined by some actor, under the proviso that the actor only produces allowable curves, i.e. curves that satisfy the criteria for computations as imposed by the spectator.

This extensional characterization gives us the generality we need for appreciating computational processes and computations as generic phenomena. It is both machine- and algorithm-free, i.e. all internal details and I/O specifics of the particular process we deal with are perfectly hidden. Indeed, these operational details are hidden in the details of the manifold and the curves, and in the criteria that make curves *allowable*, i.e. generating knowledge in the knowledge domain at hand. If more details of the process are called for, it is up to the spectator to inspect the actor in question more closely.

General theory of computation Given the general characterization of computational processes and computations, we can finally aim to develop a theory of computation that applies to both digital and non-digital domains and that is not bound to traditional computational models only. It enables us to develop a theory that is potentially applicable to computation in any context where it is meaningful. The theory is effectively a general theory of computational processes.

In order to make underlying notions like manifolds and curves precise, we have to identify suitable *action spaces* in which these concepts make sense. We will require that the action space of a computational process is a topological space of some kind, so we can define what it means for actions that are close 'in time' to be close 'in the action space' as well. In fact, for the purposes of this report, we will always require that action spaces are *metric*. Computational processes thus manifest themselves by generating curves that represent 'continuous chains of observed actions' in their action space.

This characterization of computational processes, however, also requires that the generated curves are 'allowable', i.e. that they make sense when viewed as chains of actions in a suitable *knowledge space*. Additional criteria have to imposed 'along a curve' in order that it can be interpreted as generating knowledge in the relevant knowledge space. The combination of topological and knowledge-theoretical (logical) properties gives a powerful framework for mathematical analysis.

In this report we will prove that a versatile theory can be developed along these lines. We will show that many features of computations and computational processes can be cast in this framework, and that theorems can be proved that correspond to natural properties. In particular, the framework that we develop will enable us to resolve Turing's dichotomy and prove the following result, under mild assumptions: *every discrete computational process is the 'projection' of a general, i.e. continuous, computational process.*

2.3 Cross connections

There are notable connections between our framework and the frameworks employed in various other fields. We discuss the connection to dynamical systems theory and to trace theory.

Computational vs. dynamical systems Rephrasing the view we developed, computational systems may well be described as 'families of allowable curves determined by computational processes'. This reminds of the definition of dynamical systems, which are commonly described as 'families of motions determined by evolutionary processes' [28].

Even without precise definitions, one can immediately infer that dynamical systems are computational from the perspective of our framework *only* if the motions determined by them can be viewed as 'allowable', i.e. if there is some knowledge domain in which all motions can be viewed as generating knowledge (by the criteria of a spectator). If this condition is not satisfied for a given dynamical system, then it is not computational. However, if the condition is satisfied, then it *is* computational, as we do not concern ourselves with the way a system actually generates its curves.

Conversely, computational systems may be viewed as dynamical *only* if their curves can be seen as 'motions', i.e. are determined by some system of difference or differential equations as in common dynamical systems. This will often not be the case, nor be known explicitly if it is. Also, for our purposes we do not want to rely on any internal specification of the system, as opposed to the types of analyses of dynamical systems [28].

Nevertheless, the fact that the frameworks of computational and dynamical systems have some overlaps, suggests that their theories might have overlaps as well. The connection between (the control theory of) dynamical systems and the modeling of computation (by machines) was noted before, e.g. by Arbib [2] some fifty years ago. He wrote ([2], p. 161):

> *[...] the two sciences of control theory and automata theory can gain immensely if structures and techniques of each field are examined and reworked in the light of the structures and techniques of the other.*

As a case-in-point, Arbib [3] showed how a 'tolerance relation' could take the place of continuity, to model state machines that act in discrete time as dynamical systems. In the resulting framework of *tolerance automata*, he was able to characterize an 'optimal control problem' for automata. An excellent view of the computational interpretation of dynamical systems concepts and theories was given by Stepney [38]. Siegelmann and Fishman [37] exploited the analogy to develop a theory of computational complexity for dynamical systems. See also Platzer's survey of analog and hybrid computation, explained from a discrete and continuous dynamical systems perspective [35] and the theory of analog computation through dynamical systems due to Bournez et al. [8, 9].

Nevertheless, as computations are not motions and dynamical systems are not aimed at generating knowledge, the analogies between the two notions are only of limited use to the fundamental analysis we present here. Our primary aim is to clarify the essence of computation.

Trace theory The notion of computation we follow in this report, is inspired by Turing's view that the progressing behaviour of a machine over time can be seen as a curve on a suitable manifold. This suggests a possible connection to the theory of so-called *traces*, which are symbolic representations of the sequences of actions of a machine or system (of processes) over time.

The interpretation of traces as qualified sequences of symbols over some alphabet is well studied. Traces were originally defined for understanding the behaviour of *concurrent systems*. Mazurkiewicz [26] already showed, in the nineteen seventies, that traces can represent the possible serializations of the actions of interacting processes on a finite or infinite time scale. Thus, concurrent systems may be seen as 'families of traces determined by communicating processes'.

This description of concurrent systems reminds of that of computational systems as 'families of allowable curves determined by computational processes'. However, the connection does not go further than this. At the very general level at which we are studying computational processes here, no actions are distinguished and thus there is no symbolic aspect to the curves as we shall use them. Also, generated knowledge can generally not be seen as traces. Nevertheless, some connections will appear as we proceed, e.g. when we study compositional aspects of computational processes.

2.4 Towards a topological framework

In the remainder of this report we will expand the general framework. Actors become computational processes, and what they can produce (or generate) will characterize our understanding of computation in a very broad and generic sense. Spectators turn into observers that interpret whatever the observed computations tell them and thus what these do. Eventually, computations must lead to items of reasoning, i.e. to knowledge of some kind, for their observer.

From our considerations the view emerges that computations result from the possible lines of action of computational processes which are funneling many causal effects in time. The lines of action need not be described by a program - they merely give an observable view of a 'run' of the underlying process over time, within a certain environment that evolves over time as well. In their computations, processes are continually building up knowledge in accordance with the, known or unknown, rules of an underlying knowledge domain.

In order to obtain a theory from these principles, we will first elaborate on the view of computations as trajectories, or curves, by giving a combined topological and knowledge-theoretic framework in which computations are formulated as purely mathematical objects with characteristic properties. This will lead us to a general theory of computational processes that is algorithm-free, representation-free and machine-independent and that embodies the philosophy of computation as knowledge generation. In the end we want to be able to 'program' computational processes for any purpose.

In elaborating this theory of computation, we will use the concepts of *general topology* to elicit many of its key aspects. For basis concepts and definitions we refer to any standard textbook on General Topology (e.g. [30], or see [46]). A first outline of the topological instantiation of the epistemic theory of computation was given in [45].

3 Computation - concepts and definitions

We now digress on the philosophy given in Section 2, to obtain an understanding of computation in a broadest possible way. Furthermore, we will demonstrate a possible theoretical framework that reflects it. Before we can do so, we introduce the kinds of spaces which an observer needs to distinguish.

3.1 Essential spaces and maps

In this section we define *action spaces*, *knowledge spaces* and the maps that connect them, the so-called *semantic maps.*

Action spaces When can we say that actors act like computational processes? We want to remain at a very abstract level as regards their underlying mechanisms, but some assumptions must be made about the 'world' in which they live. In [45] we argued that computations take place in *action spaces*, consisting of *action items* of some descriptive nature. Action spaces were assumed to be *topological* spaces, to reflect a presumed proximity relation among the action items in the space.

In this report we go one step further and assume that, to an observer, action items are like symbolic complexes of some kind, which he can interpret. The more precise format of the action items will depend entirely on the way the observer understands the items. It can differ widely from process to process.

Example 1. The action items corresponding to (the functioning of) a *cell* may consist of all connections, tables and measurements of biological compounds that are deemed to be of interest. An action item holds all information about the cell and its direct environment as needed for understanding it at some moment in its life (or, during its 'operation'). By definition, the action space of a cell may be uncountable.

Example 2. A well-known model of intelligent systems consists of agents whose actions can be directed according to certain 'beliefs', 'desires' and 'intentions' [10]. The action space of a *BDI-agent* consists of items that contain the (values of the) attributes, data, and expressions corresponding to its operation and current plan at any moment in time, including representations of its beliefs, desires and intentions. Assuming that an agent is observed only at discrete time intervals, its action space is countable. The action space of a *group* of BDI-agents includes the individual action items of the agents, now extended with characteristics about their observed interrelation, interaction and commonalities as well.

Action items could well be unbounded, or infinite objects. This may be due to the extent of the items themselves, or to the nature of certain attributes in it. For example, some features might be real-valued or involve unbounded lists. We will not rely on any specific structure of the action items, but merely assume that one can define a reasonable notion of 'distance' between them that accounts for the (extent of the) observable differences between between - it is not strictly needed for the theory but makes it more intuitive. In fact, we assume the following strengthening of our earlier assumption that action spaces are topological.

Action spaces are metric spaces, with the topology induced by the metric.

We use the assumption for qualifying the 'distance' from one action item to a next, as it seems imperative that computations can only bridge small distances at a time. We note here that metric spaces are *Hausdorff*, i.e. any two distinct action items have disjoint open balls around them. Thus, a computation affects the entire open ball around an item 'in one step'. (See also [31] for a different use of metric spaces in computation.)

It is up to the observer to define the action space that is ideally suited for the computational process he wants to study. Action spaces are always required to be metric, even though we do not need the assumption when we are merely interested in combinational properties of computations only. If we are only interested in 'small effects', we may replace a given metric $d(x, y)$ by a curtailed version $d_\epsilon(x, y) = \min\{d(x, y), \epsilon\}$ (which is a metric again, for any $\epsilon > 0$). We do not require action spaces to have any special properties up front, such as *separability* or *compactness.*

If some distances would have to be 'undefined', one could assume an action space to be a *generalized metric space.* A generalized metric is a distance function on pairs of points whose range is part of $\mathbb{R} \cup \{+\infty\}$ but which still satisfies the usual metric properties. We will not pursue this here.

Knowledge spaces We stated that computations must lead to 'items of reasoning' to their observer. It means that observers must have a reasoning capability, to understand what is being produced in action space and derive conclusions from it. Consequently, an observer must possess a *theory* for dealing with the (contents of the) action items in the action space. This theory could be formal or informal and may have guided the very definition of the action space to begin with.

To keep this as general as possible, we rather refer to the observer as reasoning with 'knowledge items'. These items may be described as the 'observed contents' of those action items which the observer can interpret while the computational process produces its effect in the action space. The collective domain of knowledge items will be termed a *knowledge space* rather than a theory. We make the following assumption.

Knowledge spaces are equipped with a formal or informal system for reasoning with and applying the knowledge items they contain.

The term 'knowledge' is used freely here, to refer to any form of information that can be interpreted and reasoned about by the observer. Given a knowledge space K, the system of reasoning can be logic-based or otherwise, but is always assumed to involve some kind of *entailment* or *inference* relation which preserves the 'validity' of knowledge.

Definition 1. *An inference relation $\models$ on K is any reflexive, transitive relation on the items of K.*

In general, a reasoning system may deal with finite groups of items simultaneously. Such groups of items can be considered as one, in a suitably redefined knowledge space. In the ASM framework [21] there is a knowledge space consisting of 'first-order structures', with a $\models$-relation determined by the possible transformations of one state into another by the execution of (a sequence of steps of) an ASM program. We do not digress on this here. In case $\models$ is only partially defined, we use $\models^\star$ to denote its reflexive, transitive closure.

By definition, the knowledge spaces we deal with are always *closed* under the entailment relation that applies to them. However, normally only a small part of a knowledge space is really 'known' or given beforehand (i.e., to the observer). In so-called 'theory-like' spaces we assume that this part is potentially sufficient for inferring all other knowledge in K.

Definition 2. *A knowledge space K is said to be* theory-like *if it contains a pre-defined (proper) subset K_0 of* core knowledge *such that K is the closure of K_0 under entailment.*

The knowledge spaces we encounter will normally be theory-like, with 'cores' that are easy to recognize for an observer. A *knowledge theory* is any sufficiently formalized, theory-like knowledge space.

Semantic maps Given an action space A, an observer needs to be able to *extract* (or 'see') the meaningful information ('knowledge') from those items that contain it. It is a fundamental aspect of the actor-spectator paradigm. Let E be the knowledge space employed by the observer.

Definition 3. *A* semantic map *from A to E is any* partial *mapping $\delta : A \to E$.*

Semantic maps relate action items to knowledge items, corresponding to how this is presumably done by an observer when he monitors an ongoing computation. We leave it open how observers actually *evaluate* semantic maps and employ the resulting knowledge items in the reasoning system.

Repeatedly applying δ to an ongoing computation could potentially alter the course of the computation, but we do not explicitly take this into account here. Note that $\delta(x)$ may be undefined for some $x \in A$, reflecting the fact that many action items may actually not contain knowledge that is ready for 'display'.

One may want a semantic map δ to preserve or translate certain properties of items in A into corresponding properties in E. We view this as a matter for the observer when he is defining his 'observatory'.

3.2 Computational processes and computation

We can now define computation, in a very general sense. Following the philosophy in Section 2, a computation always involves two spaces: the action space in which it is produced, and the knowledge space in which it gets its operational meaning. It also involves a computational process that actually generates the computation and an observer to give it meaning. We expand on this in a number of steps, to obtain a mathematically tractable notion.

What a computational process is First of all, we envision that a computation *winds* through action space in some way and is interpreted by the observer through the facilities of the relevant knowledge space. To formalize this, we rely on basic notions from metric topology to capture this approach. Let A be a metric space, with metric d.

Definition 4. *A* curve *(or:* path*) in A is any continuous function c with $c : [0, \infty) \to A$.*

Curves are normally defined only on bounded segments of $[0, \infty)$, but we view them as running on *ad infinitum* as a matter of principle. We rely on other mechanisms if curves are to be observed during bounded segments of 'time' only. (We could have defined curves equivalently as continuous functions $c : [0, 1) \to A$ but we do not regard $[0, 1)$ as an appropriate time domain for the purposes of this report. In other contexts, alternative definitions along this line may be meaningful.)

Notation 1 *For a given curve c, we denote $c^{init} = c(0)$.*

We can now define what we mean by a *computational process*. It is a central notion of this report, showing the actor-spectator paradigm in all detail. For our theory, it is immaterial whether a computational process actually corresponds to a physical realization. However, we contend that any computational process that does, admits a formalization of this kind.

Definition 5. *A* computational process *is any 5-tuple $P = \langle A, E, \delta, E_0, C\rangle$ where A is an action space, E is a knowledge space, $\delta : A \to E$ is a semantic map, $E_0 \subseteq E$ is a non-empty set of initial knowledge, and C is a collection of curves in A such that the following* consistency conditions *are satisfied:*

- *for every $c \in C$: $\delta(c^{init}) \in E_0$, and*
- *for all $t_1, t_2 \in [0, \infty)$ with $t_1 \leq t_2$: if $\delta(c(t_1))$ and $\delta(c(t_2))$ are both defined, then $\delta(c(t_1)) \models^\star \delta(c(t_2))$ in E.*

To keep the notation simple, we assume that the inference relation $\models$ on E is *implicit* in E's definition.

What a computation is Finally we can finally answer the question 'what a computation is', in the very general sense we have been aiming at. With all ingredients in place, the definition is easy to state.

Definition 6. *A* computation *in A is any curve c for which there is some computational process $P = \langle A, E, \delta, E_0, C\rangle$ such that $c \in C$.*

Definition 7. *Let c and c' be computations of the same computational process P. Then c' is said to be a* subcomputation *of c if there is a $T \geq 0$ such that $c'(t) = c(T + t)$, for all $t \geq 0$.*

Definition 8. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. Let $c \in C$ be a computation and $\alpha \in A$ an action item. Then c is said to* visit *α if there is a time $t \in [0, \infty)$ such that $c(t) = \alpha$.*

Note that there may be many, indeed even infinitely many times t such that $c(t) = \alpha$. Thus, if c visits β after it visits α, it may well visit α again. Later on we will distinguish computations based on the many ways in which they may 'visit' and explore items in A.

Constraints By imposing additional constraints, special properties of computational processes may be enforced. For example, suppose one wants to articulate that computations can make only a limited number of changes to action items of A per unit of time. This leads to the following definition, insofar as this property can be measured by the metric distance. The concept is well-known in metric geometry.

Definition 9. *A curve c is called* Lipschitz *if c is Lipschitz continuous, i.e. for all $t_1, t_2 \in [0, \infty) : d(c(t_1), c(t_2)) \leq \kappa |t_1 - t_2|$, for some constant $\kappa \geq 0$. In this case, κ is called a Lipschitz constant for c.*

Definition 10. *A computational process P is said to be* Lipschitz *if all curves in C are Lipschitz curves. If all curves in C are Lipschitz with the same constant, then P is said to be* uniformly Lipschitz. *A computation c is said to be* Lipschitz *if it is produced by a computational process P that is Lipschitz.*

Lipschitz continuity is a special case of *Hölder continuity* which, in turn, implies *uniform continuity.* Lipschitz curves are 'linearly bounded', in the sense that, if c is Lipschitz with constant κ, then $d(c(^{init}), c(T)) \leq \kappa \cdot T$ for any $T \geq 0$. The properties are mentioned only as examples of the kind of 'familial constraints' we will encounter later.

The general definition of computational processes is crucial for the remainder of this report. In section 4 we will give examples of all concepts that we defined.

Causal aspects It is implicit in the definition of computations that there is a causal explanation of how they produce whatever knowledge they produce. This is, in fact, an explicit requirement of computational processes in [48]. In this report we stay with the very same approach, requiring that the 'computations' we consider somehow result from a coherent scheme that can produce them. This is often all we need for a 'computational view'.

We make a number of further observations that relate to the diverse aspects of the issue of causality.

Proposition 1. *Let $P = \langle A, E, \delta, E_0, C \rangle$, let $\delta' : A \to E$ be such that $\delta' \sqsubseteq \delta$, and let $C' = \{c \in C \mid \delta'(c^{init}) \text{ is defined}\}$. If P is a computational process, then so is $P' = \langle A, E, \delta', E_0, C' \rangle$.*

The causal explanation of computations will normally be based on properties of the generating process, but alternate causes may exist. The following observation could be useful in this respect.

Proposition 2. *Let $\{c_i\}_{i=1,2,\cdots}$ be a sequence of computations from C, and assume that the sequence converges* uniformly *to a function $c : [0, \infty) \to A$. Then c is a curve in A.*

Proof. As both $[0, \infty)$ and A are metric spaces, all concepts make sense. The lemma now follows immediately from the *Uniform Limit Theorem* (cf. [30]). □

The Uniform Limit Theorem also implies that, if all curves c_i $(i \geq 1)$ are uniformly continuous and the sequence $\{c_i\}_{i=1,2,\cdots}$ converges uniformly, then the limit curve c is uniformly continuous again. As a special case, one may show that, if e.g. all curves c_i $(i \geq 1)$ are Lipschitz with a *same* constant K and $\{c_i\}_{i=1,2,\ldots}$ converges uniformly, then the sequence converges uniformly to a curve c that is Lipschitz again, with the very same constant K. Many more convergence results of this nature from topology may be applied here.

The proposition does not guarantee that limit curve c satisfies the consistency conditions again, nor that it is an element of C if it does. One needs additional properties of P if one is to draw any further conclusions here.

If no e.g. physically motivated explanation is available, a process that satisfies Definition 5 may be termed *symbolically computational*. We will only consider this in exceptional cases, e.g. when a process is constructed for a mathematical purpose. Criteria of *effectiveness* only enter if they need to be assumed by the observer. Here, we do not assume it, to maintain perfect generality. (For example, many natural processes do not normally fit this criterion but satisfy our definition.)

Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. For any computation $c \in C$, we define its *footprint* (or *range*) in the course of the computation to be the set $A_c = \{c(t) \mid 0 \leq t < \infty\}$

Proposition 3. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. For any computation $c \in C$, A_c is a separable subspace of A (hence homeomorphic to a subset of the Hilbert cube).*

Proof. As $[0, \infty)$ is a separable metric space, so is its image $A_c = c([0, \infty))$ under the continuous mapping c. By general topology, any separable metric space is homeomorphic to a subset of the Hilbert cube. $\square$

By a similar argument one sees that for any $T \geq 0$, the 'bounded' footprint $A_c(T) = \{c(t) \mid 0 \leq t \leq T\}$ is a *compact* subspace of A. Note that, by general topology, compact subspaces are separable as well.

Before proceeding, we make some more remarks about the definition of computational processes and their computations.

Definitional aspects Defining computations as curves $c : [0, \infty) \to A$ assumes implicitly that computations proceed continuously. This definition of computation corresponds to the notion of machinery as put forth in Turing's 1948 report [42]. It may seem that the requirement of 'continuity' complicates matters tremendously, but we will see in this report that it gives us all the generality we need for modeling computational processes. We note that 'real-continuous' curves may be necessary for some natural mechanisms [40].

One immediately notes that this is very different from the traditional approach in which computations are viewed as proceeding *stepwise*. As a case in point, the general ASM framework [21] still describes a computation as *a finite or infinite sequence of states corresponding to the elaboration of a program.* The state-based approach makes it an instance of a *discrete* process, at least in the view of the observer. Thus, in order to be sound, our definition requires that we settle the apparent dichotomy between discrete and continuous machinery. We will discuss this in great detail in Sections 6 and 7.

Defining computations as curves also implies an implicit orientation of computation in time. This reflects the (broadly viewed) condition of *serialisability* for computations with many components, in a way that is consistent with causal effects. Thus, memory, input and output are all viewed as being represented into the action items in A as 'strung together' by c and observed using δ, the semantic map. A similar principle is well-known for e.g. systems of distributed processes, which may be viewed as a single computational process in our framework (or, just as well, as a 'composition' of computational processes).

An important question is whether any 'familial constraints' should be satisfied by the computations of C (as curves). For example, one might want to impose that all computations of C are Lipschitz continuous or that C is *equicontinuous* as a family. However, we do *not* require computational processes or the multitude of computations they may generate to satisfy any of these familial constraints 'up front', so as to keep our theory perfectly general.

In later sections we will see that, by defining computations as curves, numerous relevant properties of computations can be expressed in a natural way. Even though metric properties may not grab the behaviour of a computational process fully, they may sufficiently narrow it down for analytic purposes. Our main interest is not in developing a new theory of curves but to explore their use in modeling computations and knowledge-generation.

Semantic aspects Given a computational process P, the semantic map enables an observer to 'read' the knowledge build-up during any computation $c \in C$, by just applying δ to $c(t)$ at times t, for $t \to \infty$. Potential side-effects are assumed to be implicit in the course of c. Semantic maps must be rich enough, to provide enough content for the consistency conditions to hold.

This (limited) observability of computations may not always suffice: an observer might well wish to observe the computation during 'windows of time', i.e. observe segments $\{c(u) \mid t - w \leq u \leq t\}$ of a computation c 'in one view' as t goes to ∞, for some fixed or flexible window size w. For now we stay with the basic principle of observing computations at arbitrary times t, leaving further aggregations aside.

The consistency conditions are crucial for a sound interpretation of the action of a process. The requirement that for every $c \in C$: $\delta(c^{init}) \in E_0$ reflects the idea that process is always observed from some 'time 0' onward. The allowable initial actions items should be observable, having δ-values that are *known* in advance, i.e. in E_0 (like instantiations of axioms). Note that this is the same as requiring that $c^{init} \in A_0$, where $A_0 = \{a \mid \delta(a) \in E_0\}$. We do not allow other sets of initial action items, as the observer would most likely not be able to recognize their elements.

Starting in c^{init} as stated does *not* imply, in general, that all inputs or environmental influences that c may experience during its 'lifetime' must be known in advance. New information may well come in later and outputs made known, in the later action items which c visits in the course of time. Thus, in general, many computations may start at the same initial item $a = c^{init}$.

The second consistency condition is the most far-reaching one. It requires that for any computation $c \in C$ and any $T \geq 0$ for which $\delta(c(T))$ is defined, the knowledge item $\delta(c(T))$ holds enough 'knowledge data' such that for all $t \geq T$, all knowledge items $\delta(c(t))$ can be derived from it within E. Only this way, a computation achieves its meaning as a computation within the knowledge space of the observer. If a computation could not be traced in the reasoning frame of the observer, it would just be *magic*.

The consistency of a computation c is rooted is the one, single knowledge item of E_0 that is assumed at its start. All further knowledge that is obtained 'down the line' should be logically derivable from this one item, if one interprets the consistency conditions as they stand. This requires that the reasoning framework of the observer, i.e. of E, is rich enough to take all possible effects of input and interaction with the environment into account that can possibly occur during the lifetime of a computation, and in fact do occur during c.

More generally, one may wish to ground a computation in a (finite) set of knowledge items, or relax the requirement of derivability 'along the line', by allowing more knowledge items from E_0 than just $\delta(c^{init})$ to be used without deriving them explicitly in earlier stages during the computation. Many of these adjustments can be accommodated by suitably redefining A, E, or even the elements of C.

Example 3. Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process, T a time moment. One may call a computation $c \in C$ *restartable* at time T, if the action item reached at time T holds all information for 'continuing' c after it is pre-empted at that time, i.e. if $\delta(c(T))$ is defined. If we let $c_T : [0, \infty) \rightarrow A$ denote the curve defined by $c_T(t) = c(t + T)$ for $t \geq 0$, then c_T indeed satisfies the consistency conditions from time $t = 0$ onward and thus is a computation again. Process $P' = \langle A, E, \delta, E_0', C'\rangle$ with $E_0' = \{\delta(c(T)) \mid c \in C \textit{ and } \delta(c(T)) \text{ defined}\}$ and $C' = \{c_T \in C \mid c \in C \textit{ and } \delta(c(T)) \text{ defined}\}$ generates precisely all computations of P that are restarted at time T after pre-emption.

3.3 What computations compute

It is now clear that, in order to state meaningful things about a computation, one should observe the computation as it is generated by the underlying computational process. This enables us, at last, to define *what a computation computes.*

Definition 11. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process, and let $c \in C$ be a computation. Then c is said to compute (or: generate) the set E_c defined by $E_c = \{\delta(c(t)) \mid 0 \leq t < \infty)\}$ $(= \delta(c))$.*

By the definition, the 'data' computed by a computation c consists precisely of all the knowledge which an observer can 'extract' from c during its lifetime. It formalizes the core dogma of the epistemic theory of computation, namely that *computations are acts of knowledge generation.*

Definition 12. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process, and let $e \in E$ be a knowledge item. Then e is said to be* computable *if and only if there is a computation $c \in C$ such that $e \in E_c$, i.e. such that there is a time $t \in [0, \infty)$ such that $\delta(c(t)) = e$.*

Considering how computations c unfold, an observer can only meaningfully extract information from them at 'observable times', i.e. at times t for which $\delta(c(t))$ is defined.

Definition 13. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process, and let $c \in C$ be a computation. The* observable trace *of c is the map $m_c : I_c \to A$ with $I_c = \{t \mid \delta(c(t))$ is defined$\}$ and for every $t \in I_c$, $m_c(t) = c(t)$.*

Thus, if an observer had any way of telling (e.g. by recognizing the action items for which δ is defined), he could restrict his activity to the times in the set I_c of a generated computation. This will become relevant in our discussion of 'discrete computations' later on. Note that, in the definition, that one always has $0 \in I_c$, for every computation c of P.

We defined E_c such that it consists of all knowledge computed by c on the way, not just the knowledge that is achieved in a final stage of c (whatever that may mean) nor the knowledge as it may be 'seen' by the observer through some additional *filter*. We leave these options aside, insofar as they do not directly follow from a modeled computational view.

Definition 14. *Let $P = \langle A, E, \delta, E_0, C \rangle$ and $P' = \langle A', E, \delta', E_0', C \rangle$ be computational processes. P and P' are said to be* equivalent *(notation: $P \equiv P'$) if for every $c \in C$ there is a $c' \in C'$ such that $E_c = E_{c'}$, and vice versa.*

We now show that any computational process P can be modified such that for any of its computations c, the modified computation does not only produce $\delta(c(t))$ at time t but the entire cumulative set $\{\delta(c(T)) \mid 0 \leq T \leq t\}$ of 'intermediate knowledge' up to time t, for any $t \geq 0$.

Theorem 1. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. Then there exists a computational process $P' = \langle A', E', \delta', E_0', C' \rangle$ such that the following holds: for every $c \in C$ there is a $c' \in C'$ such that for all $t \geq 0$, $\delta'(c'(t)) = \{\delta(c(T)) \mid 0 \leq T \leq t\}$, and all computations of P' are obtained this way.*

Proof. Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. Define A', E', δ', E_0', and C' as follows:

- let A' be the family of non-empty, compact subsets of A. The Hausdorff distance for compact sets in A, turns A' into a metric space.
- let E' be the family of non-empty subsets of E. Define a relation $\models'$ on E' such that for any two elements E_1, E_2 of E': $E_1 \models' E_2$ if and only if for every $e \in E_2$, $e \in E_1$ or there exists a $e' \in E_1$ such that $e' \models^\star e$ in E. This extends $\models$ to an inference relation on subsets of knowledge items of E, turning E' into a valid knowledge space.
- define the semantic map $\delta' : A' \to E'$ by $\delta'(B) = \{\delta(b) \mid b \in B\}$, for any subset $B \subseteq A$.
- for any computation $c \in C$, define $c' : [0, \infty) \to A'$ by $c'(t) = \{c(T) \mid 0 \leq T \leq t\}$. By continuity of c, $c'(t)$ is a compact subset of A and thus belongs to A', for any $t \geq 0$. With the Hausdorff metric in A', one easily sees that c' is continuous, hence a curve. Let $C' = \{c' \mid c \in C\}$.

– finally, let E'_0 be the subspace of E' consisting of the singleton sets $\{e\}$ with $e \in E_0$.

One easily verifies that, with these definitions, every $c' \in C'$ satisfies the consistency conditions, with δ' as the semantic map, E' as the underlying knowledge space, and E'_0 as the set of initial knowledge items.

We conclude that $P' = \langle A', E', \delta', E'_0, C' \rangle$ is a well-defined computational process. For any $c \in C$, the corresponding computation $c' \in C'$ satisfies $\delta'(c'(t)) = \delta'(\{c(T) \mid 0 \leq T \leq t\}) = \{\delta(c(T)) \mid 0 \leq T \leq t\}$. □

In later sections we will study many properties of the sets E_c, with c being generated by some computational process P. We will also be interested in the full knowledge potential of a process.

Definition 15. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. Then P is said to compute (or: generate) the set E_P defined by $E_P \equiv \bigcup\{E_c \mid c \in C\}$.*

Clearly, if an observer has access to all items in the action space to begin with, then all knowledge contained in it can be computed without any effort.

Proposition 4. *Let A be a metric space, E a knowledge space, and $\delta : A \to E$ a semantic map, all arbitrary. Then for every non-empty subset $F \subseteq A$, there is a symbolically computational process $P = \langle A, E, \delta, \delta(F), C \rangle$ such that $E_P = \delta(F)$.*

Proof. Let $F \subseteq A$. For any $a \in F$, define the map $c_a : [0, \infty) \to A$ by $c_a(t) = a$ (= constant). Clearly, every c_a is continuous and thus a curve. Let $C = \{c_a \mid a \in F\}$. As $E_0 = \delta(F)$, C satisfies the consistency conditions. Hence P is a computational process, and one easily sees that $E_P = \delta(F)$. □

Hence, all computational processes of interest will likely have an E_0 that is only a 'small' subset of $\delta(A)$, thus of E.

Finally, we define *universality* as the ability of a process to generate all knowledge items that are derivable in the knowledge space.

Definition 16. *A computational process $P = \langle A, E, \delta, E_0, C \rangle$ is called* universal *if $E_P = \{e \in E \mid e' \models^* e$ for some $e' \in E_0\}$.*

Note that, if E is theory-like, then $\{e \in E \mid e' \models^* e \text{ for some } e' \in E_0\} = E$.

3.4 Mappings between computational processes

We will frequently want to relate computational processes by means of mappings. The most common types of mappings we will use are transmorphisms, epistemorphisms, and the combination of the two, homomorphisms. We give their definitions in turn.

Basic maps Let $P = \langle A, E, \delta, E_0, C \rangle$ and $Q = \langle B, F, \mu, F_0, D \rangle$ be computational processes. There are many ways to map P to Q. The following definition can be viewed as the most basic one.

Definition 17. *Let $f : A \to B$ be continuous. We say that f is a transmorphism from P to Q ($f : P \to Q$) if for every $c \in C$, the curve $f \circ c$ is an element of D. If f is 'onto' and maps C onto D as well, then f is called an epi-transmorphism. If f has a continuous inverse, then it is called an iso-transmorphism.*

In the definition it is implicit that, if $f : P \to Q$ is a transmorphism, then for every c that is a computation of P, $f \circ c$ is a computation of Q. In the sequel we will use the definition also for mappings between processes that merely satisfy the requirements for symbolically computational processes.

Example 4. Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. There trivially are transmorphisms from P to $P \times P$ and from $P \times P$ to P (projection). However, P and $P \times P$ cannot be iso-transmorphic, as topology tells us that there can be no homeomorphism between a metric space A and its square (in general).

Epistemorphisms While one spectator may be observing a computation through the lens of a knowledge space E, another spectator may do the same from a different point of view, i.e. with a different knowledge theory in mind to explain what he observes. For example, he may have a different 'filter' to observe what is happening in a computation, as long as the consistency conditions remain satisfied in his theory.

In order to relate different views, we define the following 'structural morphism' between knowledge spaces E and F.

Definition 18. *An* epistemorphism *from E to F is any mapping $\tau : E \to F$ such that $\tau(E_0) \subseteq F_0$ and for all $\Phi, \Psi \in E$ we have that, if $\Phi \models^\star \Psi$ in E, then $\tau(\Phi) \models^\star \tau(\Psi)$ in F.*

Example 5. A *slide rule* can be seen as a computational process facilitating computations that lead from initial knowledge items $[a, b, \perp]$ to knowledge items $[a, b, a \cdot b]$. However, it can also be seen as facilitating computations that lead from initial knowledge items $[x, y, \perp]$ to knowledge items $[x, y, x + y]$. The epistemorphism that connects the two views interprets knowledge items $[a, b, d]$ as items $[\log_{10} a, \log_{10} b, \log_{10} d]$.

Epistemorphisms are a way of expressing that a different observer may 'see' another process in other terms. The following lemma expresses that computation is preserved under epistemorphisms.

Lemma 1. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process, F a knowledge space, and $\tau : E \to F$ an epistemorphism. Then $Q = \langle A, F, \tau \circ \delta, F_0, C\rangle$ is a computational process, and $E_Q = \tau(E_P) = \{\tau(E_c) \mid E_c \in E_P\}$.*

Proof. One easily verifies that Q satisfies the requirements of a computational process. By the definition of epistemorphisms, the consistency conditions that hold in P, hold in Q as well. The equality of Q and $\tau(E_P)$ follows because Q has the same computations as P. □

Homomorphisms With the concepts as we have them now, we can finally define a suitable notion of *homomorphism* for computational processes. Homomorphisms aim to relate the computational nature of different processes, by 'translating' both the curves and the knowledge build-up of one process into those of another. In other words, what transmorphisms and epistemorphisms do separately, is combined in the operation of a homomorphism.

Let $P = \langle A, E, \delta, E_0, C\rangle$ and $Q = \langle B, F, \mu, F_0, D\rangle$ be computational processes.

Definition 19. *A* homomorphism *from P to Q is any pair $h = (f, \tau)$ such that $f : A \to B$ is a transmorphism from P to Q, $\tau : E \to F$ is an epistemorphism from E to F, and the following commutative property holds: $\tau \circ \delta = \mu \circ f$.*

We will also use the definition for mappings between processes which are only seen as being computational symbolically, i.e. purely as a mathematical construct only.

The key property of homomorphisms is expressed in the following theorem, which generalizes Lemma 1.

Theorem 2. *Let $P = \langle A, E, \delta, E_0, C\rangle$ and $Q = \langle B, F, \mu, F_0, D\rangle$ be computational processes, and let $h = (f, \tau)$ be a homomorphism from P to Q. Then $\tau(E_P) = \{\tau(E_c) \mid E_c \in E_P\} \subseteq E_Q$. If f is epi, then $\tau(E_P) = E_Q$.*

Proof. Let $c \in C$ be any computation of P and let $a \in A$ be any action item for which $\delta(c(a))$ is defined (and thus an element of E_c). As $f : A \to B$ is a transmorphism, we know that $f \circ c$ is a computation of Q, i.e. $f \circ c \in D$. By the commutative property of the constituent mappings we have

$$\mu(f \circ c(a)) = (\mu \circ f)(c(a)) = (\tau \circ \delta)(c(a)) = \tau(\delta(c(a)).$$

This means that $\tau(\delta(c(a)) \in E_{f\circ c}$ and every element of $E_{f\circ c}$ is obtainable this way. Thus $\tau(E_c) = E_{f\circ c}$ and, consequently, $\tau(E_P) \subseteq E_Q$.

If f is epi, then all computations of D are of the form $f \circ c$ for some $c \in C$. In this case we have $\tau(E_P) = E_Q$. □

We distinguish the usual special cases of homomorphisms as needed. For example, a homomorphism will be called an *epimorphism* if both its constituents are epi. Note that, if f is epi, then by Theorem 2 we may as well restrict F to $\tau(E)$ and consider τ to be epi as well. If $h = (f, \tau)$ maps P to Q and f is epi, then Q is called a *homomorphic image* of P.

We will see important examples of homomorphic relationships between computational processes in Section 6.

3.5 Determinacy

The curves that constitute the computations of a computational process P offer but a 'view' of what goes on as the process develops in A. The curves may well be projections of much deeper lying complex phenomena and environmental interactions. Consequently, if two curves $c, d : [0, \infty) \to A$ intersect at time t, they need not coincide from that time onward.

Notation 2 *For any curve c, let c^t denote the curve from t onward.*

We note here that $c^{t_1} = d^{t_2}$ is short for saying that for every $\rho \geq 0$, $c(t_1 + \rho) = d(t_2 + \rho)$.

If computations follow the same course after they intersect in some action item, it may be seen as a sign of determinacy.

Definition 20. *A computational process $P = \langle A, E, \delta, E_0, C\rangle$ is said to be* operationally deterministic *if for every two curves $c, d \in C$ and all $t_1, t_2 \in [0, \infty)$ one has that, if $c(t_1) = d(t_2)$, then $c^{t_1} = d^{t_2}$.*

If P is operationally deterministic and we have computations $c, c' \in C$ such that c' 'visits' c^{init}, then c necessarily is a subcomputation of c'.

Definition 21. *A computational process $P = \langle A, E, \delta, E_0, C\rangle$ is said to be* observationally deterministic *if for every two curves $c, d \in C$ and all $t_1, t_2 \in [0, \infty)$ one has that, if $\delta(c(t_1))$ and $\delta(d(t_2))$ are both defined and $\delta(c(t_1)) = \delta(d(t_2))$, then $c^{t_1} = d^{t_2}$.*

In stead of implying equality of c^{t_1} and d^{t_2} in definition 21, we could have settled for the weaker conclusion that c and d generate the same knowledge after t_1 and t_2, respectively. We will not require this here. (See also Subsection 7.1.) If a process is observationally deterministic, then this localizes the actual generation of every knowledge item quite precisely.

Proposition 5. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be observationally deterministic. If a knowledge item e of E is computable, then there is a unique action item α that is reachable by a computation of C and has $\delta(\alpha) = e$. Moreover, all computations of C that visit α follow the same trajectory afterwards.*

Proof. This follows directly from Definition 21. □

Proposition 6. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process, and assume that $\delta : A \to E$ is at least defined in all points of A that are ever visited by a computation in C. Then, if P is observationally deterministic, P is also operationally deterministic.*

Proof. This follows directly from Definitions 20 and 21. □

The following observation characterizes those computations that are operationally deterministic. The result is elementary but we give a complete proof.

Theorem 3. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally deterministic. Then every computation c of P is either* loop-free *(i.e. visits every item of A at most once),* ultimately constant *(i.e. consists of a loop-free initial part ending in one point where c remains constant), or* ultimately periodic *(i.e. consists of a loop-free initial part followed by a non-trivial cycle).*

Proof. Let c be any computation of P. Assume that c is not loop-free, i.e. suppose that for some $T \geq 0$ there is an $\epsilon > 0$ such that $c(T) = c(T + \epsilon)$. By operational determinacy, $c(t) = c(t + \epsilon)$ for all $t \geq T$.

Claim. Either $c(t)$ is constant for $t \geq T$, or it is not and there is a smallest $\epsilon > 0$ such that $c(t) = c(t + \epsilon)$ for all $t \geq T$.

Proof. Let $S = \{\epsilon > 0 \mid c(t) = c(t + \epsilon)$ for all $t \geq T\}$, and let η be the greatest lower bound of S. If $\eta \in S$, we are done. Thus, assume $\eta \notin S$. Let $\epsilon_1 > \epsilon_2 > \cdots$ be a decreasing sequence of elements of S that converges down to η. Note that $c(t) = c(t + \epsilon_1) = c(t + \epsilon_2) = \cdots$. By continuity of c, we have $c(t) = \lim_{k\to\infty} c(t + \epsilon_k) = c(t + \eta)$. When $\eta > 0$, this proves our claim.

If $\eta = 0$, we claim that now $c(t') = c(t)$ for every $t' \geq t \geq T$. To show it, consider any $t' > t$. We construct a decreasing sequence $\nu_1 > \nu_2 > \cdots$ step by step, such that $c(t') = c(t + \nu_k)$ and $0 \leq \nu_k \leq \epsilon_k$ for all steps k we complete. Starting with $k = 1$, write $t' = t + \lambda\epsilon_1 + \nu_1$, with integer λ such that $0 \leq \nu_1 \leq \epsilon_1$. Clearly $c(t') = c(t + \nu_1)$. If $\nu_1 = 0$ we have $c(t') = c(t + 0) = c(t)$ and we are done.

Continuing inductively, suppose we have completed the $(k-1)$-st step ($k \geq 2$) of the construction and obtained the $(k-1)$-st element of the sequence with $0 < \nu_{k-1} \leq \epsilon_{k-1}$. Choose an $l \geq k$ such that $\epsilon_l \leq \nu_{k-1}$. (Such an l must exist as the sequence $\epsilon_1, \epsilon_2, \cdots$ decreases towards 0.) Write $\nu_{k-1} = \lambda\epsilon_l + \nu_k$, with integer λ such that $0 \leq \nu_k \leq \epsilon_l$. Clearly $c(t') = c(t+\nu_{k-1}) = c(t+\nu_k)$. If $\nu_k = 0$ we have $c(t') = c(t + 0) = c(t)$ and we are done. Otherwise we note that we constructed the k-th entry in the sequence, where $0 < \nu_k \leq \epsilon_l$ and thus $0 < \nu_k \leq \epsilon_k$ and $\nu_k < \epsilon_l \leq \nu_{k-1}$.

We conclude that we either find that $c(t') = c(t)$ as desired, or obtain an infinite decreasing sequence $\nu_1 > \nu_2 > \cdots$ such that $c(t') = c(t + \nu_k)$ and $0 < \nu_k \leq \epsilon_k$ for all $k \geq 1$. As this sequence is majorized by the sequence of ϵ's, we have $\lim_{k\to\infty} \nu_k = \lim_{k\to\infty} \epsilon_k = 0$ and by continuity it follows that even in this case $c(t') = \lim_{k\to\infty} c(t + \nu_k) = c(t + 0) = c(t)$.

It follows that c either leads into an item where it 'stays forever', or it leads into a non-trivial cycle, from which it will not escape. This completes the proof of the claim.

Next, in case c ends in a constant point, we argue that there is a smallest T such that for all $t \geq T$, $c(t) = c(T)$. For this, let $S = \{T \mid c(t) = c(T)$ for all $t \geq T\}$, a non-empty set. Let ρ be the greatest lower bound of S. If $\rho \notin S$, there must be a decreasing sequence $\rho_1 > \rho_2 > \cdots$ in S that converges down to ρ. Consider any t with $t > \rho$. Let l be an index such that $\rho_l \leq t$ (which must exist). Then we have $c(t) = c(\rho_l) = c(\rho_{l+1}) = \cdots$ and thus, by continuity of c, it follows that $c(t) = \lim_{l\to\infty} c(\rho_l) = c(\lim_{l\to\infty} \rho_l) = c(\rho)$. Thus $c(t) = c(\rho)$ for all $t \geq \rho$ and, hence, $\rho \in S$, a contradiction. Thus $\rho \in S$. It follows that c consists of a loop-free part until $t = \rho$ and that $c(t) = c(\rho)$ for all $t \geq T$.

Finally, in case c ends in a non-trivial cycle, one can argue in a very similar way that there must be a smallest T such that for all $t \geq T$, $c(t) = c(t + \epsilon)$. Here ϵ is the smallest cycle-length as established in the claim. (One easily sees that this ϵ is unique and identical for all T that lie on the cycle.) It follows that c consists of a loop-free part until $t = T$, followed by a cycle, i.e. such that $c(t) = c(t + \epsilon)$ for all t from then onward. □

If an operationally deterministic computation is ultimately periodic, its cycle is like an *orbit* of a system in dynamical systems theory. It serves as an *attractor* for the given computation.

Given the result of Theorem 3, it is only a small step to obtain a characterization of those computations that are observationally deterministic.

Theorem 4. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be observationally deterministic. Then every computation c of P is either a curve* along which no knowledge item is obtained more than once*, an* ultimately constant *curve (i.e. consisting of an initial part on which no knowledge item is obtained more than once but ending in a point where $\delta(c(t))$ is defined and c remains constant), or an* ultimately periodic *curve (i.e. with an initial part on which no knowledge item is obtained more than once followed by a non-trivial cycle which has at least some points where δ is defined but all the defined knowledge items along it are different).*

Proof. Let c be any computation of P. Assume that there are two different points on c where δ gives the same knowledge item, i.e. suppose that there are a $T \geq 0$ and an $\epsilon > 0$ such that $\delta(c(T))$ and $\delta(c(T + \epsilon))$ are both defined and $\delta(c(T)) = \delta(c(T + \epsilon))$. Then, by observational determinacy, we have $c(t) = c(t + \epsilon)$ for all $t \geq T$.

Now repeat the idea of the proof of Theorem 3, applying it to the set $S = \{\epsilon > 0 \mid c(t) = c(t + \epsilon)$ for all $t \geq T$ and $\delta(c(t)$ defined$\}$. We conclude that either c has an initial part on which no knowledge items can occur more than once and there is a T with $\delta(c(T))$ defined and $c(t) = c(T)$ for all $t \geq T$, or c leads into a cycle of 'length' $\epsilon > 0$ and there is T with $\delta(c(T))$ defined and $c(t) = c(t + \epsilon)$ for all $t \geq T$.

By closer scrutiny of the remaining part of the previous proof, one may argue that T can be chosen such that the part of c for t leading up to T satisfies the constraint as stated in the theorem. □

4 Computation - examples

The definition of computational processes and of computation was largely motivated on philosophical grounds. We now illustrate the concepts with a number of examples that show the versatility of the approach. We include several examples of discrete computation, in anticipation of the more general treatment of the very concept in Section 6.

In presenting the latter, we make use of *discrete records* and *traces*. A discrete record is any map $d : \mathbb{N} \to A$, which presumably represents some computation $c : [0, \infty) \to A$ which is only observed at integer times. If it does, the record is called a discrete trace. As a rule, only the discrete observations are known.

Given a discrete record d (of observations), it will be a trace only if there is a computational process that somehow bridges the gaps from $d(t)$ to $d(t + 1)$, for all $t \in \mathbb{N}$, i.e. during the intermediate times. We often require that a trace is *faithful*, i.e. that the underlying computation c fills gaps such that no same action items are visited in different gaps.

Definition 22. *Let d be a discrete record. Then d is a* discrete trace *of computation c if $c(t) = d(t)$ for every $t \in \mathbb{N}$. The trace is said to be* faithful *if for any $t_1, t_2 \in \mathbb{N}$ with $t_1 \neq t_2$, the segments $\{c(t) \mid t_1 < t < t_1 + 1\}$ and $\{c(t) \mid t_2 \leq t \leq t_2 + 1\}$ are all disjoint.*

We will give slightly more general definitions in Sections 6 and 7, when we deal with the connection between continuous and discrete computation in greater detail.

4.1 Finite-state systems

The first example deals with one of the most common models of computational action, namely *finite-state systems*. We will show that these systems fit naturally in our framework of computational processes.

For convenience we will model a finite-state system as a *Moore machine*, which associates 'output knowledge' to states rather than to transitions [29]. A Moore machine is a 6-tuple $M = \langle S, \Sigma, \Lambda, s_0, \tau, U \rangle$, where S is a finite set of states, Σ the (finite) input alphabet, Λ the (finite) output alphabet, $s_0 \in S$ the initial state, $\tau : S \times \Sigma \to S$ the state transition function, and $U : S \to \Lambda$ the output function.

After being initialized to state s_0, M reacts to a finite or infinite sequence of input symbols $\sigma_1\sigma_2\cdots$ by moving from state to state according to transition function τ. Whenever a state is entered, the machine produces an output symbol by calling on U. By allowing 'empty symbols' λ (with the obvious meaning), we may assume that M always processes *infinite sequences* of symbols $\sigma = \sigma_1\sigma_2\cdots$, where $\sigma_i \in \Sigma \cup \{\lambda\}$ for $i \geq 1$.

The processing of σ corresponds to a *directed path*, consisting of the consecutive states that are entered in the state transition diagram of M. This is a *discrete record* d_σ with $d_\sigma(t) =$ *'the t'th state that is entered in processing σ'* ($t \in \mathbb{N}$). Our aim is to show that the records are *faithful* discrete traces of the computations of a suitable computational process.

In order to prove this, we could simply embed the state transition diagram of M in the 2-dimensional plane and view d_σ as a 'sampling' of the curve that cycles through the points in the plane that correspond to the consecutive states that are passed in processing σ. However, when the state transition diagram is not planar, and most likely it isn't, this straight embedding would unavoidably lead to curves with segments that can intersect *outside* of the observed time moments $t \in \mathbb{N}$. It shows that the records are discrete traces of some process, but the traces so obtained are generally not faithful.

In stead, consider the action space $A = \mathbb{R}^3$, with the usual metric. We note the following basic fact from the theory of graph drawing.

Lemma 2. *The state transition diagram of every Moore machine (incl. λ-transitions) can be embedded in 3-dimensional space such that all states are mapped to grid points and all labeled arcs are mapped to disjoint piecewise-linear paths with at most two bends per path.*

Proof. Let $M = \langle S, \Sigma, \Lambda, s_0, \tau, U \rangle$ be a Moore machine. Extend the state transition diagram for τ by adding 'self-loops' labeled λ to every state. Let G be the resulting labeled directed graph.

Split all arcs of G by adding an auxiliary node and all self-loops by adding two. Let G' be the undirected graph obtained by subsequently omitting all labels and directions. G' is a simple undirected graph, with no parallel edges and also no self-loops.

Now, it is well-known that every undirected graph admits a *Fary embedding* in the 3-dimensional grid, with non-intersecting edges [13]. Consider this embedding of G', turn the auxiliary nodes into bending points, and add the labels and directions back in. This gives an embedding of G as claimed. $\square$

Embed the state transition diagram of M into $A = \mathbb{R}^3$ as specified in Lemma 2. Let E be the knowledge space consisting of the symbols of Λ, with the trivial inference relation. Define the semantic map $\delta : A \to E$ such that $\delta(a) = U(s)$ when point a corresponds to state s in the embedding and $\delta(a) = \mathit{undefined}$ otherwise. Let $E_0 = \{U(s_0)\}$.

Finally, consider all possible discrete records processed by M. View every record d_σ as a 'sampling' of the curve c_σ which cycles from state to state according to the processing of σ in the embedded state transition diagram, following the (disjoint) segments that connect the states during the gaps. Let C consist of all curves that can be obtained this way. Trivially, all $c \in C$ satisfy the required consistency conditions.

We conclude that the defined process $P = \langle A, E, \delta, E_0, C \rangle$ is (symbolically) computational. The discrete records of M are seen to be faithful discrete traces of the computations of P. We conclude:

> *every finite-state system is a (symbolically) computational process, observed at discrete time moments.*

We note that all computations produced by P are piecewise-linear curves, all with bending points on the grid.

In Section 9 we show that similar conclusions hold for all kinds of *infinite-state systems* as well. This includes many computing systems as they are known today, from theoretical models like Turing machines and extended versions of it, to practical devices like digital computers and the internet.

4.2 Analog computers

The next example deals with the domain of analog computation [25]. As opposed to digital computers, analog machines make inherent use of continuous variables. We will argue that our framework includes analog computation in a natural way. We base the example on a rendering of digital computing in [45].

We first need a reference model of analog computation. At an abstract level, we may view an analog computer as consisting of a finite number of physical registers that can all be adjusted by a continuous process (only). Registers can contain any real number, possibly from a qualified set, and may be used for computing or control as in digital machines. Programs can trigger chains of operations on the registers of the analog computer, where the adjustments of the registers proceed continuously in time and, when imposed, within the constraints set for them.

Consider an analog machine M, and let $J, I_1, \cdots, I_k$ be the registers of M ($k \geq 1$). Here J is a control register, I_1 the input register, I_2 up to I_{k-1} are intermediate registers, and I_k is the output register. Let S be a set of allowable programs. Programs $\pi \in S$ running on M can be seen as performing continuous transformations of the contents of $I_1, \cdots, I_k$ in according to the instructions of the program. We assume that the J register is changed (from 0 to 1) only in the last instruction of a program, if the program ends at all. For control purposes we assume that the machine has a continuous clock that starts at $t = 0$. Our aim is to show that the operation of the programs of M corresponds to the computations of a computational process P.

To achieve this, we define the components of P that we need for it. First of all, let the action space A equal $S \times [0, \infty) \times \mathbb{R}^{k+1}$. With the discrete metric on S and the usual metrics on $[0, \infty)$ and $\mathbb{R}$, A becomes a metric space (with the product metric). An action item $\langle \pi, t, J, I_1, \cdots, I_k \rangle$ essentially represents the snapshot of a program π at time t, executing its $\lfloor t \rfloor + 1$-st instruction and with register contents at time t equal to $J, I_1, \cdots, I_k$. When π is run on M with input $a \in \mathbb{R}$, it starts with action item $\langle \pi, 0, 0, a, 0, \cdots, 0 \rangle$ (thus with $t = 0$, $J = 0$ and $I_1 = a$). We assume that the $\lfloor t \rfloor + 1$-st instruction of π begins at time $\lfloor t \rfloor$ and gradually changes the contents of $I_1, \cdots, I_k$ by its choice of continuous operations of M. It completes by time $\lfloor t \rfloor + 1$, and then the next instruction is begun, and so on. All this time, the J-register remains unchanged.

If a program π happens to come to a final instruction, say at time $\lfloor t \rfloor$, we assume that the J-register is changed as well, from 0 to 1 (by a continuous process). In this way, termination of a program becomes 'noted' in an action item. We assume that, after the final instruction is completed, the program 'cycles' from then onward, by letting the t-value run on forever without changing the registers any further. Clearly, the action items provide the proper information for an observer to monitor the programs.

Let E consist of the knowledge items $\langle f, a, \Box \rangle$ ($\Box$ meaning 'no knowledge') and $\langle f, a, b \rangle$, where $f : \mathbb{R} \to \mathbb{R}$ is any partial function, $a \in \mathbb{R}$, $b \in \mathbb{R} \cup \{\bot\}$ and $f(a) = b$. E is the knowledge space of all single-parameter partial functions, with a trivial reasoning system based on straightforward inferences like $\langle f, a, \Box \rangle \models \langle f, a, b \rangle$ when $f(a) = b$.

Let the spaces A and E be linked by a semantic map δ defined as follows:

$$\delta(\langle \pi, t, J, I_1, \cdots, I_k \rangle) = \begin{cases} \text{if } t = 0,\ J = 0,\ I_1 = x \text{ and } I_j = 0\ (2 \leq j \leq k), \\ \quad \text{indicating that execution of } \pi \text{ is initialized:} \\ \quad \langle f_\pi, x, \Box \rangle \\ \text{if } J = 1, \text{ and } y \text{ appears in the output register } I_k\text{:} \\ \quad \langle f_\pi, x, y \rangle \\ \text{otherwise:} \\ \quad \textit{undefined} \end{cases}$$

where f_π is the partial function determined by program π. Furthermore $E_0 = \{\langle f_\pi, x, \Box \rangle \mid \pi \in S, x \in \mathbb{R}\}$.

For any program $\pi \in S$ and input $x \in \mathbb{R}$, define $c_{\pi,x} : [0, \infty) \to A$ as the map with

> $c_{\pi,x}(t)$ =“*the action item* $\langle \pi, x, J, I_1, \cdots, I_k \rangle \in A$*, reached at time* t *in the execution of* π *on* M*, on input* x*.*”

By the chosen metric in A, every $c_{\pi,x}$ is continuous and thus defines a curve in A. One also sees that the consistency conditions are trivially satisfied along this curve. Hence, taking $C = \{c_{\pi,x} \mid \pi \in S, x \in \mathbb{R}\}$, we conclude that $P = \langle A, E, \delta, E_0, C \rangle$ is a *computational process.* It follows that every $c_{\pi,x}$ is a computation.

It remains to confirm that the computations of P deliver what the observer wants to ‘know’. Given $\pi \in S$ and input $x \in \mathbb{R}$, we see that the knowledge computed by $c_{\pi,x}$ (or, as we might say, by π on input x) is equal to $\{\langle f_\pi, x, \Box \rangle\}$ if π does not terminate on x (i.e. if the J register remains 0), and equal to $\{\langle f_\pi, x, \Box \rangle, \langle f_\pi, x, y \rangle\}$ if π does terminates on x (i.e. if register J turns into 1 in finite time) with defined output. The former corresponds to $f_\pi(x) = \bot$, the latter to $f_\pi(x) = y$.

We conclude that, under the assumptions as we made them, *analog computation is computation*, according to our definition. We note that the same conclusion can be obtained for many other systems in which physical or chemical quantities are traced and evolved by explicit or implicit ‘programs’, including *metabolic processes* in cells, *neural networks*, and all kinds of *molecular machines.*

In the example, the observer will gather only little information about whatever is ongoing in the computational process P. In a different scenario, i.e. by using a different semantic map, the spectator may observe a larger part of the action of a computation as it is unfolding in the action space. Whether this is feasible or not depends on the observational means that are available to an observer.

4.3 Reasoning

In the 17-th century, Leibniz dreamed of having formal systems that would enable one to express argumentations and verify these by ‘rational calculation’ [36]. Nowadays, automated reasoning systems and interactive proof assistants are standard computer applications. In the final example in this section we will show that *deductive reasoning* is an instance of (discrete) computation.

Consider any formal system F, with set of axioms S and set of inference rules D. Let the sentences of F be expressible using the symbols of a finite alphabet Σ, with $|\Sigma| = k$. A *reasoning* (or: proof) in F is any finite or infinite sequence $\pi = \pi_0, \pi_1, \cdots$ with the property that for every $i \geq 0$, π_i is either an element of S or follows by applying a rule from D to one or more sentences π_j with $j < i$. We assume that finite proofs are always extended to infinity, by simply repeating their last sentence ad infinitum, and, hence, that all proofs are infinite.

We envision that reasonings are generated routinely, in some targeted manner. Our aim is to show that the process of generating proofs π can be viewed as being computational.

We will deal with proof segments as 'strings' but only *after* identifying them with a counterpart in some metric action space. Let \$ and 0 ('zero') be new symbols, i.e. $\$, 0 \notin \Sigma$. Let B be the set of all strings $\sigma = \$\sigma_0\$ \cdots \$\sigma_i\$$, with $i \geq 0$ and $\sigma_0, \cdots, \sigma_i$ arbitrary sentences of F. Before defining the action space of choice, note that every string $\sigma \in B$ corresponds uniquely to a natural number $eval(\sigma)$, using $k+2$-ary notation with the digits of $\{\$\} \cup \Sigma \cup \{0\}$ (with \$ being the largest digit and 0 the smallest). Clearly:

$$eval(\$\sigma_0\$) < eval(\$\sigma_0\$\sigma_1\$) < eval(\$\sigma_0\$\sigma_1\$\sigma_2\$) < \cdots$$

Now let $A = \mathbb{R}^2$. Action items in A will be of the form $\langle r, y \rangle$, where r corresponds to some stage in 'computing' a proof and y can be seen as a control field that will tell the observer whether 'observable' progress has been made in the process. Let $str : A \to B$ be the 'retraction' defined by:

$$str(\langle r, y \rangle) = \begin{cases} \text{if } r \in \mathbb{N},\ y = 0 \text{ and } r = \$\sigma_0\$ \cdots \$\sigma_i\$ \text{ in } k+2\text{-ary notation:} \\ \quad \$\sigma_0\$ \cdots \$\sigma_i\$ \text{ (as a string)} \\ \text{otherwise:} \\ \quad \textit{undefined} \end{cases}$$

Note that $str(\langle eval(\sigma), 0 \rangle) = \sigma$, for all $\sigma \in B$. Also, for all $\langle r, y \rangle \in A$, if $str(\langle r, y \rangle)$ is defined, then $y = 0$ and $eval(str(\langle r, y \rangle)) = r$.

From an informational viewpoint, the i-th step of a proof π not only produces π_i but the entire multiset $\{\pi_0, \cdots, \pi_i\}$ of F ($i \geq 0$). It is thus natural to let E consist of all finite multisets of sentences of F. Now define relation $\models$ on E such that for any two multisets $E_1, E_2 \in E$: $E_1 \models E_2$ if and only if for every $\alpha \in E_2$, either $\alpha \in E_1$ or α follows by applying a rule from D to a number of elements of E_1. This gives a proper inference relation on subsets of F, thus turning E into a proper knowledge space.

In order for an observer to interpret the action items in A, let the semantic map $\delta : A \to E$ be defined as follows:

$$\delta(\langle r, y \rangle) = \begin{cases} \text{if } str(\langle r, y \rangle) \text{ is defined and equal to } \$\sigma_0\$ \cdots \$\sigma_i\$ \text{ (as a string):} \\ \quad \{\sigma_0, \cdots, \sigma_i\} \text{ (as a multiset)} \\ \text{otherwise:} \\ \quad \textit{undefined} \end{cases}$$

Note that $\delta(\langle r, y \rangle)$ is defined only if $r \in \mathbb{N}$ and $y = 0$, and that it 'extracts' exactly the multiset of sentences ('knowledge') that is encoded in r when its $k+2$-ary expansion is of the right form. Let E_0 consist of all singleton subsets of F.

For any proof $\pi = \pi_0, \pi_1, \pi_2, \cdots$ in F, define the map $c_\pi : [0, \infty) \to A$ as follows:

$$c_\pi(t) = \begin{cases} \text{if } t = i + \epsilon \text{ for } i \in \mathbb{N} \text{ and } 0 \leq \epsilon \leq \frac{1}{2}: \\ \quad \langle (1-\epsilon) \cdot eval(\$\pi_0\$ \cdots \$\pi_i\$) + \epsilon \cdot eval(\$\pi_0\$ \cdots \$\pi_{i+1}\$), 2\epsilon \rangle \\ \text{if } t = i + \epsilon \text{ for } i \in \mathbb{N} \text{ and } \frac{1}{2} \leq \epsilon \leq 1: \\ \quad \langle (1-\epsilon) \cdot eval(\$\pi_0\$ \cdots \$\pi_i\$) + \epsilon \cdot eval(\$\pi_0\$ \cdots \$\pi_{i+1}\$), 2 - 2\epsilon \rangle \end{cases}$$

Thus, c_π is the 'map' that leads from action item $\langle eval(\$\pi_0\$ \cdots \$\pi_i\$), 0 \rangle$ to action item $\langle eval(\$\pi_0\$ \cdots \$\pi_{i+1}\$), 0 \rangle$ for i from 0 to ∞, by moving by straight lines to and from action item $\langle \frac{1}{2} \cdot (eval(\$\pi_0\$ \cdots \$\pi_i\$) + eval(\$\pi_0\$ \cdots \$\pi_{i+1}\$)), 1\rangle$ in 2-space, for every i.

With the common metric in A, every c_π is continuous and thus a curve in A. Also note that the action items $\langle r, y\rangle$ on the path of c_π for which $\delta(\langle r, y\rangle)$ is defined are precisely the action items of the form $\langle eval(\$\pi_0\$ \cdots \$\pi_i\$), 0\rangle$ $(i \geq 0)$, and these items have δ-value equal to $\{\pi_0, \cdots, \pi_i\}$. It follows that the consistency conditions are satisfied along every curve c_π.

Taking $C = \{c_\pi | \pi \text{ is a proof in } F\}$, we can conclude that $P = \langle A, E, \delta, E_0, C\rangle$ is a valid (symbolically) *computational process*. Hence, every c_π is a *computation*, namely the one that gives the observer all he needs to know in order to infer the consecutive steps of π. It is useful to look at his more closely.

Note that observer only needs to observe a computation c_π at *integer* times. Namely, he would only need the discrete trace $d_\pi : \mathbb{N} \to A$ with $d_\pi(i) = c_\pi(i) = \langle eval(\$\pi_0\$ \cdots \$\pi_i\$), 0\rangle$ for i from 0 to ∞. As c_π does not intersect itself, every record d_π is not just a discrete trace but a *faithful discrete trace* of c_π. We may as well define $d_\pi(i) = eval(\$\pi_0\$ \cdots \$\pi_i\$)$ and declare d_π to be a faithful *projection* of c_π. This property is the hallmark of 'discrete computation'. This will be analyzed in great detail in Section 9.

4.4 Reflection

The examples illustrate the *duality* of the framework as we defined it. On the one hand there is an 'actor' operating in action space, on the other hand there is the 'spectator'. The action spaces are the realm of 'computation' but, clearly, not every curve qualifies for it. Which curves do, is fully determined by the spectator who must be able to make sense of them. In particular, for the generative process to be computational, it is required that the reasoning system of the spectator is able to see the outcomes as generated knowledge. It conforms to the earlier view that the notion of computation is inherently *observer-dependent*.

In the examples we chose to illustrate the mathematical framework of computational processes as defined. The (logical or physical) realizability of the computational processes we defined is assumed to be implicit in the cases as they are given. In other words, we do not create new processes but merely 'mold' existing ones in the right frame in order to be viewed as being computational.

5 Operations on computational processes and computations

In this report we view computation as a *phenomenon*, interpreted and signified by an observer using a suitable knowledge theory. We gave a broad definition of the concept of computational process, to describe the generation of computations in a tangible way. Computations are then 'observed' as curves in a suitable action space which satisfy the consistency conditions for knowledge generation, according to the criteria of (the theory of) the observer. It gives a philosophical grounding of the concept of computation, in the spirit of Turing's suggestion from 1948.

We will now expand on the approach. Our aim is to treat computational processes and computations as *mathematical abstractions* and explore their key properties as recognized from an observer's perspective. This will lead to a mathematical theory that is very different from the classical 'theory of computation', focused more on computation as a notion. The theory will also enable us to address the detailed connection between discrete and continuous computation.

In this section we will first consider various kinds of *constructions* which one may carry out. We illustrate this by various examples such as the composition of processes and some ways of combining knowledge sets. Ultimately, we end up with a number of techniques for creating new (artificial) computational processes from old ones.

5.1 Re-timing computations

While computational processes are basic, they do not tell us much about the way the individual computations are actually generated. The main computational activity as it is observed will appear to the observer as a continuous transformation of items in the action space, but most likely other features are observed as well. Examples include the timing of the computation, and the use of various resources (i.e. by the computational process). This suggests various ways of manipulating processes.

A typical type of transformation is the following. It expresses the possibility of accelerating or decelerating computations. Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. For any function $g : [0,\infty) \to [0,\infty)$, let $C_g = \{c \circ g \mid c \in C\}$.

Lemma 3. *Let $g : [0,\infty) \to [0,\infty)$ be continuous and non-decreasing, with $g(0) = 0$. Then $P_g = \langle A, E, \delta, E_0, C_g\rangle$ is a (symbolically) computational process, and $E_{P_g} \subseteq E_P$. If g is strictly increasing, then $E_{P_g} = E_P$.*

Proof. Clearly, for any $c \in C$, $c \circ g$ is a curve in A again. The fact that g is also non-decreasing and has $g(0) = 0$ guarantees that, if c is a computation, then so is $c \circ g$. This follows by checking that the consistency conditions for $c \circ g$ continue to be satisfied. It follows that P_g is computational, at least symbolically. We clearly have $E_{P_g} \subseteq E_P$, as all computations of P_g are tracing computations of P. If g is $1-1$, then $E_{P_g} = E_P$. □

A function g as in Lemma 3 may be seen as a 'dial' for controlling the speed of P. If the dial is turned up $(g(t) > t)$ then P runs 'faster', if the dial is turned down $(g(t) < t)$ then P runs slower. Note that re-timing has no effect on the knowledge that is generated in a computation, only on the moments in time that specific knowledge items are produced - if they are.

Definition 23. *Let $g : [0,\infty) \to [0,\infty)$ be continuous and non-decreasing, with $g(0) = 0$. The P_g is called a* re-timing *of P. If g is $1-1$, then we speak of a strict re-timing.*

The following examples illustrate the concept of re-timing.

Example 6. Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. Let function $g : [0, \infty) \to [0, \infty)$ be defined by: $g(t) = \beta t$ for $t \geq 0$, for some $\beta > 0$. The (strict) re-timing of P by means of g transforms any computation $c \in C$ into a computation c' defined by $c'(t) = c(\beta t)$. Computation c' may be seen as the 'speed-up' (if $\beta > 1$) or 'slow-down' (if $\beta < 1$) of c by a factor β.

Example 7. Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. Let function $g : [0, \infty) \to [0, \infty)$ be defined by: $g(t) = t$ for $0 \leq t \leq 1000$ and $g(t) = 1000$ for $t > 1000$. Re-timing computations by g means that every curve c is transformed into a curve $c' : [0, \infty) \to A$ defined by $c'(t) = c(t)$ for $0 \leq t \leq 1000$ and $c'(t) = c(1000)$ for $t > 1000$. If c is a computation, then so is c'. Hence, by Lemma 3, process $P_g = \langle A, E, \delta, E_0, C_g\rangle$ with $C_g = \{c' \mid c \in C\}$ is symbolically computational. One may argue that it is, in fact, computational. The 'dial' g transforms P into a process P_g that 'freezes' every computation of P at time 1000. (Note that two computations c that are identical up to time 1000 but differ later, are mapped to the same c'.)

5.2 Switching and composing computations

Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. In order to generate a computation d, P presumably has to be set to an initial position d^{init} and act from there. If this initial position is not readily available, one may attempt to 'jump-start' d, by calling on an auxiliary computation c to lead up to this action item first. Could this be done as a single computation?

It is implicit in the way we defined computation that, once it is produced by P, a computation c unfolds as a curve in time uninterruptedly. If we choose to use c as suggested above, or experiment with it otherwise, then we will need to halt c at some point so we can switch over to another computation d of P (or of another process altogether), from there onward. The idea is that c brings P to an observable point x in A, with $x = d^{init}$ and thus, with $\delta(x) \in E_0$. This may be repeated.

These considerations leads us to the notion of *composition*, both for pairs of curves and for sets, as follows. (A first version was given in [45].)

Definition 24. *Let c, d be curves in A and let $T \in [0, \infty)$. If $c(T) = d^{init}$ and $\delta(c(T))$ is defined, then $c \circ_T d$ is the curve defined by*

$$c \circ_T d\ (t) = \begin{cases} \text{if } \ 0 \leq t \leq T \ \text{ then:} & c(t) \\ \text{if } \ t \geq T \ \text{ then:} & d(t - T) \end{cases}$$

If the preconditions are not satisfied, we say that $c \circ_T d$ is undefined.

The composed curve $c \circ_T d$, when defined, is well-defined as a curve for any $T \in [0, \infty)$. The following associativity property is easily verified.

Lemma 4. *Let b, c and d be curves in A, let $T_1, T_2 \in [0, \infty)$ with $T_1 \leq T_2$, and let $b(T_1) = c^{init}$ and $c(T_2 - T_1) = d^{init}$. If $\delta(b(T_1))$ and $\delta(c(T_2 - T_1))$ are defined, then $(b \circ_{T_1} c) \circ_{T_2} d = b \circ_{T_1} (c \circ_{T_2 - T_1} d)$.*

When we compose curves c and d that are computations of P, i.e. when c and d satisfy the consistency conditions, then the explicit requirement in Definition 24 that $\delta(c(T))$ should be defined may well be omitted: it is implicit in the requirement that $c(T) = d^{init}$. In this case, the requirements on δ may be omitted in Lemma 4 as well.

If c and d are computations, they remain effective as individual components of $c \circ_T d$ (when defined). However, more can be said. The following Lemma is crucial.

Lemma 5. *Let c and d be curves that satisfy the consistency conditions (with respect to the knowledge theory for P), and let $T \in [0, \infty)$. If $c(T) = d^{init}$, then $c \circ_T d$ is well-defined and satisfies the consistency conditions again (with respect to the same knowledge theory for P).*

Proof. It is immediate that $c \circ_T d$ is well-defined. For proving that $c \circ_T d$ satisfies the consistency conditions, we check them one at a time. First of all, note that $(c \circ_T d)(0) = c(0)$ and thus, $\delta((c \circ_T d)(0)) = \delta(c(0)) \in E_0$.

To verify the second consistency condition, consider any $t_1, t_2 \in [0, \infty)$ with $t_1 \leq t_2$. If $t_2 \leq T$, then $(c \circ_T d)(t_1) = c(t_1)$ and $(c \circ_T d)(t_2) = c(t_2)$. If $T \leq t_1$, then $(c \circ_T d)(t_1) = d(t_1 - T)$ and $(c \circ_T d)(t_2) = d(t_2 - T)$. In both cases it follows that $\delta((c \circ_T d)(t_1)) \models^* \delta((c \circ_T d)(t_2))$, as this holds for c and d respectively.

If $t_1 \leq T \leq t_2$, then $(c \circ_T d)(t_1) = c(t_1)$ and $(c \circ_T d)(t_2) = d(t_2 - T)$. Thus $\delta((c \circ_T d)(t_1)) \models^* \delta(c(T)$ and $\delta(d^{init}) \models^* \delta((c \circ_T d)(t_2))$. As $c(T) = d^{init}$, we conclude that $\delta((c \circ_T d)(t_1)) \models^* \delta((c \circ_T d)(t_2))$, by transitivity of $\models^*$.

We conclude that $c \circ_T d$ satisfies the consistency conditions in A (w.r.t. the same knowledge theory as for c and d). □

By Lemma 5 it follows that, if c and d are computations of P and $c \circ_T d$ is well-defined, then $c \circ_T d$ qualifies to be a computation again. Thus, if $c \circ_T d \in C$, then 'jump starting' d by c can be done in a single computation of P.

Clearly curves, and computations in particular, may be composed in many ways. Note that $c \circ_T d$ is defined for every $T \geq 0$, as long as $c(T) = d^{init}$ and $\delta(c(T))$ is defined. When c is *loop-free*, i.e. visits every item of A at most once, then at most one curve can be obtained this way. Otherwise, one may obtain as many curves as c visits item d^{init}. (Note that the curves that are so obtained are not necessarily all different.)

We now define composition for sets of curves, as follows.

Definition 25. *Let $\mathcal{C}, \mathcal{D}$ be sets of curves. The set of all compositions of curves in $\mathcal{C}$ and $\mathcal{D}$ is $\mathcal{C} \bigtriangleup \mathcal{D} = \{c \circ_T d |\ c \circ_T d$ is defined$\}$.*

Lemma 6. *Let $\mathcal{B}, \mathcal{C}$ and $\mathcal{D}$ be arbitrary sets of curves. Then $(\mathcal{B} \bigtriangleup \mathcal{C}) \bigtriangleup \mathcal{D} = \mathcal{B} \bigtriangleup (\mathcal{C} \bigtriangleup \mathcal{D})$.*

Proof. This follows immediately from Lemma 4. □

With the basic facts in place, we turn to computational processes again. We note that, if the curves in $\mathcal{C}$ and $\mathcal{D}$ all satisfy the consistency conditions then, by Lemma 5, so do all curves in $\mathcal{C} \bigtriangleup \mathcal{D}$. We write $c \bigtriangleup d$ instead of $\{c\} \bigtriangleup \{d\}$

when convenient. By Lemma 6, we can write $c_1 \triangle c_2 \triangle \cdots \triangle c_k$ for any curves $c_1, \cdots, c_k$ without ambiguity.

Definition 26. *A computational process $P = \langle A, E, \delta, E_0, C\rangle$ is said to be* closed under composition *if for all $c, d \in C$: $c \triangle d \subseteq C$.*

When P is closed under composition, we will also call it *compositional.* If P is understood, we will refer to C as being closed under composition when we mean that P is.

Theorem 5. *Every computational process P has a well-defined compositional closure, i.e. there is a unique process $\overline{P} = \langle A, E, \delta, E_0, \overline{C}\rangle$ with smallest $\overline{C}$ such that $C \subseteq \overline{C}$ and for every $c, d \in \overline{C} : c \triangle d \subseteq \overline{C}$.*

Proof. Let L be the cpo of all sets of curves D with $C \subseteq D$. Define the operator $F : L \to L$ by $F(X) = X \cup (X \triangle C)$. We note that by Lemma 5, when all curves in X satisfy the consistency conditions w.r.t. E, then so do all curves in $F(X)$. The closure of C is the least $Y \in L$ such that $F(Y) = Y$, if it exists.

Observe that F is *chain-continuous*, i.e. for any chain $C_1 \subseteq C_2 \subseteq \cdots$ in L we have $F(C_1) \subseteq F(C_2) \subseteq \cdots$ and $F(\bigcup_{i\geq 1} C_i) = \bigcup_{i\geq 1} F(C_i)$. By the Tarski-Kantorovich fixed-point theorem, $\sup_i F^i(C)$ is a fixpoint of F and, in fact, it is the *least* fixpoint of F in L. By induction it easily follows that all curves in $\sup_i F^i(C)$ satisfy the consistency conditions w.r.t. E. Taking $\overline{C} = \sup_i F^i(C)$, the theorem follows. □

An alternate proof of Theorem 5 may be obtained by observing that the following extensional definition satisfies the requirements: $\overline{C} = \{c_1 \triangle \cdots \triangle c_k \mid k \geq 1 \text{ and } c_1, \cdots, c_k \in C\}$.

5.3 Composing processes

It is evident how one can extend the notion of composition as defined from sets of computations to computational processes. Composition makes it possible to 'observe' computations that are the result of applying different computational processes in a row.

We note that even the parallel composition ('direct product') of processes can be defined in the present framework, as follows.

Definition 27. *The* parallel composition *of processes $P_i = \langle A_i, E_i, \delta_i, E_0^i, C_i\rangle$ ($1 \leq i \leq n$) is the computational process $P = \langle A, E, \delta, E_0, C\rangle$ with*

- *$A = A_1 \times \cdots \times A_n$, the space of action item tuples with a product- or* sup *metric imposed,*
- *$E = E_1 \times \cdots \times E_n$, the space of knowledge item tuples with component-wise entailment, and*
- *$\delta = \delta_1 \times \cdots \times \delta_n$, $E_0 = E_0^1 \times \cdots \times E_0^n$ and $C = C_1 \times \cdots \times C_n$.*

Observe that for any curves $c_i \in C_i$ ($1 \leq i \leq n$), the product map $c = (c_1, \cdots, c_n) : [0, \infty) \to A$ is continuous in any of the defined metrics on A and thus is a curve again. Hence, the parallel composition of $P_1, \cdots, P_n$ is well-defined as a process.

Parallel composition of processes as defined, assumes that the processes involved all start at the same time and operate 'at the same speed'. As in general process theory, one could qualify the product operator by allowing for different modes of re-timing for the individual processes. We do not explore this further in this report.

5.4 Combining knowledge sets

Given computational processes $P = \langle A, E, \delta, E_0, C\rangle$ and $Q = \langle B, E, \mu, E_0, D\rangle$ with the same knowledge spaces and defined sets of initial knowledge, it is natural to ask how the knowledge sets they can generate might be combined. In this Section we consider a number of options for it.

Lemma 7. *Let $P = \langle A, E, \delta, E_0, C\rangle$ and $Q = \langle B, E, \mu, E_0, D\rangle$ be as given. Then there is computational process R such that $E_R = E_P \cup E_Q$.*

Proof. Re-define P (or Q) such that the action spaces of P and Q are disjoint. Now let R be the 'union' of P and Q. □

For a further property in this context we need the following concept, which fits in our earlier discussion of re-timing computations.

Notation 3 *For any $c \in C$, let c^T be the curve defined by $c^T(t) = c(0)$ for $0 \leq t \leq T$ and $c^T(t) = c(t - T)$ for $t \geq T$.*

Clearly, for any $T \geq 0$, curve c^T is a computation again, i.e. it satisfies the consistency conditions with respect to the knowledge theory of P.

Definition 28. *A computational process $P = \langle A, E, \delta, E_0, C\rangle$ is said to be* closed under delayed start*, if for every $c \in C$ and $T \in [0, \infty)$ one has that also $c^T \in C$.*

Lemma 8. *Let $P = \langle A, E, \delta, E_0, C\rangle$ and $Q = \langle B, E, \mu, E_0, D\rangle$ be as given, and assume that P and Q are closed under delayed start. Then there is a computational process R such that $E_R \backslash (E_0 \times E_0) = E_P \cap E_Q$.*

Proof. Consider the process $R = \langle A \times B, E \cup (E_0 \times E_0), \kappa, E_0 \cup (E_0 \times E_0), F\rangle$ with the following components:

– $E \cup (E_0 \times E_0)$ is the knowledge space with core set $E_0 \cup (E_0 \times E_0)$ and an inference relation $\models^r$ which obeys the following set of rules, where $\models$ is the inference relation for E:
  - first, the expected inferences for single and pairs of knowledge items:
    - $\Phi \models^r \Psi$ if $\Phi, \Psi \in E$ and $\Phi \models^* \Psi$,
    - $(\Psi, \Pi) \models^r (\Psi', \Pi')$ if $\Psi, \Psi', \Pi, \Pi' \in E$ and $\Psi \models^* \Psi'$ and $\Pi \models^* \Pi'$.
  - next, the inferences to 'produce' single knowledge items, if these items can be derived in both components of a pair simultaneously:
    - $(\Psi, \Pi) \models^r \Phi$ if $\Psi, \Pi, \Phi \in E$ and $\Psi \models^* \Phi$ and $\Pi \models^* \Phi$.

- and finally, the rules to guarantee that single knowledge items can be used in deriving pairs again:
  - $\Phi \models^r (\Psi, \Pi)$ if $\Phi, \Psi, \Pi \in E$ and $\Phi \models^* \Psi$ and $\Phi \models^* \Pi$.

The rules guarantee that $\models^r$ is reflexive and transitive, as required for an inference relation. (Notice that the first rule is superfluous, as it is implied by the other rules.)

– $\kappa : A \times B \to E$ is the semantic map such that:
  - $\kappa(a, b) = (\delta(a), \mu(b))$ if $\delta(a), \mu(b) \in E_0$ and $\delta(a) \neq \mu(b)$,
  - $\kappa(a, b) = \delta(a)$ if $\delta(a) = \mu(b)$,
  - $\kappa(a, b) =$ 'undefined' otherwise.

– F consists of the curves $\langle c, d\rangle$ defined by $\langle c, d\rangle(t) = (c(t), d(t))$, for $t \in [0, \infty)$.

The definitions imply that, if $c \in C$ and $d \in D$, then the curve $\langle c, d\rangle \in F$ is a computation again, satisfying the consistency condition from its starting point and onward. Thus, R is a computational process.

By definition of κ, $E_R \backslash (E_0 \times E_0) \subseteq E_P \cap E_Q$. To show equality, let $\Phi \in E_P \cap E_Q$. Then there are computations $c \in C$ and $d \in D$ such that $\Phi \in E_c$ and $\Phi \in E_d$. Let $t_1, t_2 \in [0, \infty)$ be such that $\delta(c(t_1)) = \mu(d(t_2)) = \Phi$. If $t_1 = t_2$, then $\kappa(\langle c, d\rangle(t_1)) = \Phi$ and we are done. Otherwise, assume w.l.o.g. that $t_1 < t_2$. By assumption on P we have that $c^{(t_2 - t_1)} \in C$. Now $\kappa(\langle c^{(t_2-t_1)}, d\rangle(t_2)) = \Phi$. Hence $\Phi \in E_R$. We conclude that $E_R \backslash (E_0 \times E_0) = E_P \cap E_Q$. □

Corollary 1. *Let $P = \langle A, E, \delta, E_0, C\rangle$ and $Q = \langle B, E, \mu, E_0, D\rangle$ be as given. Let $|E_0| = 1$, and assume that P and Q are closed under delayed start. Then there is a computational process R such that $E_R = E_P \cap E_Q$.*

Proof. This is immediate from the proof of Lemma 8. □

5.5 Generating knowledge by expansion

Given a computational process P, one might use the knowledge P can generate by its computations to expand the set E_0, and use this expanded set to provide P with the starting knowledge for new computations that were not 'enabled' before. By iterating this, one obtains a way of exploring a much larger subspace of E than P does on its own. This process is studied in more detail in [45]. It can be shown that the complete subspace of E that can be explored in this way is *well-defined* subset of E.

In the following sections we consider the deeper structural, compositional and computational properties of general computational processes.

6 Symbolic prototypes of computational processes

In the epistemic theory of computation, the observer plays a crucial role. When observing a process P, his semantic 'lens' δ and actual knowledge theory E determine which 'curves' of P qualify as computations and which do not. One may argue that, in practice, it works the other way around: a set C of 'intended computations' is given, and the question is which E and δ could 'explain' the given curves as computations.

In this section we wonder how one might understand a given set of curves, generated by a process P, without fixing any particular knowledge theory E ahead of time. What do we need, to explain the behaviour of the curves as computations, at least in theory? And, what do different interpretations of P as a computational process have in common?

Fix an action space A and a set of initial knowledge E_0. Suppose we allow the observer to use any possible semantic map δ, provided this map satisfies the following constraints, for two given, non-empty subsets A_0, A_1 of A, with $A_0 \subseteq A_1$:

- $\{a \mid \delta(a) \in E_0\} = A_0$, and
- $\{a \mid \delta(a)\ is\ defined\} = A_1$.

In addition, suppose that the curves c that we will encounter always satisfy the following base property:

- $c(0) \in A_0$,

corresponding to the, reasonable, fact that curves must always start in an action item with observable initial knowledge.

Now let C be a set of curves in A that we want to observe ('explain') as being computations, subject to the given constraints. We show how, for every C, one can always construct a process P_C that is 'symbolically computational' and explains C 'abstractly' as computations, based solely on the 'flow' of the curves. P_C will be called the *symbolic prototype* of C.

Next we argue that the process P_C exhibits the maximum possible knowledge theory that may enable an observer to achieve an explanation of the curves in C as being computations, in the following sense:

Let $P = \langle A, E, \delta, E_0, C\rangle$ be any computational process that 'explains' C as a set of computations, subject to the constraints on the semantic map δ and on the curves. Then P is a homomorphic image of P_C.

As a consequence, the symbolic prototype of a set of curves C is a key ingredient in answering the problem we deal with in this section. The question which E and δ could 'explain' a given set of curves C as 'computations' is completely reduced to the problem of finding a suitable homomorphism ϕ that maps its symbolic prototype onto a concrete computational process.

We first give and explain the definition of symbolic prototypes. Next we prove the key results for it.

6.1 Symbolic prototyping

Let A and E_0 be fixed. Also, assume that the semantic maps δ and curves c we allow, all respect the constraints given above. Let P be a process and C the set of observable curves it can generate. When we observe C, what determines that its curves are computations? This seems to be the essence of the question what it is that makes P a computational process.

The crucial question is to what extent this issue depends on the curves in C and to what extent on a specific knowledge theory used by the observer in interpreting their effect. In this section we show that the two aspects are heavily intertwined, i.e. the curves in C themselves determine the very knowledge theory we need in arguing for their computationality.

Symbolic knowledge space In order to determine their computationality, we have to capture the 'dynamics' of the curves in C, subject to the constraints we assumed. The idea is that there is implicit information hidden in the curves that we want to extract.

A key notion in modeling computations by curves is the gradually advancing journey through consecutive items in A, by some mechanism that causes it. The gradually advancing knowledge build-up is driven by the same dynamics. We capture this as follows.

Let C be an arbitrary, non-empty set of curves in A. As stipulated above, we assume that all curves of C are properly *grounded*: for all $c \in C$, we assume that $c(0) \in A_0$, i.e. $\delta(c(0)) \in E_0$. We do not assume beforehand that the curves satisfy any further consistency conditions, with respect to any theory.

Define the knowledge space $K = K_C$ as follows. We give the definition in parts: a core set K_0^C, the space K_C, and a relation $\models_{K_C}$ on K_C, as follows:

$$K_0^C = \{[a, c, t] \mid a \in A, c \in C, t = 0, c(t) = a, \text{ and } a \in A_0\},$$

$$K_C = K_0^C \cup \\ \cup\{[a, c, t] \mid a \in A, c \in C, c(0) \in A_0, t > 0, c(t) = a, \text{ and } a \in A_1\},$$

and

$$[a, c, t] \models_{K_C} [a', c', t'] \text{ if and only if } c = c' \text{ and } t \leq t'.$$

where in the latter definition we assume that the tuples $[a, c, t]$ and $[a', c', t']$ both belong to K_C. Notice, in the definition of K_0^C, that we assumed that $c(0) \in A_0$ for all $c \in C$. Thus K_0^C, and hence K_C, is always *non-empty.*

One readily observes that $\models_{K_C}$ is an inference relation on the items of K_C, and that K_C is closed under inferencing. Hence, K_C is valid as a *knowledge space.* More specifically, the following lemma may be observed.

Lemma 9. *K_C is a theory-like knowledge space with core K_0^C.*

Proof. We argued that K_C is closed under $\models_{K_C}$, which is the requirement for being a knowledge space. However, taking K_0^C as the core set, one easily shows that K_C is the closure of K_0^C under $\models_{K_C}$, i.e. within the space $\{[a, c, t] \mid a \in A, c \in C, t \in [0, \infty), c(t) = a, \text{ and } a \in A_1\}$. Hence, K_C is theory-like. □

One may notice that K_C consists of knowledge items that precisely represent the generic information on all action items that may potentially contain knowledge – according to any δ – if A would be explored with curves from C, without being specific on *what* the knowledge extracted by δ might be. K_C only depends on E_0, the subsets A_0 and A_1 of A, and the curves in C, and not on any specific knowledge theory.

Taking the constraints as given, knowledge theory K_C is completely determined by the behaviour of the curves in C. It is the abstract rendering of what the curves can potentially explore. But, does this enable an observer to see the curves of C as computations?

Symbolic process We now proceed to show that K_C can be utilized as the knowledge space of a *symbolic* process $P_C = \langle A_C, K_C, \delta_C, K_0^C, M_C\rangle$. P_C is called symbolic, because at this stage we do not assume any pre-defined mechanism that might 'generate' the curves of C.

In order to specify P_C we have to define its action space, its semantic map, and its potential set of computations. First we define its action space A_C, as follows:

$$A_C = \{\langle a, c, t\rangle \mid a \in A, c \in C, \text{ and } t \in [0, \infty)\}.$$

A_C represents the realm that can be maximally explored by means of the curves of C, in a 'full information' sense. Note that $A_C = A \times C \times [0, \infty)$ is a metric space (with the discrete metric for C and the common product metric on the space) and thus A_C is a valid action space.

Next, we define the appropriate semantic map δ_C that 'reads' the items of A_C and interprets them in the knowledge theory K_C:

$$\delta_C(\langle a, c, t\rangle) = \begin{cases} \text{if } [a, c, t] \in K_C \text{ then:} & [a, c, t] \\ \\ \text{otherwise:} & \textit{undefined} \end{cases}$$

Finally, we define the set of 'computations' of P_C. As A_C is a metric space, it makes sense to try and interpret the curves of C as curves in A_C. To this end, for any $c \in C$, let $c_C : [0, \infty) \to A_C$ be the map defined by:

$$c_C(t) = \langle c(t), c, t\rangle, \text{ for any } t \in [0, \infty).$$

Indeed, for any curve $c \in C$, the mapping c_C is continuous as well and thus a curve in A_C. Hence, we can define a perfect 'mirror set' of C in P_C, as follows.

$$M_C = \{c_C \mid c \in C \text{ and } c(0) \in A_0\}$$

Recall that we assumed that for all $c \in C$, $c(0) \in A_0$ at the outset, but we include it again for clarity. It is clear that M_C is fully 'isomorphic' to C, differing only in the full representation of its actions.

Definition 29. *Process $P_C = \langle A_C, K_C, \delta_C, K_0^C, M_C\rangle$ is called the* symbolic prototype *of the set of curves C.*

Considering the process $P_C = \langle A_C, K_C, \delta_C, K_0^C, M_C\rangle$ as it is now defined, we see that all components of P_C depend solely on the set of curves C (and the initial constraints). The crucial observation now is that P_C can be shown to be 'symbolically computational', i.e. the curves in M_C satisfy the consistency conditions even though we have left it fully open how the curves are generated.

Theorem 6. *The symbolic prototype $P_C = \langle A_C, K_C, \delta_C, K_0^C, M_C\rangle$ of any non-empty set of curves C in A is symbolically computational.*

Proof. We verify that the curves in M_C satisfy the required consistency conditions. The first condition requires that all curves have their starting knowledge in K_0^C, but this follows by assumption. After all, requiring that for all $c_C \in M_C$ we have $\delta_C(c_C(0)) \in K_0^C$ is equivalent to requiring that for all $c \in C$ we have $c(0) \in A_0$, and this was assumed for all curves in C.

For the remaining consistency condition, consider any curve $c_C \in M_C$ and any times $t_1, t_2 \in [0, \infty)$ with $t_1 \leq t_2$. Without loss of generality, let $t_1 < t_2$. Suppose that both $\delta_C(c_C(t_1))$ and $\delta_C(c_C(t_2))$ are defined, This means that both $[c_C(t_1), c, t_1]$ and $[c_C(t_2), c, t_2]$ are defined in K_C, and that consequently $[c_C(t_1), c, t_1] \models_{K_C} [c_C(t_2), c, t_2]$. This proves that the required consistency of knowledge build-up along every curve in M_C is satisfied.

We conclude that P_C is computational, symbolically. □

The proof of Theorem 6 shows that the 'dynamics' of the curves in M_C is sufficient for obtaining valid computations, i.e. assuming only that the curves 'start right' and 'unfold as curves' after that, but nothing more. Short of declaring *any* set of curves to be computational, this shows the importance of an interpretation of P_C as a concrete process. Thus, the only piece that is still missing, is some kind of 'embodiment' of the process, to perform the computations.

Lemma 10. *Let $P_C = \langle A_C, K_C, \delta_C, K_0^C, M_C\rangle$ be the symbolic prototype of some non-empty set of curves C in A. Viewed as a symbolically computational process, P_C is both operationally and observationally deterministic.*

Proof. Consider any two curves c_C and d_C of M_C. If, for some $t_1, t_2 \in [0, \infty)$, we have $c_C(t_1) = d_C(t_2)$, then $\langle c(t_1), c, t_1\rangle = \langle d(t_2), d, t_2\rangle$ and hence $c = d$ and $t_1 = t_2$. Thus, in particular, $c^{t_1} = d^{t_2}$. It follows that P_C is operationally deterministic.

Following a very similar line of argumentation, let $\delta_C(c_C(t_1)) = [c(t_1), c, t_1]$ and $\delta_C(d_C(t_2)) = [d(t), d, t_2]$ be defined and let $\delta_C(c_C(t_1)) = \delta_C(d_C(t_2))$. Then necessarily $c = d$ and $t_1 = t_2$, hence, $c^{t_1} = d^{t_2}$ again. Thus P_C is observationally deterministic. □

As a consequence, symbolic prototypes always are repeatable processes when they are considered as being (symbolically) computational. See Subsection 7.1 for a further explanation of repeatable processes and their properties.

6.2 From symbolic prototype to computational process

Let us consider the lead question again. Thus, let C be a non-empty set of curves in A. The question we consider is, whether and how an observer might understand, or interpret, the curves in C as the possible product of a computational process, using a suitable knowledge theory and an adequate 'looking glass' δ. Have we come close to an answer?

Clearly, for C to be understood as a set of computations, it is necessary that the set of initial knowledge E_0 and the semantic map δ of the observer satisfy the condition that $\delta(c(0)) \in E_0$, for all $c \in C$. What can we say if this condition is satisfied? Is it sufficient?

Notice that, by choosing the ranges A_0 and A_1 in any way we want, subject to the conditions that $A_0 \supseteq \{c(0) \mid c \in C\}$ and $A_1 \supseteq A_0$, a 'symbolic prototype' process P_C is obtained with the following properties: it explains M_C, a full information rendering of C, and is, at least, symbolically computational, by Theorem 6. Thus, the necessary condition above is close to being sufficient, at least symbolically. However, more can be said.

Suppose that $P = \langle A, E, \delta, E_0, C\rangle$ is a computational process that indeed 'explains' C as being computational. Let $A_0 = \{a \mid \delta(a) \in E_0\}$ and $A_1 = \{a \mid \delta(a) \textit{ is defined}\}$, and let $P_C = \langle A_C, K_C, \delta_C, K_0^C, M_C\rangle$ be the symbolic prototype as it is determined by A_0 and A_1 (and E_0).

The following result shows how process P is intimately linked to process P_C, for the choices of E_0, A_0 and A_1 that apparently work to explain C as being computational. Recall that E_P is the set of all knowledge items generated by the computations of a process P.

Theorem 7. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process, and let $P_C = \langle A_C, K_C, \delta_C, K_0^C, M_C\rangle$ be the symbolic prototype as determined by $A_0 = \{a \mid \delta(a) \in E_0\}$ and $A_1 = \{a \mid \delta(a) \textit{ is defined}\}$ (and E_0). Then P is a homomorphic image of P_C, i.e. there are an epi-transmorphism $f : P_C \to P$, and an epistemorphism $\tau : K_C \to E$ such that $\tau(E_{P_C}) = E_P$.*

Proof. Let P_C and P be as given. In order to show that P is a homomorphic image of P_C, we need to construct a mapping $h = (f, \tau)$ that satisfies the terms of Definition 19 with f epi. Choose the constituent maps as follows.

Let $f : A_C \to A$ be the map defined by $f(\langle a, c, t\rangle) = a$, for all $\langle a, c, t\rangle \in A_C$. As a map between metric spaces, f is seen to be continuous. To see that f is a transmorphism, let c_C be an arbitrary symbolic computation of P_C, i.e. with $c \in C$. Then $f \circ c_C : [0, \infty) \to A$ is the mapping defined by

$$f \circ c_C(t) = f(c_C(t)) = f(\langle c(t), c, t\rangle) = c(t),$$

and thus $f \circ c_C \equiv c$ and hence $f \circ c_C \in C$. It follows that f is a transmorphism. Also, f is clearly epi.

Next, define $\tau : K_C \to E$ by $\tau([a, c, t]) = \delta(a)$, for any $[a, c, t] \in K_C$ (with $a \in A$, $c \in C$ and $t \in [0, \infty)$). By the given specification of K_C, τ is well-defined, i.e. its values are all 'defined' on K_C.

To verify that τ is an epistemorphism, first observe that $\tau(K_0) \subseteq E_0$. This is immediate from the definition of K_0. Next, consider any two items $[a, c, t]$ and $[a', c', t']$ with $[a, c, t] \models_K [a', c', t']$. Then $c = c'$ and $t \le t'$ and, consequently, a precedes a' in time along the curve c. By the consistency conditions for P, it follows that $\delta(a) \models_E \delta(a')$, in other words that $\tau([a, c, t]) \models_E \tau([a', c', t'])$. This proves that τ is an epistemorphism.

Finally, we note that the commutativity property holds: $\tau \circ \delta_C = \delta \circ f$. This follows because for any $\langle a, c, t\rangle \in A_C$, if $a \in A_1$ then $[a, c, t] \in K_C$ and thus

$$\tau \circ \delta_C(\langle a, c, t\rangle) = \tau([a, c, t]) = \delta(a) = \delta \circ f(\langle a, c, t\rangle).$$

If $a \notin A_1$, then both $\tau \circ \delta_C(\langle a, c, t\rangle)$ and $\delta \circ f(\langle a, c, t\rangle)$ are *undefined.*

The completes the proof that P is a homomorphic image of P_C under the mapping $h = (f, \tau)$. By Theorem 2 we have that $\tau(E_{P_C}) = E_P$. □

For a direct proof that $\tau(E_{P_C}) = E_P$, notice that $\tau(E_{P_C}) = \tau(\bigcup_{c \in C}\{[c(t), c, t] \mid t \in [0, \infty) \text{ and } [c(t), c, t] \in K_C\}) = \bigcup_{c \in C}\{\delta(c(t)) \mid t \in [0, \infty) \text{ and } \delta(c(t)) \text{ defined}\} = E_P$.

Considering the question again, when and how a given set of curves may be regarded as being computational, we see that the symbolic prototypes of C are crucial:

> *every concrete computational process P that 'explains' C as being computational must be the homomorphic image of a symbolic prototype of C. All processes P that do, can be obtained in this way.*

Observe that the definition of a prototype P_C for the set C relies purely on the support of the δ of the observer, and on the elements of C as being curves. No further facts of δ play a role. Indeed, prototypes do not even rely on any concrete knowledge theory, other than what is implied by the 'flow' of the curves. Note that one may always take $E_0 = \{c(0) \mid c \in C\}$.

The task of finding a suitable prototype P_C for C does not necessarily make it any easier to determine a concrete computational process P for C. However, in a nutshell, Theorem 7 tells us that symbolic prototypes represent the essence of the process structure that is required.

6.3 From computational process to symbolic prototype

We now change our perspective, and consider sets of curves C which we know to be computational. Let $P = \langle A, E, \delta, E_0, C\rangle$ be any computational process that generates C. What is the essence of P's operation? And, what is the forcing power behind the consistency conditions? These are only some of the deep questions one may ask.

The symbolic analysis above suggests that the computational behavior of P relies rather more on the 'dynamics' of the curves and the scope of the 'lens' δ than on any specific details of the observer's knowledge theory. Concretely, P's qualities seem to depend solely on E_0, the support of δ, and the curves in C satisfying the consistency conditions. No detailed properties of E seem needed, i.e. other than those implied by the definition of δ on the items of A. This can be made concrete as follows.

Let $P = \langle A, E, \delta, E_0, C\rangle$ be as given. Define $A_0 = \{a \in A \mid \delta(a) \in E_0\}$ and $A_1 = \{a \in A \mid \delta(a) \text{ } is \text{ } defined\}$.

Definition 30. *The symbolic prototype $P_C = \langle A_C, K_C, \delta_C, K_0, M_C\rangle$ of C determined by the sets A_0 and A_1 implied by P, is called the* symbolic prototype *of P.*

By Theorem 6 above, the symbolic prototype of P is 'symbolically computational'. It may be viewed as abstracting the dynamics of P's computations, in a way that is independent of the concrete knowledge build-up (although this is clearly what the process is supposed to do for us).

Two main theorems may now be observed which describe the distinguishing features of symbolic prototypes. The first result characterizes precisely what is abstracted in a prototype, i.e. when two different processes have the same symbolic prototype.

Theorem 8. *Let $P = \langle A, E, \delta, E_0, C \rangle$ and $Q = \langle A, F, \mu, F_0, C \rangle$ be computational processes. Assume that for all $a \in A$, $\delta(a)$ is defined if and only if $\mu(a)$ is defined, and $\delta(a) \in E_0$ if and only if $\mu(a) \in F_0$. Then P and Q have the same symbolic prototype.*

Proof. This follows by equating the definitions of P_C and Q_C, respectively. □

The next result expresses the (mathematical) relationship between a computational process P and its symbolic prototype P_C. It is a reformulation of the result in Theorem 7.

Theorem 9. *Every computational process is a homomorphic image of its symbolic prototype.*

Proof. See the proof of Theorem 7. □

Theorem 9 shows that symbolic prototypes are a valid abstraction of the dynamical behaviour of a computational process. It is conceivable that the prototype of a process P is computational again, e.g. in case the computations in M_C could be generated by a modification of the mechanism that generates the computations in C. This could be imagined if the set C is countable.

Viewing P_C as a symbolically computational process, Theorem 9 leads to several useful further observations. The first observation completes the view of P_C as an abstraction of P.

Corollary 2. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process, and let $P_C = \langle A_C, K_C, \delta_C, K_0, M_C \rangle$ be its symbolic prototype. Then there is an epistemorphism $\tau : K_C \to E$ such that P and $\tau \circ P_C$ generate the same knowledge, i.e. $\tau(E_{P_C}) = E_P$.*

Proof. It is implicit in Theorem 7 that there is an epistemorphism $\tau : K_C \to E$ such that $\tau(E_{P_C}) = E_P$. By Lemma 1, the set $\tau(E_{P_C})$ is precisely the knowledge set generated (symbolically) by the symbolic process $\tau \circ P_C = \langle A_C, K, \tau \circ \delta_C, K_0, M_C \rangle$. □

Finally, we observe the following interesting mathematical side-result.

Corollary 3. *Let P be an arbitrary computational process. Then P is the homomorphic image of a symbolically computational process that is both operationally and observationally deterministic.*

Proof. By Theorem 9, P is a homomorphic image of its symbolic prototype P_C. By Lemma 10, P_C indeed has the desired properties. □

7 Observing computations

We now proceed with the study of computational processes, in the present general setting. Let A be an arbitrary action space, with metric d, and let E be a knowledge space. Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. In the epistemic philosophy, observers play a crucial role, but what can they really observe of a computation other than the knowledge that is generated? We first consider repeatable processes. Next, we define operational and observational discreteness for computational processes.

7.1 Repeatable processes

Computational processes are defined generally, without recourse to any description or, indeed, algorithm. This allows the general interpretations we aimed for. Yet, in practice one will want a process to satisfy an important property that tends to hold especially for 'programmed' systems: *repeatability.*

Definition 31. *A computational process $P = \langle A, E, \delta, E_0, C \rangle$ is said to be* repeatable *if for every $e \in E_0$, there is at most one $c \in C$ such that $\delta(c^{init}) = e$.*

In repeatable processes, same initial knowledge leads to same computations. Hence, repeatable processes 'can be run again' and are thus ideal if one wants a process to be used for experimentation. Theoretically, repeatable computations can be 'indexed' by their c^{init}.

Lemma 11. *For every computational process $P = \langle A, E, \delta, E_0, C \rangle$ there is a repeatable computational process $Q = \langle A', E', \delta', E_0', C' \rangle$ and an epistemorphism $\tau : E' \to E$ such that $E = \tau(E')$ and for every $c \in C$ there is a $c' \in C'$ such that $E_c = \tau(E_c')$ (and vice versa).*

Proof. Let P be as given. Consider the process $Q = \langle A \times C, E \times C, \delta', E_0 \times C, C' \rangle$ such that for every $a \in A$ and $c \in C$ we have that $\delta'([a, c]) = [\delta(a), c]$ and $C' = \{c' \mid c \in C\}$, where for every $c \in C$, the map $c' : [0, \infty) \to A \times C$ is defined by $c'(t) = [c(t), c]$.

Let d_A be the metric on A and d_C the discrete metric on C. Then $A \times C$ is a metric space any common product metric like the *2-product metric.* One easily verifies that every map $c' \in C'$ is a curve in $A \times C$ under this metric. Furthermore, we note that $E \times C$ is a valid knowledge space, with inference relation $\models'$ defined such that $[e_1, c_1] \models' [e_2, c_2]$ if and only if $e_1 \models e_2$ (in E) and $c_1 = c_2$. We conclude that Q is a computational process.

We first observe that Q is a repeatable process. This follows because, if for two computations $c', d' \in C'$ one has that $c'(0) = d'(0)$, then necessarily $c = d$. Next, consider the mapping $\tau : E' \to E$ defined by $\tau([e, c]) = e$. Clearly τ is an epistemorphism. Also, consider any computation $c \in C$. Then for the computation $c' \in C$ defined by $c'(t) = [c(t), c]$ one has clearly has $E_c = \tau(E_c')$ (and vice versa). $\square$

In the proof we assumed that the computations of C can be used to index themselves. If they are determined by 'programs', one could use those in stead.

Lemma 12. *Observationally deterministic processes are repeatable.*

Proof. This can be seen directly from Definition 21. □

Note that, if Definition 21 had not been chosen the way it was, we would not have been able to conclude now that observationally deterministic processes are repeatable. We refer to Subsection 3.5 for a general characterization of the computations generated by observationally deterministic processes.

7.2 Discrete processes

Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. Recall from Definition 13 that we identified the set $\{t \mid \delta(c(t))\ \textit{is defined}\}$ as the domain of the *observable trace* of a computation c.

One may argue that an observer can only observe a computation at discrete moments, and not continuously. If there is nothing more to be observed, this would tell the observer all he can possibly distill from the computation. Then, and only then, he can limit himself to the observed moments and to what he observes at these times (for this computation). This leads to the following definitions.

Definition 32. *A process $P = \langle A, E, \delta, E_0, C\rangle$ is called* operationally discrete *if for every computation $c \in C$, the set $\{t \mid \delta(c(t))$ is defined$\}$ has no accumulation points (in $\mathbb{R}$).*

Definition 33. *A process $P = \langle A, E, \delta, E_0, C\rangle$ is called* observationally discrete *if for every computation $c \in C$, the set $\{c(t) \mid \delta(c(t))$ is defined$\}$ has no accumulation points (in A).*

We note here that a subset of a topological space has no accumulation points if and only if it is *closed* ('contains all its accumulation points' - in this case because the subset has none) and *discrete* ('has only isolated points'). Note also that $\{t \mid \delta(c(t))\ \textit{is defined}\}$ is never empty, as 0 always belongs to the set (due to the consistency conditions). Consequently, the set $\{c(t) \mid \delta(c(t))\ \textit{is defined}\}$ is never empty either.

Lemma 13. *Let P be operationally discrete. Then for every computation c of P, the set $\{t \mid \delta(c(t))$ is defined$\}$ is either finite, or countably infinite and unbounded.*

Proof. Let P be operationally discrete, and let c be a computation of P. Then $\{t \mid \delta(c(t))\ \textit{is defined}\}$ must be a discrete set. By general topology it follows that it is either finite or countably infinite. In the latter case, the set must be unbounded. For, if it were bounded, general topology tells us that it would have an accumulation point in $\mathbb{R}$. This would contradict the fact that the set has none. □

A relevant consequence for operationally discrete processes that we will exploit later, is the following property. It states that for every time $T \in [0, \infty)$, there is a first later moment t in time for which $\delta(c(t))$ is defined, if there is moment for which it is defined at all.

Lemma 14. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally discrete. Let $T \in [0, \infty)$ be such that* $\{t > T \mid \delta(c(t)) \text{ is defined}\}$ *is non-empty. Then the set* $\{t > T \mid \delta(c(t)) \text{ is defined}\}$ *has a minimum element.*

Proof. As P is operationally discrete, the set $\{t > T \mid \delta(c(t)) \textit{ is defined}\}$ cannot have an accumulation point. This means that its infimum must be an element of the set, i.e. be its minimum. □

Without loss of generality, the domains $\{t \mid \delta(c(t)) \textit{ is defined}\}$ may always be assumed to be countably infinite (and unbounded), by virtue of the following observation.

Lemma 15. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally discrete. Then there exists an operationally discrete computational process $P' = \langle A', E, \delta', E_0, C'\rangle$ such that $P \equiv P'$ and for every computation $c' \in C'$,* $\{t \mid \delta(c'(t)) \text{ is defined}\}$ *is countably infinite and unbounded.*

Proof. Let P be as given. Consider the process $P' = \langle A \times C, E, \delta', E_0, C'\rangle$, with δ' such that for every $a \in A$ and $c \in C$ we have $\delta'([a, c]) = \delta(a)$, and $C' = \{c' \mid c \in C\}$, where for every $c \in C$, the map $c' : [0, \infty) \to A \times C$ is defined by $c'(t) = [c(t), c]$.

Let d_A be the metric on A and d_C the discrete metric on C. Then $A \times C$ is a metric space, e.g. under the common 2-product metric. As in the proof of Lemma 11, one verifies that every map $c' \in C'$ is a curve in $A \times C$ under this metric. Hence, P' is a computational process, and every computation $c' \in C'$ is a 'replica' of $c \in C$ within its own slice $A \times \{c\}$ of $A \times C$. In particular, P' is again operationally discrete.

Modify δ' into a new semantic map δ'' as follows. If $c \in C$ is such that $\{t \mid \delta(c(t)) \textit{ is defined}\}$ is infinite, then we leave all values $\delta''([a, c]) = \delta'([a, c]) = \delta(a)$ unchanged. If $c \in C$ is such that $\{t \mid \delta(c(t)) \textit{ is defined}\}$ is finite, then we leave $\delta''([a, c]) = \delta'([a, c]) = \delta(a)$ unchanged whenever $\delta(a)$ is defined, but redefine $\delta''([a, c])$ if it is not, as follows.

As $\{t \mid \delta(c(t)) \textit{ is defined}\}$ is finite (and non-empty), it must have a largest element, say T (by a similar argument as in Lemma 14). Consequently, the items $c(t)$ have undefined δ-value for all $t > T$. Thus, we can safely set $\delta''([a, c])$ to be $\delta(c(T))$ for any or all of these items. In particular, define $\delta''([c(T + k), c]) = \delta''([c(T), c]) = \delta(c(T))$ for all $k \in \mathbb{N}$. For all items whose δ''-value does not get modified this way, we leave $\delta''([a, c]) = \delta'([a, c]) = \delta(a)$ undefined.

One easily verifies that the modification of δ' into δ'' leaves the consistency conditions along the curves $c' \in C'$ unaffected. Thus $P'' = \langle A \times C, E, \delta'', E_0, C'\rangle$ is again a computational process. Clearly, P'' is operationally discrete again also, and for every $c' \in C'$, the set $\{t \mid \delta(c'(t)) \textit{ is defined}\}$ is now always countably infinite and unbounded. The equivalence of P and P'' is immediate, as $E_c = E_{c'}$, for every $c \in C$. □

Although the action space $A \times C$ used in the proof is generally large, it may be noted that $|A \times C| = \max(|A|, |C|)$ as soon as one the cardinalities $|A|$ or $|C|$ is infinite. Under mild conditions on C we can prove Lemma 15 by just modifying δ, i.e. *without* needing a different action space.

Example 8. Let $P = \langle A, E, \delta, E_0, C \rangle$ be an operationally discrete computational process with the property that for every $c \in C$ and time T, there is a time $t > T$ such that $c(t) \notin \{c_1(t) \mid c_1 \in C \textit{ and } c_1 \neq c\}$. Assume that C is countable. Then there is an operationally discrete computational process $P' = \langle A, E, \delta', E_0, C' \rangle$ equivalent to P that already satisfies the desired property of Lemma 15. The (inductive) proof proceeds along similar lines as above.

If P is operationally discrete, then the set $\{c(t) \mid \delta(c(t)) \textit{ is defined}\}$ is closed and at most countably infinite too. This follows because it is the image of a closed and at most countable infinite set, namely $\{t \mid \delta(c(t)) \textit{ is defined}\}$, under a continuous map, namely c. An important fact is the following.

Theorem 10. *Let P be observationally discrete, i.e. for every $c \in C$ the set $\{c(t) \mid \delta(c(t))$ is defined$\}$ has no accumulation points (in A). Then P is operationally discrete.*

Proof. Let c be an arbitrary computation of P. Suppose $\{t \mid \delta(c(t)) \textit{ is defined}\}$ had an accumulation point in $\mathbb{R}$, say T. Then, by continuity of c, $c(T)$ would be an accumulation point of $\{c(t) \mid \delta(c(t)) \textit{ is defined}\}$. This would contradict the assumption that P is observationally discrete. □

The properties of observationally discrete processes give us the proper basis for developing the notion of discrete computation.

8 Continuous vs. discrete computation

We defined computation as a phenomenon that unfolds in continuous time. It is the mathematical rendition of Turing's claim that *'all machinery can be regarded as continuous'* [42]. However, in many classical conceptions, computation is considered as a phenomenon that is unfolding in *discrete* time. Even Turing qualified his claim by noting that *'when it is possible to regard [machinery] as discrete it is usually best to do so'* [42].

The distinction between the 'continuous' and the 'discrete' view is often seen as a dichotomy in the field of computation and as an obstacle for developing a general theory. But, do we really have different notions of *computation* here? In this section we will resolve the dichotomy and reconcile the seemingly different notions that are involved.

The first question to be addressed is, what 'discrete computation' is in the first place. Staying with the epistemic philosophy adhered to in this report, discrete computations must be 'produced' by a computational process and generate 'knowledge' in some way. However, we can't just impose a discrete time-scale (or clock) on a process. A computation should only be called discrete if there is an observational criterion for doing so.

In this section we develop the epistemic philosophy of computation for the discrete case. We first define *discrete processing* as the acting mechanism underlying discrete computation. We will argue that discrete processing systems can be seen as a generalization of *tolerance automata* as introduced by Arbib in the 1960's [3]. We then give a plausible criterion for when a discrete processing system may be called 'computational'. In the next section we prove the fundamental facts that relate computational discrete processing systems and (observationally discrete) computational processes.

8.1 Discrete processing and computation

We begin by considering what an observer 'sees', when he observes something that we might call a *discrete computation.* How might the phenomenon be generated, and how should it be defined so it fits the very large number of instances in which the notion of discrete computation occurs? It is a question very much like the one we considered for computation in general.

Following our earlier approach, the observed phenomenon must be produced by some underlying system of a similar nature as a computational process, except that this system manifests itself only at times chosen from some unbounded discrete subset $Obs \subset \mathbb{R}$. The processing takes place in an action space, and action items can be read out *only* at times t with $t \in Obs$. Whatever happens in between the chosen times is assumed to be 'hidden' or 'unknown' to the observer. In particular, we assume that *no observable knowledge is produced during these in-between intervals.*

The underlying processing system can be seen as generating *discrete maps* that unfold in an action space U. We distinguish between general, operational discrete maps and the stronger observational variant.

Definition 34. *A discrete map is any mapping $v : Obs(v) \to U$, where the time domain $Obs(v) \equiv \{t_0^v, t_1^v, \cdots\}$ is some countable, unbounded subset of $\mathbb{R}$ with $t_0^v = 0$ and $t_0^v < t_1^v < \cdots$.*

Definition 35. *A discrete map $v : Obs(v) \to U$ is said to be* observationally discrete *if, in addition to the properties in Definition 34, we also have that $\{v(t_0^v), v(t_1^v), \cdots\} \subseteq U$ has no accumulation point (in U).*

Discrete maps are defined in a completely general way, but presumably there is some additional information around that explains how (and why) the discrete maps can be generated by the underlying system. One may think of $Obs(v)$ as a clock that ticks precisely at the times that the system produces knowledgeable information. We only consider unbounded time domains (cf. Lemma 15).

Discrete maps will play the same role in discrete computation as curves do in the continuous case.

Notation 4 *For any discrete map v, we write $v_i = v(t_i^v)$.*

Notation 5 *For any discrete map v, we write $v^{init} = v_0$.*

Returning to the observer, we may assume then that he observes a sequence of *action items* as it is being produced in a step-wise fashion following the course of some discrete map. The observed information consists, in principle, of the *knowledge items* that are distilled from the action items passed on the way, using a suitable *semantic map.*

This leads to the following analogue of Definition 5 for the case of discrete computation.

Definition 36. *A* discrete processing system *is any 5-tuple $Q = \langle U, E, \delta, E_0, V\rangle$ where U is an action space, E is a knowledge space, $\delta : U \to E$ is a semantic map, $E_0 \subseteq E$ is a non-empty set of initial knowledge, and V is a collection of discrete maps $v : Obs(v) \to U$ such that the following* consistency conditions *are satisfied:*

- *for every $v \in V$, $\delta(v^{init}) \in E_0$, and*
- *for all time indices i, j with $i \leq j$: if $\delta(v_i)$ and $\delta(v_j)$ are both defined, then $\delta(v_i) \models^\star \delta(v_j)$ in E.*

Again we assume that the observer has some idea of how (and why) a discrete system is doing what it does. The given definition is intentionally broad, and articulates only the bare essence of the generative systems we want to include and no more. We normally assume that in discrete processing systems, δ is defined on all action items that are possibly visited by a discrete map.

Whereas the definition is very broad, we may sometimes have to assume that for all discrete maps v, the times t_i^v for $i = 0, 1, \cdots$ are either known to the observer or signalled to him 'at runtime' through some observable property of the system (like the ticking of a clock). In this way the actor-spectator paradigm makes sense also in the discrete case.

In observing discrete maps v, it may well be possible that the observer sees some kind of convergence in the sequence $v_0, v_1, \cdots$ (with or without an actual limit). This is not excluded in general but, when it is, the generated map will be observationally discrete. It may be argued that discrete processing systems that are truly discrete, only display observationally discrete behaviours (maps), from an observer's perspective.

Definition 37. *A discrete processing system $Q = \langle U, E, \delta, E_0, V\rangle$ is said to be* observationally discrete *if all maps in V are observationally discrete.*

By imposing further constraints, one may limit discrete processing systems to those systems which have a specific property. In particular, in the next section we will add some constraints that intend to express when a discrete processing system may be called *computational.*

Tolerance systems As in the continuous case, it may be reasonably to impose a proximity condition on the successive steps in a discrete map. This corresponds to the credible property of discrete processing systems that in between any two consecutive observable time moments of a discrete map, only a small number of changes can be effectuated in the recently visited action item.

One can only speak of proximities in space U if there is a way to define what it means for action items to be 'close'. A very similar question can be recognized in Arbib's work in the 1960's [2, 3] when he considered how one might adapt the continuous behaviour of control systems to a discrete setting. He proposed to use the notion of *tolerances* for it, originally due to Zeeman [52] in his work on proximities in visual perception in the brain.

Definition 38. *A* tolerance *on U is any binary relation ξ on U that is reflexive and symmetric.*

If ξ is a tolerance on U, then $[u_1, u_2] \in \xi$ is meant to indicate that u_1 and u_2 are 'proximate' or 'near', i.e. it is seen as a 'tolerable' jump for a discrete computation to move from u_1 to u_2 'in one step' in the action space. If we apply this to discrete maps and discrete processing systems, we obtain the following plausible notions.

Definition 39. *A discrete map $v : Obs(v) \to U$ is called ξ-continuous if for all time indices i we have $[v_i, v_{i+1}] \in \xi$.*

Definition 40. *A discrete processing system $Q = \langle U, E, \delta, E_0, V \rangle$ is called a* tolerance system *if there is a tolerance ξ on U such that every $v \in V$ is ξ-continuous.*

Tolerance systems attempt to close in on the continuous behaviour of computational processes, without assuming or simulating actual computationality. Common examples of tolerance systems are the following, where d_U denotes the metric on U:

- *Small step systems*: for all $v \in V$ and indices i, $d_U(v_i, v_{i+1}) \leq \epsilon$, for some constant $\epsilon > 0$.
- *Lipschitz systems*: for all $v \in V$ and indices i, $d_U(v_i, v_{i+1}) \leq \kappa |t_i^v - t_{i+1}^v|$, for some constant $\kappa > 0$.
- *Hölder systems*: for all $v \in V$ and indices i, $d_U(v_i, v_{i+1}) \leq \kappa |t_i^v - t_{i+1}^v|^{1/p}$, for some constants $\kappa, p > 0$.

Tolerances were used by Arbib [3] in defining *tolerance automata*, a kind of control systems generating discrete motions m in an action space X by means of step-wise transitions. Transitions are determined by the current state and an input signal, and are assumed to proceed such that every step is bound by the tolerance ξ specified for the automaton, i.e. such that $[m(t), m(t+1)] \in \xi$ for every $t \in \mathbb{N}$.

We conclude that motions as generated by tolerance automata are closely related to discrete maps, with $\mathbb{N}$ as the set of observable times. Let a tolerance automaton be called *metric* if its tolerance space X is metric. One easily shows the following.

Theorem 11. *For every metric tolerance automaton M there exists a tolerance system Q such that the discrete maps produced by Q are precisely the motions generated by M.*

8.2 Computational discrete processing systems

We have argued that discrete computation is captured, very generally, by the notion of discrete processing systems. However, this notion merely captures the phenomenon at the level of the observer, emphasizing that all meaningful information from the ongoing process is obtained by just observing it during the times of some unbounded discrete subset of $\mathbb{R}$.

We now consider the question when, and how, the processing can actually be regarded as a form of computation. It is natural to link this question to our understanding of computations in general. When can discrete maps be viewed as the observable effect of computations as we know them?

Let $Q = \langle U, E, \delta, E_0, V \rangle$ be a discrete processing system, and let $v : Obs(v) \to U$ be one of its discrete maps. Thus, we are observing Q at times $t_0^v (= 0) < t_1^v < \cdots$ and assumed that by means of some 'background process' it has a way of 'bridging' the gaps in between the observable moments. We assumed also, that this process does *not* produce any observable knowledge in between the t_i^v's. If Q is to be computational, then this background process has to be a computational process.

Following this line of reasoning, we can now define what it means for a system Q to be computational. We first define an auxiliary concept for metric spaces.

Definition 41. *U is said to be* embeddable *in metric space A if there exists a bijective isometry h between U and a closed subspace of A.*

Definition 42. *A discrete processing system $Q = \langle U, E, \delta, E_0, V \rangle$ is said to be* computational *if and only if there is a metric space A and a computational process $P = \langle A, E', \delta', E_0, C \rangle$ such that:*

- *U is embeddable in A, i.e. there is a bijective isometry h between U and a closed subspace U_h of A,*
- *$E \subseteq E'$ and $\delta' \upharpoonright U_h = \delta \circ h^{-1}$,*
- *for every discrete map $v : Obs(v) \to U$ of Q, $h \circ v : Obs(v) \to U_h$ is an observable trace of P, and*
- *for every observable trace $m_c : I_c \to A$ of P, $m_c(I_c) \subseteq U_h$ and $h^{-1} \circ m_c : I_c \to U$ is a discrete map of Q.*

If Q is computational, every process P that satisfies the above constraints is called a background process *for Q.*

The first condition in the definition says that U is part of some bigger action space A, in which the 'gaps' are bridged by process P. We require that U_h is *closed*, to ensure that discrete maps of Q do not suddenly get accumulation points (in A) when they are observed in the bigger space. Although we may not want to exclude this in general, it makes perfect sense to do so for observationally discrete systems.

The further conditions say that P acts as a background process of Q, bridging the gaps in between observable time moments and connecting the time steps of any discrete map of Q computationally. The third and fourth condition say that the computations of P do precisely this, and that they do not generate any observable information 'during' any of the gaps. (It means that we may as well assume that $E' = E$, as there is no need for having a broader knowledge space for P.)

The definition precisely expresses the philosophy that any discrete computation, or processing system, is the observed 'image' of an underlying process that fills the in-between intervals of the corresponding discrete map. Note in the definition that P must be *operationally discrete*, if it is to have the stated property. If Q is observationally discrete, then P must be observationally discrete as well.

It should be noted that not just any operationally discrete computational process P can serve as a background process for some discrete processing system Q. We have argued earlier that for all discrete maps v of Q, the times t_i^v $(i = 0, 1, \cdots)$ should be implicitly or explicitly identifiable for the observer. It thus makes sense to assume this for the observable traces of P as well, if P is to serve as a useful background process.

In concrete terms, the latter requirement would mean that the times in all sets I_c $(c \in C)$ would be 'detectable' of 'known', in some way that is facilitated in the functioning of P. In this way, *each discrete map v of Q appears as the 'clocked projection' of a suitable computation of P, where each map v may require an own 'clock' to achieve it.*

We can now finally define 'what a discrete computation is', in the general sense we have aimed at and in perfect analogy to Definition 6.

Definition 43. *A* discrete computation *is any discrete map v for which there is a discrete processing system $Q = \langle U, E, \delta, E_0, V\rangle$ that is computational and has $v \in V$.*

Definition 44. *A discrete computation v is called* observationally discrete *if there is a discrete processing system $Q = \langle U, E, \delta, E_0, V\rangle$ with $v \in V$ that is both observationally discrete and computational.*

Many discrete processing systems are observed using integer-valued or symbolic parameters and are thus of the form $Q = \langle \mathbb{N}^k, E, \delta, E_0, V\rangle$, for some $k > 0$. With the Euclidean metric, the space $\mathbb{N}^k$ is embeddable in $\mathbb{R}^k$, and the question whether Q is computational typically reduces to the question whether there is a computational process $P = \langle \mathbb{R}^k, E', \delta', E_0, C\rangle$ such that every $v \in V$ has a *continualization* in C. Here, a continualization of v is any curve $c_v : [0, \infty) \to \mathbb{R}^k$ such that $c_v \upharpoonright Obs(v) \equiv v$.

Discrete maps $v : Obs(v) \to \mathbb{N}^k$ have numerous continualizations. Every curve obtained by connecting v_i to v_{i+1} by a path in $\mathbb{R}^k$ for $i = 0, 1, \cdots$ results in a continualization of v, and all continualizations of v are of this form. Thus, for Q to be computational, one needs a process P that can generate suitable continualizations in a way that is *computationally explainable.*

If the discrete maps of Q result from a natural process, then this process often is a prime candidate for acting as a suitable background process P. If the maps of Q results from an artificial source like a mathematical machine model, then it is often sufficient to extend this model by having it produce a *straight-line path* in between each v_i and subsequent v_{i+1} that it generates. In this case, the computations in P will all be piecewise linear.

Technically, a background process may need to 'know' v_{i+1} in order to compute the continualization from v_i to v_{i+1}. This may be done in several ways, for instance by using the mechanism of the discrete map at 'double speed' to compute a shadow representation of v_{i+1} in the background by time $\frac{1}{2}(t_i + t_{i+1})$ while the curve c_v goes straight in some direction, and re-directing the straight-line towards the concrete v_{i+1} from then on until this item is concretely reached at time t_{i+1}. Note that c_v remains piecewise linear. We assume that continualizations can be subsumed in background processes as needed and use them routinely in this report.

Discrete processing systems with $U = \mathbb{N}^k$ are only a special case of all discrete systems we consider. In Section 9 we will see how the given definitions enable us to explain how discrete processing systems relate to suitable computational processes in detail.

We conclude with a general observation. Let A, B be metric spaces, with B a subspace of A. We call B *path-connected* in A, if for every pair of points (b_1, b_2) with $b_1, b_2 \in B$ there is a path in A connecting b_1 and b_2.

Theorem 12. *Let $Q = \langle U, E, \delta, E_0, V\rangle$ be a computational discrete processing system, and let $P = \langle A, E', \delta', E_0, C\rangle$ be a background process for Q (as in Definition 42). Suppose that for every two items $u_1, u_2 \in U$ there are a discrete map $v \in V$ and time indexes $i, j \geq 0$ such that $v_i = u_1$ and $v_j = u_2$. Then U_h is path-connected in A.*

Proof. Consider any two points of U_h. By bijectivity of h, there are items $u_1, u_2 \in U$ such that the two points equal $h(u_1)$ and $h(u_2)$, respectively. By the assumption, there must be a discrete map $v \in V$ and times t_i^v, t_j^v such that $v(t_i^v) = u_1$ and $v(t_j^v) = u_2$. By computationality of Q, there must be an observable trace m_c of P with the property that $m_c(t_i^v) = h(u_1)$ and $m_c(t_j^v) = h(u_2)$. The curve segment of computation $c \in C$ between moments t_i^v and t_i^v establishes a path between $h(u_1)$ and $h(u_2)$. As the points of U_h were chosen arbitrarily, it follows that U_h is path-connected in A. $\square$

Theorem 12 provides a necessary condition for a discrete processing system Q to be computational: *its action space U must be embeddable in a metric space in which it is path-connected.*

9 Discrete computation

We now explore discrete computation more precisely. What are the consequences of the definition of computationality for discrete processing systems? Can one say more than what is expressed in the definition, about the relationship between discrete and continuous computation? Have we obtained a framework that proves the hypothesis from Section 2.2, that discrete processing systems can be viewed as 'projections' of general (continuous) processes?

In this section we define the notions of projection, faithful projection, and also of codability for discrete computation. We then prove that *discrete processing systems are computational if and only if they are the (faithful) projection of an operationally discrete computational process.* We also prove a stronger result for the case of observationally discrete processing systems. We then argue that, under suitable extra conditions, effectively codable discrete processing systems always qualify for being computational.

The results show how discrete computation can effectively be viewed as an instance of general, continuous computation. They also show that, mathematically speaking, Turing's claim that *'all machinery can be regarded as continuous'* [42] does not need to make an exception for discrete computation.

We conclude the section with a number of examples that show the applicability of the results, including the important cases of classical *state-based models of computation* (like Turing machines) and *internet searching.*

9.1 Projections

Let $Q = \langle U, E, \delta, E_0, V \rangle$ be a discrete processing system. We have argued that Q is computational if there is a computational process P such that the discrete maps of Q and the observable traces of P correspond and the same knowledge is accumulated along the corresponding computations. This is already a strong relationship, but we expect more from a 'projection'.

Referring to Definition 42, we focus on the mapping of observable traces $m_c : I_c \to A$ of P onto discrete maps $h^{-1} \circ m_c : I_c \to U$ of Q. If an observer is keeping track of a computation c as if it were a computation of Q, he is essentially observing the mapping $c : [0, \infty) \to A$. He knows that $c(t) \in U_h$ for $t \in I_c$ (by definition), and it would clearly be desirable to have the converse as well. If the converse property holds, we call Q the *projection* of P.

Definition 45. *A discrete processing system $Q = \langle U, E, \delta, E_0, V \rangle$ is said to be the* projection *of a computational process $P = \langle A, E', \delta', E_0, C \rangle$ if A and P satisfy the conditions of Definition 42 and for every computation c of P, $c(t) \in U_h$* if and only if *$t \in I_c$.*

The extra condition implies that, when we observe a computation c of P, then we can effectively restrict c to its points in space U_h and project those with h^{-1} onto U to obtain the corresponding discrete map of Q. In other words, if the observer has some criterion for recognizing action items from U_h (or of U if h is known), the condition enables him to know whether the computation he is observing has reached an observable time of the discrete map - short of testing whether δ' is defined in the item.

Recall that observationally discrete processing systems were the most ideal type of discrete systems, from an observer's perspective. The following theorem shows that all observationally discrete processing systems that are computational can be seen as projections of observationally discrete computational processes. It is a very important case of a more general result we prove in this section. The technique to achieve it, is to let every computation of the 'background process' run in an own private segment of the action space, making itself present in U only when it is 'time to be observable' (and only then).

Theorem 13. *An observationally discrete processing system is computational if and only if it is the projection of an observationally discrete computational process.*

Proof. The if-part is immediate, as the definition of projection is a refinement of the definition of computationality. The observational discreteness of the computational process in the definition automatically implies that the discrete processing system involved is observationally discrete as well.

For the only-if part, let $Q = \langle U, E, \delta, E_0, V\rangle$ be a discrete processing system and assume that it is observationally discrete and computational. Let A and P be as per Definition 42. By implication, P must be observationally discrete. We show that A and P can be modified such that the required condition of Definition 45 is satisfied as well.

For any $v \in V$, let A^v be a copy of A and $h^v : U \to A^v$ (a copy of) the bijective isometry as implied by Definition 42. Let A' be the space obtained by taking the *direct sum* of U and all spaces A^v. For all $v \in V$ and $i \in \mathbb{N}$, identify item $h^v(v_i)$ of A^v with item v_i of U. Note that this automatically identifies items $h^v(v_i)$ and $h^w(w_i)$ as well, provided $v_i = w_i$ in U.

Formally, the identifications define an equivalence $\equiv$ on A' and make A' a *quotient space*. By general topology, A' is a pseudometric space, but in our case a stronger fact can be shown. Here we use that the maps v of Q are all observationally discrete, and that for all $v \in V$, $h^v(U)$ is closed in A^v.

Claim. A' is a metric space.

Proof. Let d_U and d_A denote the metrics on the spaces U and A^v, respectively. Define the distance function $d(x, y)$ on A' as follows:

– for any $x \in A'$,
 · $d(x, x) = 0$.
– if $x, y \in U$, then
 · $d(x, y) = d_U(x, y)$.
– if $x \in U$ and $y \in A^v$, then
 · $d(x, y) = \inf\{d_U(x, v_i) + d_U(v_i, v_j) + d_A(h^v(v_j), y) \mid i, j \geq 0\}$.
– if $x \in A^v$ and $y \in U$, then
 · $d(x, y) = d(y, x)$.
– if $x \in A^v$ and $y \in A^v$ then
 · $d(x, y) = d_A(x, y)$
– if $x \in A^v$ and $y \in A^w$ and $v \neq w$ then
 · $d(x, y) = \inf\{d_A(x, h^v(v_i)) + d_U(v_i, w_j) + d_A(h^w(w_j), y) \mid i, j \geq 0\}$.

The definition in case $x \in U$ and $y \in A^v$, could be simplified to $d(x, y) = \inf\{d_U(x, v_i) + d_A(h^v(v_i), y) \mid i \geq 0\}$ but we leave it in the given form for symmetry reasons. We now verify that $d(x, y)$ is a metric on A'.

(I) For all $x, y \in A'$, $d(x, y) = 0$ if and only if $x \equiv y$. The if-part holds by definition. For the only-if part we proceed by case analysis. as follows.

– if $x, y \in U$, then $d(x, y) = d_U(x, y) = 0$ implies that $x = y$ and thus $x \equiv y$, as d_U is a metric.
– if $x \in U$ and $y \in A^v$, then $d(x, y) = 0$ implies that $\inf\{d_U(x, v_i) + d_U(v_i, v_j) + d_A(h^v(v_j), y) \mid i, j \geq 0\} = 0$ and hence that $\inf\{d_U(x, v_i) \mid i \geq 0\} = 0$ and $\inf\{d_A(h^v(v_j), y) \mid j \geq 0\} = 0$ as well. As $\{v_0, v_1, \cdots\}$ and $\{h^v(v_0), h^v(v_1) \cdots\}$ have no accumulation points in U and A^v respectively (in the latter case because the set has no accumulation point in $h^v(U)$ and $h^v(U)$ is a closed

subset of A^v), the only way the infima can be 0 is when $x = v_i$ and $y = h^v(v_j)$ for some $i, j \in \mathbb{N}$. But $d(x, y) = 0$ then implies that $d_U(v_i, v_j) = 0$ and thus that $v_i = v_j$. It follows that $x \equiv y$.

- if $x \in A^v$ and $y \in U$, then $d(x, y) = d(y, x) = 0$ implies that $x \equiv y$ as well, by the preceding case.
- if $x \in A^v$ and $y \in A^v$, then $d(x, y) = d_A(x, y) = 0$ implies that $x = y$ and thus $x \equiv y$, as d_A is a metric.
- if $x \in A^v$ and $y \in A^w$ and $v \neq w$, then $d(x, y) = 0$ implies that necessarily $x = h^v(v_i)$ and $y = h^w(w_j)$ for some $i, j \geq 0$, by a very same argument as above. But then $d(x, y) = 0$ implies that $d_U(v_i, w_j) = 0$ and hence that $v_i = w_j$. It follows that $x \equiv y$.

(II) For all $x, y \in A'$, $d(x, y) = d(y, x)$. This follows from the symmetries in the definition.

(III) For all $x, y, z \in A'$, $d(x, z) \leq d(x, y) + d(y, z)$. This follows by case analysis again. In stead of going through all possible cases, we only show it for the most general situation in which $x \in A^u, y \in A^v$ and $z \in A^w$ with u, v, w all distinct. We now note the following chain of inequalities, for all i, j and k::

- By the fact that h^v is an isometry and the triangle inequality for d_A we have:
$d_U(v_j, v_k) = d_A(h^v(v_j), h^v(v_k)) \leq d_A(h^v(v_j), y) + d_A(y, h^v(v_k))$.
- By the triangle inequality of d_U we have:
$d_U(u_i, w_l) \leq d_U(u_i, v_j) + d_U(v_j, v_k) + d_U(v_k, w_l)$.
- By combining the previous inequalities, we obtain:
$d_A(x, h^u(u_i)) + d_U(u_i, w_l) + d_A(h^w(w_l), z) \leq$
$(d_A(x, h^u(u_i)) + d_U(u_i, v_j) + d_A(h^v(v_j), y)) + (d_A(y, h^v(v_k)) + d_U(v_k, w_l) + d_A(h^w(w_l), z))$.

By taking infima on the right and the left, we obtain that $d(x, z) \leq d(x, y) + d(y, z)$. □

We conclude that A' is a proper action space. Also, we see that U is trivially embeddable in A', by taking the identity mapping of U onto its copy U in the direct sum.

Claim. U is closed in A'.
Proof. Let x be an accumulation point of U and suppose that $x \notin U$, i.e. in A'. Then for some $v \in V$, $x \in A^v$ and there is a sequence of points $x_0, x_1, \cdots$ in U such that $\inf\{d_U(x_i, v_j) + d_A(h^v(v_j), x) \mid i, j \geq 0\} = 0$. This means that necessarily $\inf\{d_A(h^v(v_j), x) \mid j \geq 0\} = 0$, and thus two cases can arise:

(i) there is a j such that $x = h^v(v_j)$, or
(ii) x is an accumulation point of $\{h^v(v_0), h^v(v_1), \cdots\}$ in A^v.

In case (i) it follows that $x \equiv v_j$, contradicting that $x \notin U$ (in the quotient space). In case (ii) it would follow that $x \in U_{h_v}$, by the fact that U_{h_v} is closed in A^v. However, as $\{v_0, v_1, \cdots\}$ has no accumulation point in U, $\{h^v(v_0), h^v(v_1), \cdots\}$ cannot have an accumulation point in U_{h_v} (within A^v). Thus, this case cannot happen either. We conclude that a contradiction occurs in all cases, and that U must contain all its accumulation points in A'. Hence U is closed in A'. □

Finally, define $P' = \langle A', E', \delta'', E_0, C' \rangle$ to be the process with δ'' and C' defined as follows:

- for $x \in U$ and $y \in A^v$: $\delta''(x) = \delta(x)$ and $\delta''(y) = \delta(y)$.
- for any $c \in C$, let c' be the curve $h^v \circ c$ in A^v with v the discrete map of Q that corresponds to its observable trace, i.e. such that $v = h^{-1} \circ m_c$ (cf. Definition 42). In other words: c is the curve that 'connects the gaps' in map v as a computation. Let $C' = \{c' \mid c \in C\}$.

It is straightforward to see that P' is a computational process, and that A' and P' inherit all the properties of A and P required to satisfy Definition 42 for making Q computational. However, we obtain more.

Claim. Q is the projection of P'.

Proof. Consider any computation $c' \in C'$. Let $c' = h^v \circ c$ be the incarnation of a computation $c \in C$, now running fully in the subspace $A^v \subseteq A'$. By construction, c' hits subspace $U = U_h$ of A' only in the items $h^v(v_i)$ $(i \geq 0)$, these precisely being the items which were identified with items in U. Hence, an item visited by c' is in U_h if and only if it is visited in its observable trace. □

This proves the theorem. □

In the given proof, process P' realizes every computation of P by delegating it to an own subspace. If P is a meaningful computational process, which we assume, then it may well be argued that P' is as well.

Before we continue, we prove the following analogue to Theorem 12. Let A, B be metric spaces, with B a subspace of A. We call B *externally path-connected* in A, if for every pair of points (b_1, b_2) with $b_1, b_2 \in B$ there is a path in A connecting b_1 and b_2 such that the part between b_1 and b_2 runs entirely in $A \setminus B$. (In other words: the path 'hits' B only in the points b_1 and b_2 and in no other points of B.) Similar to Theorem 12 one can show:

Theorem 14. *Let $Q = \langle U, E, \delta, E_0, V \rangle$ be a discrete processing system, and let Q be the projection of a computational process $P = \langle A, E', \delta', E_0, C \rangle$ (as in Definition 45). Suppose that for every two items $u_1, u_2 \in U$ there are a discrete map $v \in V$ and time indexes $i, j \geq 0$ such that $v_i = u_1$ and $v_j = u_2$. Then U_h is externally path-connected in A.*

Recall that Theorem 12 gave a (necessary) condition for a discrete processing system to be computational. By Theorem 14 we can now strengthen this condition, at least for discrete processing systems that are observationally discrete, to the requirement that *their action space must be embeddable in a metric space in which it is externally path-connected.* (In the next section we will see that this holds for all discrete processing systems that are computational.)

9.2 Faithful computations

The construction in Theorem 13 very much depended on the fact that Q was assumed to be observationally discrete. However, what can be said about computational discrete processing systems in general? To answer this question, we reconsider the requirements as we distinguished them for projections, refine them and strengthen Theorem 13 considerably.

Let $Q = \langle U, E, \delta, E_0, V \rangle$ be a discrete computational system, let it be computational, and let $P = \langle A, E', \delta, E_0, C \rangle$ be a 'background process' that realizes the discrete maps of Q. Necessarily, P is operationally discrete. Referring to Definition 42 for notations, consider any observable trace $m_c : I_c \to A$ of P, where $I_c = \{t \mid \delta(c(t)) \ \textit{defined}\}$. Let $I_c = \{t_0^c, t_1^c, \cdots\}$ with $0 = t_0^c < t_1^c < \cdots$.

Notation 6 *For any computation c of P, we write $c_i = c(t_i^c)$.*

If the observer is keeping track of c as if it were the computation of a discrete map of Q, he observes c at the times t_i^c $(i \geq 0)$. By computationality, we know that $c_i \in U_h$ for $i \geq 0$ (by definition 42). In Section 9.1 we argued that it would be desirable for the observer to have $c(t) \in U_h$ if and only if $t \in I_c$. This led us to Theorem 13.

In order to generalize Theorem 13 to all systems, we need to look at the notion of projection again and strengthen it. In fact, in addition to what we required for ordinary projections, we now also impose that during all intervals (t_i^c, t_{i+1}^c) (the 'gaps' in between the observations) the observed process is uniquely 'owned' by the interval in which it operates. Thus, observing c, it never happens that we see an action item during some interval (t_i^c, t_{i+1}^c) that also occurs during some other interval (t_j^c, t_{j+1}^c) $(i \neq j)$. It means that the observer is never confused about the gap period he is observing.

Definition 46. *A computation c of background process P is said to be* faithful *if the following properties hold:*

- *for all $i \geq 0$, $c(t) \in U_h$ if and only if $t \in I_c$, and*
- *for all $i, j \geq 0$ with $i \neq j$, the segments $\{c(t) \mid t_i^c \leq t \leq t_{i+1}^c\}$ and $\{c(t) \mid t_j^c < t < t_{j+1}^c\}$ are disjoint,*

The remainder of this subsection is devoted to the notion of faithfulness as it is now defined. Note that faithfulness is a property of the computations of P, the background process to-be of Q, and not of Q itself. The following observation can be made. The proof is similar to that of Theorem 13, but now the construction avoids the assumption of observational discreteness.

Lemma 16. *All computations of P can be assumed to be faithful.*

Proof. Suppose not all computations of P were faithful. We show how to modify P into a new process P'', that satisfies the constraints of P and does have the desired property.

By assumption, every computation c of P corresponds to a discrete map of Q, and thus maps its observable times, i.e. the elements of I_c, into the subspace $U_h \subseteq A$. Now proceed as follows.

For any $c \in C$ and $j \in \mathbb{N}$, let A_j^c be a copy of A and $h_j^c : U_h \to A_j^c$ (a copy of) the trivial identification of U_h in A, i.e. of the bijective isometric image of U in A. Let A'' be the space obtained by taking the *direct sum* of U_h and all spaces A_j^c. For all $c \in C$ and $j \in \mathbb{N}$, identify the items $h_j^c(c_j)$ and $h_j^c(c_{j+1})$ of A_j^c with the items c_j and c_{j+1} of U_h, respectively. Note that this automatically identifies, for example, the items $h^c_j(c_j)$ and $h^e_k(e_{k+1})$ as well, provided $c_j = e_{k+1}$ in U_h. Every A^c_j is thus 'pinned' to (at most) two items in U_h.

Formally, the identifications define an equivalence $\equiv''$ on A'', turning A'' into a quotient space. Once again A'' is a pseudometric space, but we will show that it is metric in our case.

Claim. A'' is a metric space.

Proof. Let d_U and d_A denote the metrics on the spaces U_h and A^c_j, respectively. (Here d_U is the metric of U 'after taking the bijective isometry h' and thus the restriction of d_A to U_h. The metric on A^c_j is likewise a 'copy' of the metric on A.) Define the 'implied' distance function $d(x, y)$ on A'' as follows:

– for any $x \in A''$,
  · $d(x, x) = 0$.
– if $x, y \in U_h$, then
  · $d(x, y) = d_U(x, y)$.
– if $x \in U_h$ and $y \in A^c_j$, then
  · $d(x, y) = \min\{d_U(x, c_j) + d_A(h^v_j(c_j), y),\ d_U(x, c_{j+1}) + d_A(h^v_j(c_{j+1}), y)\}$.
– if $x \in A^c_j$ and $y \in U_h$, then
  · $d(x, y) = d(y, x)$.
– if $x \in A^c_j$ and $y \in A^c_j$ then
  · $d(x, y) = d_A(x, y)$
– if $x \in A^c_j$ and $y \in A^e_k$ and $c, j \neq e, k$ then
  · $d(x, y) = \min\{d_A(x, h^v_j(c_j)) + d_U(c_j, e_k) + d_A(h^e_k(e_k), y),$
  $d_A(x, h^v_j(c_j)) + d_U(c_j, e_{k+1}) + d_A(h^e_k(e_{k+1}), y),$
  $d_A(x, h^v_j(c_{j+1})) + d_U(c_{j+1}, e_k) + d_A(h^e_k(e_k), y),$
  $d_A(x, h^v_j(c_{j+1})) + d_U(c_{j+1}, e_{k+1}) + d_A(h^e_k(e_{k+1}), y)\}$.

One easily verifies that $d(x, y)$ is a metric on A''. Symmetry and the triangle inequality are satisfied because $d(x, y)$ is the implied pseudometric on A'', and thus we only need to show the 'identity of indiscernibles':

– For all $x, y \in A''$, $d(x, y) = 0$ if and only if $x \equiv'' y$.

However, this trivially holds also. The if-part holds by checking the definitions. For the only-if part one only needs to check the different cases, as follows.

– if $x, y \in U_h$, then $d(x, y) = d_U(x, y) = 0$ implies that $x = y$ and thus $x \equiv'' y$, as d_U is a metric.
– if $x \in U_h$ and $y \in A^c_j$, then $d(x, y) = 0$ implies that $\min\{d_U(x, c_j) + d_A(h^c_j(c_j), y)\} = 0$ and hence that $d_U(x, c_j) = 0$ and $d_A(h^c_j(c_j), y) = 0$. It follows that $x = c_j$ and $h^c_j(c_j) = y$, and hence that $x \equiv'' y$.
– if $x \in A^c_j$ and $y \in U_h$, then $d(x, y) = d(y, x) = 0$ implies that $x \equiv'' y$ as well, by the preceding case.
– if $x \in A^c_j$ and $y \in A^c_j$, then $d(x, y) = d_A(x, y) = 0$ implies that $x = y$ and thus $x \equiv'' y$, as d_A is a metric.
– if $x \in A^c_j$ and $y \in A^e_k$ and $c, j \neq e, k$, then $d(x, y) = 0$ implies that necessarily (at least) one of the four expressions in the minimization equals 0, say

$d_A(x, h^v_j(c_{j+1})) + d_U(c_{j+1}, e_k) + d_A(h^e_k(e_k), y) = 0$. Then we necessarily have $d_A(x, h^v_j(c_{j+1})) = d_U(c_{j+1}, e_k) = d_A(h^e_k(e_k), y) = 0$ and thus $x = h^v_j(c_{j+1})$, $c_{j+1} = e_k$ and $h^e_k(e_k) = y$. We conclude that $x \equiv'' y$. The other cases follow similarly. □

We conclude that A'' is a proper action space. Also, we see that U is trivially embeddable in A'', by simply taking h to map U onto its copy U_h in the direct sum.

Claim. U_h is closed in A''.
Proof. Let x be an accumulation point of U_h and suppose that $x \notin U_h$, i.e. in A''. Then for some $c \in V$ and $j \geq 0$, $x \in A^c_j$ and there is a sequence of points $x_0, x_1, \cdots$ in U_h such that $\inf_i d(x_i, x) = 0$, i.e.

$$\inf_i \min\{d_U(x_i, c_j) + d_A(h^c_j(c_j), x),\ d_U(x_i, c_{j+1}) + d_A(h^c_j(c_{j+1}), x)\} = 0$$

For the 'constants' $d_A(h^c_j(c_j), x)$ and $d_A(h^c_j(c_{j+1}), x)$ this means that either $d_A(h^c_j(c_j), x) = 0$ or $d_A(h^c_j(c_{j+1}), x) = 0$, or both. Hence, $x = h^c_j(c_j)$ or $x = h^c_j(c_{j+1})$, or both and $c_j = c_{j+1}$. It follows that $x \equiv'' c_j$ or $x \equiv'' c_{j=1}$ (or both, if $c_j = c_{j+1}$). But, this contradicts that $x \notin U$, i.e. in the quotient space. □

Finally, define $P'' = \langle A'', E', \delta'', E_0, C'' \rangle$ to be the process with δ'' and C' defined as follows (with the obvious identifications):

- for $x \in U_h$ and $y \in A^c_j$: $\delta''(x) = \delta \circ h^{-1}(x)$ and $\delta''(y) = \delta(y)$.
- for any $c \in C$, let c'' be the curve in $\bigcup_j A^c_j \subseteq A''$ defined by: for any $j \in \mathbb{N}$ and $t \in [t^c_j, t^c_{j+1}]$, $c''(t) = h^v_j(c(t))$. As all c in C have observable traces across $[0, \infty)$, every c'' is well-defined. Note that c'' is a curve, and that it 'connects the gaps' in the observable trace m_c as a computation. Let $C'' = \{c'' \mid c \in C\}$.

It is straightforward to see that P'' is a computational process, and that A'' and P'' inherit all the properties of A and P required to satisfy Definition 42 for making Q computational. However, we obtain more.

Claim. Every computation $c'' \in C''$ is faithful.
Proof. We verify Definition 46. By the definition of c'', if $t = t^c_j$ for some $j \geq 0$ (i.e. $t \in I_c$), then $c''(t) = h^c_j(c(t^c_j)) \in U_h$ (in A''). The converse holds as well, as $c'' : [0, \infty) \to \bigcup_j A^c_j$ and every A^c_j 'hits' U_h in the points $c_j = c(t^c_j)$ and $c_{j+1} = c(t^c_{j+1})$ only. For the second condition in Definition 46, consider any $i, j \geq 0$ with $i \neq j$. By the definition of c'' we have $\{c(t) \mid t^c_i \leq t \leq t^c_{i+1}\} \subseteq A^c_i$ and $\{c(t) \mid t^c_j \leq t \leq t^c_{j+1}\} \subseteq A^c_j$. The subspaces A^c_i and A^c_j are disjoint except, possible for an incidence in the (observable) endpoints of the segments. Thus the condition holds. □

We conclude that P'' satisfies all the requirements for being a valid background process and moreover, that P'' is faithful. □

In the given proof, P'' simply realizes every computation of P but now, after an observational moment of a computation has passed, it delegates the next part of the computation to an own subspace, until the next observational moment is reached. If P is a meaningful computational process, then we may consider P'' to be as well. This leads to the following, general result.

Theorem 15. *A discrete processing system is computational if and only if it is the projection of an operationally discrete computational process.*

Proof. The if-part is immediate again, as the definition of (faithful) projection implies computationality. The operational discreteness of the computational process is a necessary condition in order to obtain a valid discrete processing system.

Conversely, let $Q = \langle U, E, \delta, E_0, V \rangle$ be a discrete processing system and assume it is computational, with background process P as in Definition 42. If all computations of P satisfy the condition in Definition 45, we are done. If not, then modify P into a new computational process P'' as in Lemma 16. By the Lemma, P'' is a background process for Q, now with the property that all its computations are faithful. In particular, the requirements of Definition 42 are satisfied. The result follows. □

In Lemma 16, we obtained a stronger property for process P'' than we actually needed for Theorem 15. Consequently, a stronger property can be claimed for the projections we constructed.

Definition 47. *A discrete processing system $Q = \langle U, E, \delta, E_0, V \rangle$ is said to be the* faithful projection *of a computational process $P = \langle A, E', \delta', E_0, C \rangle$ if A and P satisfy the conditions as specified in Definition 42 and every computation c of P is faithful.*

Theorem 16. *A discrete processing system is computational if and only if it is the faithful projection of an operationally discrete computational process.*

One may also notice that the construction in Lemma 16 preserves the property of observational discreteness: if P is observational discrete, then so is the new process P'' (as it has the same observational traces). Thus, we can strengthen Theorem 13 as follows:

Corollary 4. *An observationally discrete processing system is computational if and only if it is the faithful projection of an observationally discrete computational process.*

One may notice that the construction in Lemma 16 did not really need of process $P = \langle A, E', \delta', E_0, C \rangle$ that U_h was closed in A, even though U_h proved to be closed in the action space A'' of process P''. Thus, if one is willing to expend the complexity of constructing P'' from P, one does not need to require closedness of the subspace in the definition of embeddability (Definition 41, in order to obtain our results for discrete processing systems. However, the current definitional set-up seems to be the most natural one.

Finally, note that Theorems 15 and 16 imply that the necessary condition for computationality of discrete processing systems proved in Theorem 14 holds for *all* discrete systems, not just for the ones that are observationally discrete. In fact, by Theorem 16, one can strengthen Theorem 14 as follows.

Let A, B be metric spaces, with B a subspace of A. We call B *strongly externally path-connected* in A, if for any finite set of pairs $\{(b_{1,1}, b_{1,2}), \cdots, (b_{k,1}, b_{k,2})\}$

with $b_{i,1}, b_{i,1} \in B$ $(1 \leq i \leq k)$ the following holds: for any pair $(b_{i,1}, b_{i,2})$ in the set there is a path γ_i in A connecting $b_{i,1}$ and $b_{i,2}$ such that the part of γ_i in between $b_{i,1}$ and $b_{i,2}$ runs entirely in $A \setminus B$, and for any i, j with $i \neq j$ the paths γ_i and γ_j do not intersect except possibly in an endpoint. By a similar argument as before one can show:

Theorem 17. *Let $Q = \langle U, E, \delta, E_0, V \rangle$ be a discrete processing system, and let Q be the faithful projection of a computational process $P = \langle A, E', \delta', E_0, C \rangle$ (as in Definition 47). Suppose that for every finite collection of pairs $\{(u_{1,1}, u_{1,2}), \cdots, (u_{k,1}, u_{k,2})\}$ with items $u_{i,1}, u_{i,2} \in U$ $(1 \leq i \leq k)$, there are a discrete map $v \in V$ and distinct time indices $j_{1,1}, j_{1,2} \cdots, j_{k,1}, j_{k,2}$ such that $v_{i,1} = u_{j_{i,1}}$ and $v_{i,1} = u_{j_{i,2}}$ $(1 \leq i \leq k)$. Then U_h is strongly externally path-connected in A.*

By Theorem 16, Theorem 17 is applicable to all discrete processing systems.

9.3 Codable processing systems

So far we studied discrete processing systems in very general terms, allowing them to act in arbitrary metric spaces. The discrete computations they could generate proved to be exactly equal to 'clocked observations' of continuous computations as generated by a suitable computational process. It is the foundational answer to Turing's 1948 claim that *'all machinery can be regarded as continuous'*. Before we get into some examples, we consider the question whether the theory can be made more tangible.

In practice, many computational processes are observed by tracking and/or graphing some finite number of numeric parameters in discrete or continuous time. Even in symbolic computation, the parameters may often be viewed as being numeric. (We ignore issues of conversion and simply assume Euclidean distance between parameter tuples.) In the discrete case, this leads to discrete processing systems whose action space is embeddable in a metric space $\mathbb{R}^k$, for some $k > 0$.

Definition 48. *A discrete processing system Q is said to be* k-codable*, for some $k > 0$, if Q is computational and admits a background process P which has action space $\mathbb{R}^k$. A discrete processing system Q is said to* codable *if it is k-codable for some $k > 0$.*

The systems we considered in Subsections 4.1 (Finite-state systems) and 4.3 (Reasoning) are seen to be examples of 1-codable processing systems. We note that, technically, the fact that $\mathbb{R}^k \equiv \mathbb{R}$ could be used to argue that every k-parameter system is 1-codable, provided the metric on $\mathbb{R}^k$ is chosen to fit the equivalence. As this usually does not give a very natural metric, we do not consider this to be standard.

Consider a k-codable processing system Q. By definition we know that Q admits a background process P with action space $\mathbb{R}^k$. From the results proved earlier in this section, we know that Q must be the projection of a variant P' of P and the faithful projection of a 'more refined' variant P'' of P. It is intuitive that P' is operationally more detailed than P, and that P'' will be even more so. We make this concrete in the theorems below.

As we noted, the systems we presented in Subsections 4.1 (Finite-state systems) and 4.3 (Reasoning) are instances of 1-codable processing systems. In the case of finite-state machines M, we proved that the corresponding processing system was the projection of a computational process with action space $\mathbb{R}^2$ and the *faithful* projection of a process with action space $\mathbb{R}^3$. We now show that these observations are no coincidence, by proving that both are instances of more general facts.

Theorem 18. *Let $Q = \langle U, E, \delta, E_0, V\rangle$ be a k-codable discrete processing system, for some $k > 0$. Then Q is the projection of a computational process $P' = \langle \mathbb{R}^{k+1}, E', \delta', E_0, C'\rangle$.*

Proof. The argument basically follows that of Theorem 13 but we can do with a simplified form of it. Let Q be as given, and let $P = \langle \mathbb{R}^k, E', \delta'', E_0, C\rangle$ be a background process for Q, necessarily satisfying Definition 42. Let h be the bijective isometry between U and a closed subspace U_h of $\mathbb{R}^k$.

We now create a new background process $P' = \langle \mathbb{R}^{k+1}, E', \delta', E_0, C'\rangle$ for Q as follows. First, we embed $\mathbb{R}^k$ into the space $\mathbb{R}^{k+1}$ by adding a coordinate, i.e. by the map $\rho : \mathbb{R}^k \to \mathbb{R}^{k+1}$ defined by $\rho(\langle x\rangle) = \langle x, 0\rangle$. We see that $\rho\circ h$ is a bijective isometry between U and the set $U_{\rho\circ h} = \rho(U_h) = \langle U_h, 0\rangle \subseteq \langle \mathbb{R}^k, 0\rangle \subset \mathbb{R}^{k+1}$, that $\langle U_h, 0\rangle$ is closed in $\langle \mathbb{R}^k, 0\rangle$ and thus, by general topology, that this set is also closed in $\mathbb{R}^{k+1}$.

Subsequently, define $\delta' : \mathbb{R}^{k+1} \to E'$ such that the interpretation of action items using δ is preserved:

$$\delta'(\langle x, y\rangle) = \begin{cases} \text{if } y = 0 \text{ then:} & \delta''(\langle x\rangle) \\ & \\ \text{if } y \neq 0 \text{ then:} & \textit{undefined} \end{cases}$$

Finally we modify the computations of P. Let $c \in C$ be a computation of P, let $m_c : I_c \to \mathbb{R}^k$ be its observable trace, and let $I_c = \{t_0^c, t_1^c, \cdots\}$ with $0 = t_0^c < t_1^c < \cdots$ (by operational discreteness). As before we write $c_i = c(t_i^c)$. Now define $c' : [0, \infty) \to \mathbb{R}^{k+1}$ as follows:

$$c'(t) = \begin{cases} \text{if } \ t_i^c \leq t \leq \frac{t_i^c + t_{i+1}^c}{2} \ \text{ then:} & \langle c(t), 2 \cdot \frac{t - t_i^c}{t_{i+1}^c - t_i^c}\rangle \\ & \\ \text{if } \frac{t_i^c + t_{i+1}^c}{2} \leq t \leq t_{i+1}^c \text{ then:} & \langle c(t), 2 \cdot \frac{t_{i+1}^c - t}{t_{i+1}^c - t_i^c}\rangle \end{cases}$$

Note that c' is continuous and behaves like c, but now with a piecewise linear component added. Let $C' = \{c' \mid c \in C\}$.

As we assumed that the times t_i^c are implicitly or explicitly known or planned for P, process $P' = \langle \mathbb{R}^{k+1}, E', \delta', E_0, C'\rangle$ is a valid computational process again. In fact, as the observational traces of c and c' are effectively the same (up to an extra zero coordinate) for any $c \in C$, it follows that P' is a valid background process for Q again. Moreover, for any $c' \in C'$ we have that $c'(t) \in U_{\rho\circ h} = \langle U_h, 0\rangle \subseteq \langle \mathbb{R}^k, 0\rangle$ if and only if the last coordinate of $c'(t)$ is 0, i.e. when $t = t_i^c$ and thus $t \in I_c = I_{c'}$. Hence Q is a projection of P'. □

Let a computation $c : [0, \infty) \to \mathbb{R}^k$ be called *monotone* if for any times t_1, t_2 with $t_1 < t_2$ we have $c(t_1) \prec c(t_2)$, where $\prec$ is the common, coordinate-wise partial order on $\mathbb{R}^k$.

Corollary 5. *Let $Q = \langle U, E, \delta, E_0, V\rangle$ be a k-codable discrete processing system, let $P = \langle \mathbb{R}^k, E', \delta'', E_0, C\rangle$ be a background process for Q, and assume that all computations in C are monotone. Then Q is the faithful projection of a computational process $P' = \langle \mathbb{R}^{k+1}, E', \delta', E_0, C'\rangle$.*

Proof. Construct process P' as above. It is easily noted that for every $c \in C$, if c is monotone, then c' is faithful. It follows that Q is a faithful projection of process P'. □

Notice in the corollary that, in order for the computations in P to be monotone, all discrete maps of Q must be monotone as well (with the obvious definition of the latter). Example 4.3 (Reasoning) may be seen as a concrete instance of Corollary 5.

In order to obtain faithful projections in general, we extend the construction in the proof of Theorem 18.

Theorem 19. *Let $Q = \langle U, E, \delta, E_0, V\rangle$ be a k-codable discrete processing system, for some $k > 0$. Then Q is the faithful projection of a computational process $P' = \langle \mathbb{R}^{k+2}, E', \delta', E_0, C\rangle$.*

Proof. The argument now parallels that of Lemma 16, in a simplified form. Let Q be as given, and let $P = \langle \mathbb{R}^k, E', \delta'', E_0, C\rangle$ be a background process for Q that satisfies Definition 42. Let h be the bijective isometry that must exist between action space U and a closed subspace U_h of $\mathbb{R}^k$.

We now create a new background process $P' = \langle \mathbb{R}^{k+2}, E', \delta', E_0, C'\rangle$ for Q as follows. First, we embed $\mathbb{R}^k$ into the space $\mathbb{R}^{k+2}$ by adding two coordinates, i.e. by the map $\chi : \mathbb{R}^k \to \mathbb{R}^{k+2}$ defined by $\rho(\langle x\rangle) = \langle x, 0, 0\rangle$. As before, it follows that $\chi \circ h$ is a bijective isometry between U and the set $U_{\chi\circ h} = \chi(U_h) = \langle U_h, 0, 0\rangle \subseteq \langle \mathbb{R}^k, 0, 0\rangle \subset \mathbb{R}^{k+2}$, that $\langle U_h, 0, 0\rangle$ is closed in $\langle \mathbb{R}^k, 0, 0\rangle$ and, by general topology, that it is thus closed in $\mathbb{R}^{k+2}$.

Next, we define $\delta' : \mathbb{R}^{k+2} \to E'$ such that the interpretation of action items using δ is preserved:

$$\delta'(\langle x, y, z\rangle) = \begin{cases} \text{if } y = 0 \text{ and } z = 0 \text{ then:} & \delta''(\langle x\rangle) \\ \\ \text{if } y \neq 0 \text{ or } z \neq 0 \text{ then:} & \textit{undefined} \end{cases}$$

Finally, we modify the computations of P as follows. Let c be a computation of P, let $m_c : I_c \to \mathbb{R}^k$ be its observable trace, and let $I_c = \{t_0^c, t_1^c, \cdots\}$ with $0 = t_0^c < t_1^c < \cdots$ (by operational discreteness of P). Write $c_i = c(t_i^c)$ as before. Define $c' : [0, \infty) \to \mathbb{R}^{k+2}$ as follows:

$$c'(t) = \begin{cases} \text{if } \; t_i^c \leq t \leq \frac{t_i^c + t_{i+1}^c}{2} \; \text{ then:} & \langle c(t), 2 \cdot \frac{t - t_i^c}{t_{i+1}^c - t_i^c}, 2 \cdot t_{i+1}^c \cdot \frac{t - t_i^c}{t_{i+1}^c - t_i^c}\rangle \\ \\ \text{if } \; \frac{t_i^c + t_{i+1}^c}{2} \leq t \leq t_{i+1}^c \; \text{ then:} & \langle c(t), 2 \cdot \frac{t_{i+1}^c - t}{t_{i+1}^c - t_i^c}, 2 \cdot t_{i+1}^c \cdot \frac{t_{i+1}^c - t}{t_{i+1}^c - t_i^c}\rangle \end{cases}$$

Note that c' is continuous and behaves 'largely' like c, now with two piecewise linear components added. Notice in the $(k+2)$-nd component of $c'(t)$ that for all $i \geq 0$, $t^c_{i+1} \neq 0$. Let $C' = \{c' \mid c \in C\}$.

As before, one may argue that $P' = \langle \mathbb{R}^{k+2}, E', \delta', E_0, C' \rangle$ is a valid computational process. In fact, as the observational traces of c and c' are effectively the same (except for the added zero coordinates), for any $c \in C$, it follows that P' is a valid background process for Q again.

Claim. Every computation $c' \in C'$ is faithful.

Proof. First of all, for any $c' \in C'$ we have that $c'(t) \in U_{\chi \circ h}\langle U_h, 0, 0\rangle \subseteq \langle \mathbb{R}^k, 0, 0\rangle$ if and only if the last two coordinates of $c'(t)$ are 0, i.e. when $t = t^c_i$ for some $i \geq 0$ and thus $t \in I_c = I_{c'}$.

Now consider any item $c'(t)$, for any $c \in C$ and $t \notin I_{c'}$. Let $c'(t) = \langle c(t), \eta, \kappa \rangle$, necessarily with $\eta, \kappa \neq 0$. From the definition of c' we see that $\kappa = t^c_{i+1} \cdot \eta$. Consequently, we can deduce the value of t^c_{i+1}, and thus of t^c_i and i. Given η, this means that (at most) two values of t are possible and both satisfy $t^c_i < t < t^c_{i+1}$, for the time index i we found. Thus, for all $i, j \geq 0$ with $i \neq j$, the segments $\{c'(t) \mid t^c_i \leq t \leq t^c_{i+1}\}$ and $\{c'(t) \mid t^c_j < t < t^c_{j+1}\}$ must be disjoint. □

We conclude that Q is a faithful projection of P'. □

The examples in Sections 4.1 (Finite-state systems) and 4.3 (Reasoning) show that the given results are essentially best possible as regards dimensionality.

9.4 Discrete processing systems - examples

The examples in Sections 4.1 (Finite-state systems) and 4.3 (Reasoning) were recognized as examples of discrete processing systems. We now give a few more examples that show the general applicability of the results.

State-based systems In the classical theory of computation, many 'machine models' are known that are all based on some kind of recipe for the stepwise, systematic transformation of a data configuration of some kind. At any moment in time, the data configuration is typically composed of the current state of the processor(s), the data stored in memory, and the present interaction status (describing the status of the various input and output channels). In many cases, the data configurations can be encoded in a finite *instantaneous description* of the system, using symbols from some finite alphabet Σ, with proper tokens for separating the various components of the configuration.

Consequently, a classical machine model can be seen as a discrete processing system $Q = \langle \Sigma^*, E, \delta, E_0, V \rangle$ that produces discrete maps v corresponding to the allowable runs of the system, with $v_\sigma : \mathbb{N} \to \Sigma^*$ such that $v_\sigma(t) =$ *the t-th instantaneous description entered in run σ*. We take Σ^* to be a metric space with distances based on viewing strings in Σ^* as integers in $||\Sigma|| + 1$-ary notation and thus as unique integers in $\mathbb{N}$. We let E be the knowledge space of possible outputs, and define $\delta : \Sigma^* \to E$ such that $\delta(\alpha) = \kappa$ when α is the encoding of an instantaneous description with output κ and $\delta(\alpha) = undefined$ otherwise.

This generalizes the example in Section 4.1 (Finite-state systems) to a very broad class of machine models of discrete computation, from Turing machines and random-access machines to all sorts of extended versions of it. Let M be any classical machine from this class, and let Q_M be the discrete processing system corresponding to M.

Lemma 17. *Q_M is a 1-codable processing system.*

Proof. As $\mathbb{N}$ is closed in $\mathbb{R}$, we see that Σ^* with the chosen metric is embeddable in $\mathbb{R}$, say by means of the isometry h. Next, every discrete map v generated by Q_M can be continualized to a curve in $\mathbb{R}$ by adding ('computing') paths between successive items $h(v_i)$ and $h(v_{i+1})$ in $\mathbb{R}$, for $i = 0, 1, \cdots$ (as in Section 8.2). In this way we obtain a background process $P = \langle \mathbb{R}, E', \delta', E_0, C \rangle$ of Q_M, which is computational by assumption on M. Hence Q_M is 1-codable. $\square$

It is reasonable to assume that the continualizations referred to in the proof can be achieved by adding 'straight-line paths' between the successive items $h(v_i)$ and $h(v_{i+1})$ for $i = 0, 1, \cdots$. In this case the background process P we constructed will consist of piecewise-linear computations only. Applying Theorem 19, this gives the following result:

for every discrete-state model of computation M, the corresponding discrete processing system Q_M is the faithful projection of a computational process P_M with action space $\mathbb{R}^3$ and piecewise-linear curves as computations.

The result may be seen as the strongest characterization of discrete computation in the classical sense of the theory of automata. The actual action space of P_M may be a $\mathbb{R}^k$ for some k different from 3, depending on the encoding of M's instantaneous descriptions and the metric on them. It is the general result underlying the case of finite-state systems discussed in Section 4.1.

Internet searching The final example probes the domain of *internet searching*, from the viewpoint of the current notions. It refines the example as we originally envisioned it in [45].

Let W denote the worldwide web, M a computer connected to W, and π the client program of a search engine running om M. We assume π is fed queries and attempts to answer them using information from W. We begin by viewing the actions of π as that of a suitable discrete processing system.

We assume that π is operating in discrete time steps, at all times acting on a finite set of parameters in numeric encoding. Let U be the space consisting of all instantaneous descriptions of π, running on M and accessing parts of W. Clearly, U may be seen as a, potentially very large, metric space isometric to an $\mathbb{N}^k$ for some, potentially very large, $k > 0$.

Let E be the space of all 'query;answer' pairs as they may be known from ('found on') W, enriched with a straightforward inference relation defined on them. U and E are linked by the map $\delta : U \to E$ that delivers ('reads out') the content of the 'query' and 'answer' parameter fields to the observer as they are contained in the relevant instantaneous description. (This could give undefined if the search is not finished or came up with no answer. If the answer is a list of facts or links, δ might select a top percentile of that list.) Furthermore, let base set E_0 consist of the instantaneous descriptions of π corresponding to a valid starting configuration, with any allowable (expressible) query q in the query field and 'blank' in the answer field of it.

For any query q, let v_q be the discrete map $\mathbb{N} \to U$ that lists the consecutive instantaneous descriptions that result from searching the web with q. If the search 'stops', v_q is assumed to simply cycle on the answer forever. The semantic maps reads it out when called. Let V be the set of all discrete maps as they are obtained this way. We contend that the discrete processing system $Q_\pi = \langle U, E, \delta, E_0, V \rangle$ naturally represents 'internet searching' as a case of discrete computation.

Lemma 18. *Q_π is a k-codable processing system, where k is the 'dimensionality' of π.*

Proof. By assumption, π's action space U is isometric to $\mathbb{N}^k$, where k is the number of parameters encoded in its instantaneous descriptions. As before, $\mathbb{N}^k$ is closed in $\mathbb{R}^k$, and thus U is embeddable in this very space.

For every query q, the discrete map v_q can be continualized as in the previous example, by adding piecewise-linear paths between consecutive items of v_q. The resulting process is a valid background process for Q_π with an action space equal to $\mathbb{R}^k$. Thus Q_π is k-codable. $\square$

Using that $\mathbb{N}^k \equiv \mathbb{N}$ (i.e. have equal cardinality), one may define an implied metric on $\mathbb{N}^k$ such that $\mathbb{N}^k$ becomes, in fact, isometric to $\mathbb{N}$. The semantic map δ can be adapted such that it 'decodes' the isometry and gives the same output as the original δ. Thus, if one is willing to afford the encoding, Q_π actually becomes a 1-codable system. We do not consider this a very natural representation.

We conclude that internet searching is definitely computational, according to our framework. Leaving matters of coding aside, the following result is obtained:

> *for every search engine π, the corresponding discrete processing system Q_π is the faithful projection of a computational process P_π with action space equal to $\mathbb{R}^k$ (some $k > 0$) and piecewise-linear curves as computations.*

This concludes our analysis of the phenomenon of discrete computation, seen as an observed outcome of continuous computational processes.

9.5 Reflection

In the first part of this report we took Turing's claim that *'all machinery can be regarded as continuous'* [42] and developed the theoretical framework to support it. This lead to the notion of computational processes and to a definition of computations in a broad sense. In Sections 8 and 9 we studied the implication for the domain of discrete computation.

In his 1948 report [42], Turing did not present an overall framework that would encompass both the continuous and the discrete view of 'machinery'. He merely stated that

> *[...] All machinery can be regarded as continuous, but when it is possible to regard it as discrete it is usually best to do so.*

This strongly suggests that he perceived discrete computation as a special view of continuous computation, namely when this view is sufficiently informative of what the computation achieves.

The attempt to develop a general philosophy of discrete computation, led us to the notion of discrete processing systems and the discrete maps they can potentially generate. Here, discrete processing systems are presented as discretely observed computational processes, in a very broad sense and conforming to the idea expressed by Turing.

In Sections 8 and 9 we have subsequently shown that this approach is viable. The theoretical framework is intuitive and of an attractive generality. Turing's clear suggestion that it should be possible to regard discrete computation as an instance of what general, continuous, machinery can do, is well reflected in the definitions and in the various results that express that 'discrete processing systems are projections of computational processes'.

In the following sections, we turn to the study of general computational processes again, knowing that discrete computation is a special case of it.

10 Functional processes and step-generation

The challenge in understanding computational systems of any kind is to discover the driving regulatory mechanisms and laws that can explain, and possibly predict, the computations they generate in their action space. This leads us to a further aspect of discrete computations, namely the challenge of explaining the knowledge-carrying steps as the result of a 'law' of some kind. Could an attentive observer be led to the discovery of an unmistakable *step-generating mechanism*, based on some regularity in the computations? Could this mechanism even have the character of a *program*, whatever this might be?

In our epistemic philosophy of computation there is no standard notion of 'step' as we know it in traditional machine models. At best one could define a 'step' as the action of a computational process during an interval in between two action items of A with defined δ-values, with some notion of 'interval' that is set by an observer. It is generally not applicable, as it hides the underlying computation and its causal effects over time. Nevertheless, an observer may still want to track a computation in terms of 'steps', with the steps being generated by a 'repeatable' process (triggered anew every time a step is completed).

In this section we will show that this scenario is feasible for those computational processes that are both operationally discrete and operationally or observationally deterministic. (Recall that observationally discrete processes are operationally discrete too, by Theorem 10, and also that all observationally deterministic processes are repeatable, by Lemma 12.)

To achieve it, we resume the discussion of repeatable processes from Subsection 7.1 and define what it means when such a process may be said to compute a *function*. We then show that repeatable processes can indeed be made to serve as step-generating mechanisms. Finally, we comment on the analogy between step-generating processes and 'programs'.

10.1 Functions from repeatable processes

Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process, and assume that P is repeatable. It means that the computations $c \in C$ are completely determined by the δ-value of their initial position. Consequently, the computations of P lead to a functional relation between the initial knowledge they start from and the knowledge they produce in the resulting course of action.

This property of repeatable processes allows us to compute *functions* over the knowledge space E. We just initialize P at a feasible initial point, watch the unique computation that it generates from there, and extract a unique outcome from the computation, by some criterion. In this report we chose to run a computation to infinity, observing whether it ends up converging to a specific knowledge item. Of course we rely on the observer's capabilities for inspecting the computations here.

Definition 49. *A computation $c \in C$ is said to converge to knowledge item $e \in E$ if there is a time $u \in [0, \infty)$ such that $\delta(c(t)) = e$ for all $t \geq u$.*

We say that a computation c 'eventually converges' if there is a time u and a knowledge item $e \in E$ such that $\delta(c(t)) = e$ for all $t \geq u$. In this case we also say that c 'converges at time u'.

We could have used other definitions here. For example, we could have used 'strong convergence', requiring that a computation converges when it actually stabilizes on one specific *action item* of A. However, this stronger criterion is not within the normal observational powers of the observer, who only has the semantic map δ to 'view' computations. Therefore we do not use this stronger version here.

In the following definition we assume that $P = \langle A, E, \delta, E_0, C\rangle$ is a repeatable process.

Definition 50. *The* repeatable function $f_P : E_0 \to E$ *computed by P is the partial function determined as follows. For any e_0 with $e_0 \in E_0$:*
- *if there is no $c \in C$ with $\delta(c^{init}) = e_0$, then $f_P(e_0) =$* undefined.
- *if $c \in C$ has $\delta(c^{init}) = e_0$ and c does not converge, then $f_P(e_0) =$* undefined.
- *if $c \in C$ has $\delta(c^{init}) = e_0$ and c converges to $e \in E$, then $f_P(e_0) = e$.*

By the nature of repeatable processes, f_P is well-defined. However, as in the classical theory of computation, we have no means for telling beforehand whether $f_P(e_0)$ is 'defined' or not, for any particular starting item e_0. This has very similar consequences as in the classical theory.

If repeatable processes are used to compute functions over their knowledge domain, we also refer to them as *functional processes*.

Computable functions If we let E_0 correspond to a space of 'input values' and all of E to a space of potential 'output values', then repeatable processes give a means for studying computable functions, using the broad notion of computation as defined here.

It would be interesting to develop a theory of computable functions, now using the notion of repeatable functions. This would require the ability to manipulate and/or combine different repeatable functions, hence repeatable processes, in ways that go beyond our present assumptions. For example, one might need that convergence is in fact *observable*, i.e. that the item to which a computation converges in fact contains some knowledge that the action of the process has converged. We leave the development of a general theory of repeatable functions as an interesting topic.

10.2 Steps in operationally discrete processes

Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. From an observational viewpoint, the computations of P unfold as curves in A, passing through action items whose 'information content' can be observed (extracted) by means of the semantic map δ. We now ask whether the curves, as they unfold, may result from some kind of 'law' that governs the underlying generative process.

If A would be a continuous vector space like $\mathbb{R}^n$, one might attempt to describe the curves by multivariate Taylor expansions and roll them out in discrete steps. In our case, we do not have this analytic possibility. In stead, we consider another possibility for the observer to view the development of the computations in time, namely that of determining *steps* along every curve, in some meaningful way, and having a 'modified' computational process generate the steps.

In general one will not be able to subdivide a computation into steps meaningfully. No matter how small the steps are taken, one will loose loosing sight of all of the computation and all of the information (knowledge) that it generates during a step. There is a clear exception, namely the case of (operationally or observationally) *deterministic*, *operationally discrete processes*.

In this section we will show that the computations of these processes always allow for a natural subdivision into steps, in such a way that no actual information is lost 'during a step'. For completeness, we recall the definition of operationally discrete processes from Section 7 (Definition 32). The processes were discussed at length in Sections 7 and 9.

Definition 51. *A process $P = \langle A, E, \delta, E_0, C \rangle$ is called* operationally discrete *if for every computation $c \in C$, the set $\{t \mid \delta(c(t)) \text{ is defined}\}$ has no accumulation points (in $\mathbb{R}$).*

Example 9. Deterministic, operationally discrete computational processes are not rare. To see this, consider the discussion of state-based systems in Subsection 9.4. It was argued that all classical machine models could be described as computational systems over an action space consisting of their 'instantaneous descriptions'. (The descriptions need not be finite, as in the case of Abstract State Machines.) In all these models, the stepwise action of their underlying program can be effectuated in this space, while maintaining the stepwise proximity requirement that is needed for the continuity of the resulting curves. (For Abstract State Machines, this requires the Bounded Exploration Postulate.)

Specialize this setting to deterministic machine models that are driven by a single program π, operating on pre-defined inputs. If the program terminates in finite time, we let it cycle on its final configuration so its (discrete) computations are valid as discrete maps on infinite time. It may be argued that for every π, the *1-codable discrete processing system* that models the processing of inputs by a machine of this kind, has a background process that is operationally discrete and operationally deterministic.

Characteristic properties The simple, but relevant property of discrete computational processes that we will exploit, is the following. We use the following helpful notation.

Notation 7 *For $c \in C$ and $T \geq 0$, let* $\mathrm{next}_c(T) = \min\{t > T \mid \delta(c(t)) \text{ is defined}\}$ *if the minimum exists, and* $\mathrm{next}_c(T) = \text{undefined}$ *otherwise,*

For discrete computational processes, the value of $next_c(T)$ is always *defined* when $\{t > T \mid \delta(c(t)) \textit{ is defined}\}$ is non-empty, i.e. for every time $T \in [0, \infty)$, there is a *first* later moment t for which $\delta(c(t))$ is defined, unless there is no $t > T$ at all for which $\delta(c(t))$ is defined. (This follows from Lemma 14.)

For any computation $c \in C$ and time $T \geq 0$ with $\delta(c(T))$ defined, the action interval from T up to $next_c(T)$, if the latter is defined, qualifies as a 'step' in c: after all, for any time t with $T < t < next_c(T)$, one has $\delta(c(t))$ = 'undefined'. Nothing is 'missed' if we skip these intermediate times. It is fine that steps depend on T, i.e. that they are potentially of different length depending on their starting moment T.

It is a different matter that, in this way, the notion of 'step' apparently depends on the particular computation c we happen to be considering. In the deterministic case, this is less of a problem than it seems at first sight. The following fact is an immediate consequence of the distinctive behaviour of all deterministic, operationally discrete processes.

Lemma 19. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally discrete, and let P also be operationally or observationally deterministic. Let $c, d \in C$, $\alpha \in A$ and $t_1, t_2 \in [0, \infty)$ be such that $c(t_1) = d(t_2) = \alpha$, with $\delta(\alpha)$ defined. Then, both* $\mathrm{next}_c(t_1)$ *and* $\mathrm{next}_d(t_2)$ *exist, or both have the value* undefined. *If both exist, there is a $\rho > 0$ such that* $\mathrm{next}_c(t_1) = t_1 + \rho$ *and* $\mathrm{next}_d(t_2) = t_2 + \rho$ *(and clearly $c(t_1 + \rho) = d(t_2 + \rho)$).*

Proof. If $c(t_1) = d(t_2) = \alpha$ with $\delta(\alpha)$ defined, then $c^{t_1} = d^{t_2}$, by the fact that P is operationally or observationally deterministic. This means that $\{t > t_1 \mid \delta(c(t)) \textit{ is defined}\} = \{t > t_2 \mid \delta(d(t)) \textit{ is defined}\}$. Consequently, as P is operationally discrete, both $next_c(t_1)$ and $next_d(t_2)$ exist, or both have the vale *undefined*. If both exist, Lemma 14 and the fact that $c^{t_1} = d^{t_2}$ imply that $next_c(t_1) = t_1 + \rho$ and $next_d(t_2) = t_2 + \rho$ for some $\rho > 0$. By the fact that $c^{t_1} = c^{t_2}$, it follows that $c(t_1 + \rho) = d(t_2 + \rho)$. □

It follows from the lemma that, once computations begin to *concur* (even in the eyes of the observer), then so do the step notions for them.

Extension Ultimately, we want more than just a notion of 'step' alone. We want to be able to *link* all 'consecutive' observable action items α and β along a curve (or: computation) c in some lawlike manner, i.e. such that β follows from α by some modified process. Lemma 19 suggests that this may be like generating a *gradient flow* leading from α to β, and from β to the next observable action item, and so on.

This interpretation turns out to be valid, for all deterministic discrete processes. Recall the following definition (cf. Definition 8).

Definition 52. *An action item $\alpha \in A$ is said to be* visited *by computation $c \in C$ if there is a $t \in [0, \infty)$ such that $c(t) = \alpha$. We say that α is* observable *if $\delta(\alpha)$ is defined.*

If an action item α is visited by c, one usually has a moment t in mind for which $c(t) = \alpha$. The following observation extends Lemma 19.

Theorem 20. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally discrete, and also operationally or observationally deterministic. Let $\alpha \in A$ be observable, i.e. such that $\delta(\alpha)$ is defined. Then, if α is visited by c, and later items x along c follow with $\delta(x)$ defined as well, then this is the case for all computations of P that visit α. Moreover, the next action item x that follows α and is observable, is uniquely determined, i.e. the action item is the same for all computations of P that visit item α.*

Proof. Let α be visited by c, let $\delta(\alpha)$ be defined, and let t_1 be any time such that $c(t_1) = \alpha$. Assume that $\{t > t_1 \mid \delta(c(t))\ \textit{is defined}\}$ is non-empty. P is operationally discrete, and thus $\textit{next}_c(t_1)$ exists . Say we have $c(\textit{next}_c(t_1)) = \beta$, for some $\beta \in A$.

Consider any other time t_2 and any computation $d \in C$ (possibly different from c) such that $d(t_2) = \alpha$. Given that P is operationally or observationally deterministic, it follows that $c^{t_1} = d^{t_2}$. As we assumed that $\{t > t_1 \mid \delta(c(t)\ \textit{is defined}\}$ is non-empty, so is $\{t > t_2 \mid \delta(d(t))\ \textit{is defined}\}$. Because c and d coincide from times t_1 and t_2 onward respectively, we have that $\textit{next}_d(t_2)$ is defined also and, hence, that $d(\textit{next}_d(t_2)) = \beta$, by applying Lemma 19.

We conclude that, if α be visited by any computation c, and further action items with defined δ-value follow along c, then these items follow along the curve of any computation passing through α, and β must be the next action item with a defined δ-value that is visited, by all of them □

10.3 Step functions and their iteration

We have shown that computations of deterministic, operationally discrete processes P can be subdivided into observer-relevant 'steps', in such a way that during the steps no meaningful information is lost and, moreover, consecutive steps are linked in a deterministic fashion. (Without the latter property, the observation would hold for all operationally discrete processes.)

From a philosophical perspective, it is interesting to note that the time interval spanned by a step, or, better yet. the arc length between its end points, may be seen as a measure for how difficult it is to generate a new piece of knowledge. We will discuss these aspects later, in Section 11.

By the stated property, computations of P may be looked at in a step-by-step way, as a sequence of consecutively generated steps. We now elaborate on the idea that this sequence might be seen as the result of an *iteration*, namely of a separate, step-generating process.

Step functions To begin with, we consider the generation of steps in computations of P in more detail. Theorem 20 showed that, if an observable action item α is visited by a computation c, then the next observable action item β that is visited, if it exists, is *uniquely determined*, for all computations that ever visit α. Thus, the succession of steps is governed by a fixed rule that maps observable action items to 'next' observable action items. It is expressed in the following definition.

Definition 53. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally discrete, and let it also be operationally or observationally deterministic. The* step function *determined by P is the partial function $g_P : A \rightarrow A$ defined as follows, for any $\alpha \in A$:*

- *if α is not visited by any computation in C, then $g_P(a) =$* undefined.
- *if $\delta(\alpha)$ is undefined, then $g_P(a) =$* undefined.
- *if α is visited by, say, computation $c \in C$ and $\delta(\alpha)$ is defined, and there is at least one more action item with a defined δ-value α that is visited by c after it visited α, then $g_P(\alpha) = \beta$, where β is the next action item with a defined δ-value visited by c.*
- *if α is visited by, say, computation $c \in C$ and $\delta(\alpha)$ is defined, and there are no more action items with a defined δ-value that are visited by c after it visited α, then $g_P(\alpha) =$* undefined.

Theorem 21. *The step function g_P is well-defined, for any process P that is operationally discrete and operationally or observationally deterministic.*

Proof. This follows immediately from Theorem 20. In particular, the 3-rd and 4-th clauses of the definition are independent of the particular computation c that visits item α, by the properties of P. $\square$

Clearly, the consistency conditions that are required for all computations of P, manifest themselves at the step level. In particular we have the following.

Proposition 7. *For all $\alpha \in A$, if $g_P(\alpha) \neq undefined$, then $\delta(\alpha) \models^* \delta(g_P(\alpha))$, where $\models$ is the inference relation of E.*

Proof. This follows because, when $g_P(\alpha) \neq undefined$, there must be a computation $c \in C$ and times $t_1, t_2 \in [0, \infty)$ with $t_1 < t_2$ such that $c(t_1) = \alpha$ and $c(t_2) = g_P(\alpha)$. Thus $\alpha \models^* g_P(\alpha)$, as implied by the consistency condition that must hold for c. $\square$

Iteration When we graphically represent g_P as a function on A, with arrows pointing from α to $g_P(\alpha)$ for all items $\alpha \in A$ with $g_P(\alpha)$ defined, then we obtain the field of *computational flow* for the computation of P. Using the notion of *discrete traces* defined in Section 4, we may observe now that any computation c of P may be fully traced by starting in c^{init} and iterating g_P from this point on, step by step. More generally:

> *the discrete traces of the computations in C are fully determined by the starting points in the set $C_0 = \{c^{init} \mid c \in C\}$ and the step function g_P.*

Finally, the following theorem summarizes that the discrete traces indeed give us all the information that P can generate, and that no information is lost at the level of the steps. For $i \in \mathbb{N}$, let g_P^i denote the i-th iterate of g_P, where g_P^0 is understood to be the identity map on A.

Theorem 22. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally discrete, and let it also be operationally or observationally deterministic. Then $E_P = \delta(\bigcup_i g_P^i(C_0))$, where $C_0 = \{c^{init} \mid c \in C\}$.*

In the characterization of E_P in Theorem 22, the time-dimension of the computations is no longer visible.

10.4 Step-generating processes

For the processes P we are considering, it would be interesting if the values of g_P could be computed in 'unit time' (whatever this means at this stage). It is a common assumption in classical views of discrete computation. But is it always the case, in general?

Differentiability For computing g_P-values one might use process P itself. One might try to start a computation of P at any allowable $\alpha \in A$ with $\delta(\alpha) \in E_0$, and rely on the observer to *detect* when a step is completed, by asking him to monitor the computation until he sees that a next action item is visited for which δ is defined. We believe an observer will not be able to do this in general, regardless of whether he understand the underlying mechanism or not. We prefer the step function to be delivered by a computational process again.

If we want to compute g_P, we need a repeatable process that can produce it. In the following definition we specify the requirements for this.

Definition 54. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be observationally discrete, and let it also be operationally or observationally deterministic. P is said to be* differentiable *if there is a repeatable computational process $R_P = \langle A', A, \delta', A_0, C'\rangle$ such that:*

- $A_0 = \{\alpha \mid \alpha$ is visited by a computation in C *and* $\delta(\alpha)$ is defined$\}$,
- *the repeatable function computed by R_P is equal to g_P: $f_{R_P} = g_P$.*

If a process R_P as required in the definition exists, it is called a *step generating process* for P. A step generating process 'explains' the observable steps of the computations of P. Note that the knowledge space of R_P is equal to A, the action space of P. We assume the trivial inference relation on A. The set A_0 need not to be known explicitly. Step generating processes are not necessarily unique.

Clearly, if P is differentiable, a step generating process for P exists only if there is a mechanism that effectuates it. Then, when R_P is known, the observable part of any computation $c \in C$ can be generated provided R_P is initialized to the proper starting state with δ'-value equal to c^{init}. Then one should run R_P, wait for it to converge, and repeat. Each time R_P converges, it delivers the next observable action item of c as 'knowledge'. In practical use, convergence should be observable as soon as it occurs.

Existence of step generating processes The computability of the step generating functions g_P is an interesting question for all deterministic, operationally discrete processes P. If the values of g_P could be assumed to be computable in 'unit time' (whatever this means here), then the computations of P may as well be re-timed and run on the standard time scale $0, 1, \cdots$ in stead. This is a common assumption in classical views of discrete computation, which finds its proper perspective here.

One may argue that a step generating process for P should be derivable from P itself. It requires that the mechanism that underlies the behaviour in between the observable stages of P's computations can be made concrete and understood, in a computational sense. Steps are normally implied by the control of the process, and play a role in the perception of a process as a discrete processing system, in the sense of Definition 36.

Overall, if a process P is deterministic and operationally discrete, then it must have the following property, if its computations are to be generable by individual step generation:

> *P should be differentiable, and it should admit a step generating process R_P of which all computations are (observably) convergent.*

As an illustration, we argue that many classical machine models have precisely this property. Philosophically, it means that the property seems to isolate the essence of computational discrete processing in classical theory.

Theorem 23. *Let $\mathcal{M}$ be a classical deterministic machine model, governed by a program π. Then $\mathcal{M}$'s behaviour can be modeled by a process $P = \langle A, E, \delta, E_0, C \rangle$, with C consisting of the computations that correspond to the processing of predefined inputs under π. Moreover, P can be defined such that it is differentiable and has a step generating function whose computations all converge.*

Proof. (Sketch.) In Example 9 we argued that $\mathcal{M}$ may be seen as a '1-codable discrete processing system', with a background process P that is operationally discrete and operationally deterministic. The step generating function g_P is seen to correspond precisely to the function that maps instantaneous descriptions of $\mathcal{M}$ to instantaneous descriptions of $\mathcal{M}$, by the stepwise action of π. This immediately implies that P is differentiable, as the values of g_P can simply be computed by the repeatable process R_P that works like P when given (the knowable action item corresponding to) an instantaneous description to start from, but now converges as soon as the next instantaneous description is obtained as knowledge item. R_P is easily designed such that all its computations converge. $\square$

In the case of the theorem, one may well assume that action items α of R_P have a defined δ'-value if and only if α corresponds to (the knowable action item corresponding to) an instantaneous description of $\mathcal{M}$. If the machine does not have any instantaneous descriptions that lead to themselves again 'in one step', then R_P may be assumed to be observationally deterministic. This will be the case for many classical models.

10.5 Reflection

The notions in this section were tuned to a particular class of processes, namely to discrete processes that are deterministic. These processes appear to abstract the most salient properties of many classical models of discrete computation that have studied. It is intriguing that the functional properties of deterministic, operationally discrete processes seem sufficient to obtain all characteristics of stepwise processing normally associated with discrete systems.

Even more far-reaching is the following observation. Recall that, in our approach to computation, we made *no* assumptions on what mechanism precisely triggers the underlying computational processes. In particular, no notion of *program* was assumed, nor needed. However, for the case of deterministic, operationally discrete processes P, a notion of program seems to emerge naturally when we look at their behaviour.

To argue this, assume that P satisfies the property we identified above. Thus, assume that P admits a step generating process R_P whose computations are all (observably) convergent. Then R_P may be seen as a program for P, i.e. for its computations. Moreover, this may well define the notion of program in a generic way, very generally and independent of any descriptive context like a programming language or algorithmic concepts. It is the proper notion for the general framework here. It would be interesting to develop a 'model of (discrete) computation' based on this notion.

Clearly, if we would require a process R_P to be operationally discrete, and also operationally or observationally deterministic again, then this would allow us to define a concept of *higher-order differentiation*. For example, a second-order step-generating function would be step-generating for a first-order process like R_P. We leave the exploration of this possibility as an interesting topic for further study.

Finally, the fact that step generating processes emerge naturally for the types of processes we are considering, is interesting from a philosophical viewpoint. After all, it points to the potential existence of an explicit mechanism of causation in their generation of knowledge (or 'emergence'): each time a new item is reached in the generative process, a next causal piece of knowledge is generated (by the next step).

11 Complexity of computational processes

Given our current understanding of computations and of computational processes, the question arises whether and how common notions of 'computational behaviour' may be captured in the framework. Is it possible to define notions related to the limiting behaviour of computations and notions of computational rigor or difficulty in the present framework? How might it relate to the computation of concrete knowledge sets?

In this section we first discuss some aspects of the behaviour of computations 'in the limit', namely the possibility of stabilization and some general properties of loops. Next we consider so aspects of structural and computational complexity and prove several general results for them. Finally, we reflect on the question what properties of computational processes may be 'knowable'.

11.1 Stabilizing computations

Let $P = \langle A, E, \delta, E_0, C \rangle$ be some computational process. If a computation is traced, it may well occur that it gets stuck at a particular action item. We consider the phenomenon in general terms and discuss how it may influence our view of how knowledge may be generated by computations.

Definition 55. *A computation $c \in C$ is said to stabilize at time $u \in [0, \infty)$ if $\delta(c(u))$ is defined and $c(t) = c(u)$ for all $t \geq u$.*

We will say that a computation c 'eventually stabilizes' if there is a time u such that c stabilizes at time u. The following observation follows easily from the continuity of computations in A.

Lemma 20. *If a computation stabilizes at a finite time u, then there is an earliest moment at which it stabilizes.*

Proof. Let $c \in C$ be an arbitrary computation, and assume that the set $S_c = \{u \mid c \textit{ stabilizes at time } u\}$ is non-empty. Let $\nu \geq 0$ be the greatest lower bound of S_c. If $\nu \in S_c$, we are done.

Suppose, on the other hand, that $\nu \notin S_c$. Then there must be an infinite decreasing sequence $\nu_1 > \nu_2 > \cdots$ in S_c that converges down to ν. Let t be an arbitrary time moment with $t > \nu$, and let $i \geq 1$ be an index such that $t \geq \nu_i$. (Such an index must exist.) Then $c(t) = c(\nu_j)$ for all $j \geq i$, as all ν_j are elements of S_c. Hence $c(t) = \lim_j c(\nu_j) = c(\lim_j \nu_j) = c(\nu)$, by continuity of c. Consequently, $\delta(t) = \delta(\nu)$ and, in particular, $\delta(\nu)$ is defined as $\delta(t)$ is. Thus $\nu \in S_c$, contradicting our supposition.

It follows that $\nu \in S_c$, which proves the result. □

We encountered a concept closely related to stabilization in our earlier discussion of repeatable processes, namely *convergence* (cf. Definition 49). It may be seen as the 'observable' counterpart of stabilization. Note that, if a computation stabilizes at some time $u \geq 0$, then it also converges at that time. The converse holds only in special cases.

Lemma 21. *Let P be observationally deterministic. Then a computation c of P eventually stabilizes if and only if it eventually converges.*

Proof. We only need to show the 'if'-part. Thus, let $c \in C$ be an arbitrary computation, and suppose that c eventually converges. Let $u \in [0, \infty)$ and $e \in E$ be such that $\delta(c(t)) = e$ for all $t \geq u$. Consider any time t with $t \geq u$. As $\delta(c(t) = \delta(c(u))$ by convergence, it follows by the observational determinacy of P that $c^t = c^u$, hence $c(t) = c(u)$. Thus c stabilizes at time u. □

Now recall that observationally deterministic processes are repeatable (cf. Lemma 12) and that, for repeatable processes P, the repeatable function determined by P is denoted by f_P (cf. Definition 50). It leads us to the following further observation.

Corollary 6. *Let P be observationally deterministic. Then for any $e_0 \in E_0$ we have that $f_P(e_0)$ is defined if and only if there is a computation $c \in C$ with $\delta(c^{init}) = e_0$ which eventually stabilizes. In the later case, $f_P(e_0) = \delta(c(u))$ if c stabilizes at time u.*

Proof. This immediately follows from Definition 31 and Lemma 21. Note that, by observational determinacy, if a c exists with the stated property, it is uniquely determined. □

Corollary 6 directly applies the theory in Section 10. To check this, let P be operationally discrete, and let it also be operationally or observationally deterministic. Let P be differentiable, and suppose that it has a step generating process R_P that is observationally deterministic. (This will be the case for many classical machines, as argued in the comments following Theorem 23.) Then the values of g_P can be computed by R_P by watching for stabilization (if possible) rather than convergence of the process.

Knowledge by stabilization Generalizing this, if the stabilization of the computations of a given process were detectable by an observer, it could give us an alternate way to define the computed knowledge of a computation. (Compare Definition 15.)

Definition 56. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. Write $S_P = \bigcup \{E_c \mid c \in C$ and c eventually stabilizes$\}$. Then S_P is said to comprise the knowledge computed by P 'by stabilization'.*

Clearly $S_P \subseteq E_P$. To say more about the two sets, we need the following notation and a definition.

Notation 8 *For any $c \in C$ and time $T \in [0, \infty)$, let c_T be the curve defined by $c_T(t) = c(t)$ for $0 \leq t \leq T$ and $c_T(t) = c(T)$ for $t \geq T$.*

Definition 57. *P is said to be* closed under pre-emption *if for all computations $c \in C$ and times $T \in [0, \infty)$ with $\delta(c(T))$ 'defined', one has that $c_T \in C$.*

One may well argue that all processes should be closed under pre-emption, once the observer, or any operator for that matter, has the capability of 'freezing' them at any observable moment. The following observation is evident.

Proposition 8. *If P is closed under pre-emption, then $S_P \equiv E_P$, i.e. whatever knowledge P computes, it can compute by stabilization.*

Let $I_c = \{t \mid \delta(c(t)) \text{ is defined}\}$ denote the domain of the observable trace of a computation c (cf. Definition 13). If P is closed under pre-emption, then for any computation $c \in C$ the sequence of curves $\{c_T\}_{T \in I_c}$ approximates c and may be said to 'converge' to c. It defines the approximation of C by 'relevant' finite segments.

In general, it will not be 'detectable' by an observer whether a computation of a given process eventually stabilizes. (In the classical theory of computation, it is an example of an 'undecidable' property.) In Subsection 11.5 we will look further into the question what may be 'knowable' about computational processes.

11.2 Loop condensation

Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process and let $c \in C$ be a computation. We called c *loop-free* if $c : [0, \infty) \to A$ is 1-1. The notion proved important when we analyzed the behavioral characteristics of operationally deterministic processes in Subsection 3.5.

If a computation c is not 1-1, then we say that it has a 'loop' or a 'cycle'. If $c(t_1) = c(t_2) = \alpha$ for some times t_1 and t_2 with $t_1 < t_2$, then we say that c has a loop at α. What is the nature of loops in computations? Can one always eliminate them, like in typical machine computations, if the ultimate goal of a computation, whatever it might be, lies beyond the occurring loops?

We first give some general observations. Then we discuss ways in which loops might be 'condensed'. Let $c \in C$.

Definition 58. *For every $\alpha \in A$ and $T \in [0, \infty)$, let $L_c(\alpha, T) = \{t \geq T \mid c(t) = \alpha\}$. Computation c is said to loop on α at time T or later if either $L_c(\alpha, T)$ is finite and $|L_c(\alpha, T)| \geq 2$ or $L_c(\alpha, T)$ is infinite. We say that c has a loop, if it loops on some $\alpha \in A$ at time 0 or later.*

If a computation c loops on $\alpha \in A$, it does not necessarily mean that it loops on α 'forever', i.e. at time T or later for every $T \in [0, \infty)$. In our approach to computation, c may 'cycle' on a same action item for some time and then not return to the same action item ever again afterwards. For the case of operationally deterministic processes, the following observation can be made.

Proposition 9. *Let P be operationally deterministic. Then for every $c \in C$, $\alpha \in A$ and $T \in [0, \infty)$, the set $L_c(\alpha, T)$ is either empty, a singleton, or infinite.*

Proof. Suppose $L_c(\alpha, T)$ contains at least two elements. Suppose $L_c(\alpha, T)$ were finite. Let t_1 and t_2 be the two largest elements of $L(\alpha, T)$, with $t_1 < t_2$. As $c(t_1) = c(t_2) = \alpha$, it follows from the operational determinacy of P that $c^{t_1} = c^{t_2}$. Then $2 \cdot t_2 - t_1 > t_2$ $(> T)$ and $c(2 \cdot t_2 - t_1)) = c(t_2 + (t_2 - t_1)) = c(t_1 + (t_2 - t_1)) = c(t_2) = \alpha$ and hence $2 \cdot t_2 - t_1 \in L_c(\alpha, T)$. This contradicts that t_2 was the largest element of $L_c(\alpha, T)$. Hence $L_c(\alpha, T)$ must be infinite. □

One may observe also that, if $L_c(\alpha, T)$ is infinite, it can have a cardinality up to that of the continuum.

In general computational processes, the behaviour of loops in computations is not different from that of cycles in curves. We need to make a few observations before we can discuss the possibilities for loop condensation.

Lemma 22. *For every $c \in C$, $\alpha \in A$ and $T \in [0, \infty)$, the set $L_c(\alpha, T)$ is (topologically) closed.*

Proof. From topology we know that a set is closed if and only if it contains all of its limit (or accumulation) points. Let $\{t_n\}$ be a sequence of times in $L_c(\alpha, T)$, and assume that $t_n \to u$, for some $u \in [0, \infty)$. Then $u \geq T$, and by continuity of c, $c(u) = c(\lim_n x_n) = \lim_n c(x_n) = \alpha$. Thus $u \in L_c(\alpha, T)$. It follows that $L_c(\alpha, T)$ is closed. □

It follows from Lemma 22 that, for any $T \in [0, \infty)$, if c loops on α at time T or later, then there is an earliest (smallest) time $t \geq T$ such that $c(t) = \alpha$. Then either c remains constant for a while, or forever, or there is an earliest (smallest) time $t > t'$ such that $c(t) = c(t) = \alpha$ again. There is no guarantee that c visits item α again after time t'.

Lemma 23. *Let $c \in C$ and $\alpha \in A$. Assume that $L_c(\alpha, T)$ is bounded, and let $u = \inf\{t \geq T \mid c(t) = \alpha\}$ and $v = \sup\{t \geq T \mid c(t) = \alpha\}$. Then u and v are well-defined and $u, v \in L_c(\alpha, T)$.*

Proof. Because $L_c(\alpha, T)$ is assumed to be bounded, both u and v are well-defined. By Lemma 22, $L_c(\alpha, T)$ is closed and thus we are done, as closed and bounded subsets of $\mathbb{R}$ contain their infimum and supremum. □

Simplifying loops Consider a computation c, and suppose that c has loops. We now consider whether and how c may be altered ('simplified') so it has fewer or simpler loops, in some intuitive sense.

In general, if one wants to change a computation and preserve some local structure of it, one is led to the *homotopy theory* of curves in metric spaces. In our case, the situation is even more complicated, as we also have to keep track of the 'intermediate knowledge build-up' along a curve, which cannot just be modified by a continuous transformation of any kind, without the danger of loosing data or even computationality.

In the present general setting, only a few options seem to exist for the 'removal' or simplification of loops. By Lemmas 22 and 23, one of the following situations can occur if a computation c loops on an item α:

A: there are times $u, v \in [0, \infty)$ with $u \leq v$ such that c visits α for the first time at time u and for the last time at time v (and maybe many more, possibly infinitely many more, times in between),

B: there is a time $u \in [0, \infty)$ such that c visits α for the first time at time u, and for every time $T \geq u$ there is a $t \geq T$ such that c visits α again at time t.

Considering the first possibility, one obvious way to 'condense' the loop at α would be to redefine c into a computation c' with: $c't) = c(t)$ for $0 \leq t \leq u$ and $c'(t) = c(t + v - u)$ for $t \geq u$. A second possibility would be to view 'dwelling' as a way to neutralize a loop, i.e. to redefine c into a computation c' with: $c't) = c(t)$ for $0 \leq t \leq u$ and $t \geq v$, and $c(t) = c(u) = c(v) = \alpha$ for $u \leq t \leq v$. This option is easily modified so it works also for the second looping possibility.

The disadvantage of both approaches is that they potentially (also) eliminate all knowledge c generates in between the visits to α. In the second case, c is even 'straightened' entirely after time u. Clearly, this is no problem if c is known to converge at of before time u.

In general, the condensations are feasible only if c generates no (new) knowledge during the loop that isn't compensated for in some way.

11.3 Structural complexity

In the definition of computational processes, we have left it fully open how these processes actually 'produce' their computations. This gives the theory the extreme generality we want, but at the same time it makes it hard to discuss any 'familial' properties of the computations that are produced by a single process. What sort of properties really bind or bound the computations of a process?

In the preceding sections, we distinguished a number of important properties which computational processes may have, such as determinacy and discreteness. One may say that these properties place severe constraints on the behaviour of computations and on their observability. Nevertheless, we have seen that many processes share these properties as 'behavioral characteristics' of their individual computations. What can we say about the *complexity* of the generated computations?

Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process. In this subsection, we first discuss various aspects of the complexity of C as a 'family' of computations and the impact of determinacy on it. We are especially interested in the phenomenon that many different computations of C can lead to a same action item α in A. The 'computational past' of α consists of many intertwined segments of computations. The following two notions are central to our analysis.

Definition 59. *$P = \langle A, E, \delta, E_0, C \rangle$ is called* operationally finitary *if for all $\alpha \in A$, the set $\{c \in C \mid c \text{ visits } \alpha\}$ is finite.*

Definition 60. *$P = \langle A, E, \delta, E_0, C \rangle$ is called* observationally finitary *if for all $\alpha \in A$ with $\delta(\alpha)$ defined, the set $\{c \in C \mid c \text{ visits } \alpha\}$ is finite.*

Thus, a computational process is called operationally finitary if for every $\alpha \in A$, there are only finitely many computations that lead to α or, stated differently, if there are only finitely many 'starting points' where a computation can begin that leads to α. For observationally finitary processes, this is limited to α's with $\delta(\alpha)$ defined.

If P is operationally finitary, it is also observationally finitary. The converse certainly holds if $\delta(\alpha)$ is at least defined for all $\alpha \in A$ that are ever visited by a computation in C. (In Proposition 6 we showed that under this very same condition, operational and observational determinacy are equivalent.) A slightly more general observation can be made.

Proposition 10. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. Suppose that, for every $\alpha \in A$ visited by a computation of C, there are finitely many items $\beta \in A$ with $\delta(\beta)$ defined such that every computation $c \in C$ that visits α also visits (at least) one of these items β later on. Then, if P is observationally finitary, it is also operationally finitary.*

Proof. Let $\alpha \in A$ be arbitrary. If $\{c \in C \mid c \textit{ visits } \alpha\}$ is empty, we are done. Otherwise we may assume that, by the property of P, there is a finite set $B \subseteq A$ such that $\delta(\beta)$ is defined for every $\beta \in B$ and every computation $c \in C$ that visits α also visits (at least) one of the items $\beta \in B$ later on. It follows that $\{c \in C \mid c \textit{ visits } \alpha\} \subseteq \bigcup_{\beta \in B}\{c \in C \mid c \textit{ visits } \beta\}$. Because P is observationally finitary, $\{c \in C \mid c \textit{ visits } \beta\}$ is finite for every $\beta \in B$, hence $\{c \in C \mid c \textit{ visits } \alpha\}$ is finite too. Thus P is operationally finitary. $\square$

Structural effects The familial consequences of determinacy manifest themselves most emphatically when we consider how computations might 'intersect' in their action space. To see this, let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally deterministic. In Theorem 3 we saw that the computations of P could have one of the following characteristics, which we will now call *types*: loop-free (type I), ultimately constant (type II), and ultimately periodic (type III).

The following observation is easily made. It illustrates the structural effect of determinacy.

Proposition 11. *Let P be operationally deterministic, and let c and c' be two computations of P. If c and c' intersect as curves in A, then they must have the same type.*

Proof. If c and c' intersect, then there are an item $\alpha \in A$ and times t_1 and t_2 such that $c(t_1) = c'(t_2) = \alpha$. By operationally determinacy of P, it follows that c and c' have identical behaviour in time afterwards, thus for $t \to \infty$. This implies that they must be of the same type. $\square$

It follows from the given argument that, if c and c' intersect and are both of type III, then c and c' end up in the *same* non-trivial cycle. If P is observationally deterministic, then the same proposition holds for all c and c' that intersect in an item α with $\delta(\alpha)$ defined.

Let α be any action item in A. If P is operationally deterministic then, by Proposition 11, all computations that ever visit α (at their own time) have the same type and proceed along the very same trajectory after their visit to α. If $\delta(\alpha)$ is defined, the same holds when P is observationally deterministic. But, more can be said when P is also operationally (or, observationally) finitary.

Let P be operationally deterministic. For any $\alpha \in A$, let $C_\alpha = \{c \in C \mid c \textit{ visits } \alpha\}$. It follows from Lemma 22 that for any $c \in C_\alpha$, there is an earliest (or smallest) time $t = t_{c,\alpha}$ such that $c(t) = \alpha$. Let $\widetilde{c_\alpha}$ denote the segment of c

from $t = 0$ up to $t = t_{\alpha,c}$. For all $c \in C_\alpha$, the segments $\widetilde{c_\alpha}$ are loop-free, and compact as subsets of A. Also, if $c, c' \in C_\alpha$, then by operational determinacy we have $c^{t_{c,\alpha}} = (c')^{t_{c',\alpha}}$.

If two computations of C_α meet 'earlier', i.e. if their segments $\widetilde{c_\alpha}$ and $\widetilde{c'_\alpha}$ already meet in, say, item β before they meet in α, then they coincide already from β onward. In fact, the following property holds.

Lemma 24. *If $\widetilde{c_\alpha}$ and $\widetilde{c'_\alpha}$ meet before they meet at α, then there is an earliest (smallest) time $t = t_{\alpha,c,c'} < t_{\alpha,c}$ at which c runs into c'.*

Proof. Suppose $\widetilde{c_\alpha}$ intersects $\widetilde{c'_\alpha}$ in at least one more item, before they meet at α. Let $S = \{t \mid t \in [0, t_{\alpha,c}]$ *and* $c(t)$ lies on c' before c' visits α for the first time$\}$, and let u be the greatest lower bound of S. Certainly $(u, t_{\alpha,c}] \subseteq S$, by operational determinacy of P. If $u \in S$, then $t_{\alpha,c,c'} = u$ and we are done.

Suppose that $u \notin S$. Then there is a decreasing sequence $t_1 > t_2 > \cdots$ of times in S with $u = \lim_i t_i$. Obviously we have $c(u) = \lim_i c(t_i)$, by continuity of c, and it follows that the sequence $c(t_1), c(t_2), \cdots$ is a Cauchy sequence (within segment $\widetilde{c_\alpha}$). We also know that every $c(t_i)$ lies on the curve c' and thus, for every $i \geq 1$ there must be a t'_i such that $c'(t'_i) = c(t_i)$. Hence, the sequence $c'(t'_1), c'(t'_2), \cdots$ is a Cauchy sequence as well, now within the segment $\widetilde{c'_\alpha}$

As $\widetilde{c'_\alpha}$ is compact in A, it is also *complete.* Thus $c'(t'_1), c'(t'_2), \cdots$ must have a limit, say $c'(u')$ within the subset. Because

$$d(c(u), c'(u')) \leq d(c(u), c(t_i)) + d(c(t_i), c'(v')) = d(c(u), c(t_i)) + d(c'(t'_i), c'(v'))$$

for every $i \geq 1$, a straightforward half-ϵ argument shows that $d(c(u), c'(u')) = 0$. Thus $c(u) = c'(u')$, with $u < t_{\alpha,c}$ and, because $c(u) \neq \alpha$, also $u' < t_{\alpha,c'}$. Hence $u \in S$, a contradiction. □

Notation 9 *For $c, c' \in C_\alpha$, let $t_{\alpha,c,c'}$ denote the earliest time t at which c meets c', i.e. in α or an item before that (on c).*

We clearly have that $t_{\alpha,c,c'} = t_{\alpha,c}$ when c does not meet c' before they meet at α. By Lemma 24, $t_{\alpha,c,c'}$ is well-defined.

Now let P be operationally deterministic and operationally finitary. Define the following 'graph' T_α. The set of vertices of T_α consists of all items of A where a computation $c \in C_\alpha$ begins, item α, and all items $c(t_{\alpha,c,c'})$ where c and c' are two computations that meet at α or earlier. The edges of T_α are formed by linking any two vertices that occur consecutively on the curve of some computation $c \in C_\alpha$, except that α is not regarded to have any successors. By the assumptions on P, T_α is a finite and well-defined structure.

Lemma 25. *Let P be operationally deterministic and operationally finitary. Then, for all $\alpha \in A$, T_α is a (directed) finite tree.*

Proof. If C_α is empty, then T_α consists of a single vertex α only and we are done. Thus, let C_α be non-empty. As P is operationally finitary, we know that C_α is finite and, hence, that T_α is finite as well.

Now note that, as a finite graph, T_α is connected. This follows because every vertex of T_α lies on a finite 'path' from that vertex to α, namely the path implied by the computation(s) that lead from this vertex to α.

Next, we note that T_α is also cycle-free. This is straightforward when the intersecting computations are all of type I or all of type II. In case they are all of type III, two case occur. If α does not lie on the cyclic part of a curve, then it does not lie on the cyclic part of any curve and we effectively have an intersection pattern as for type I curves. If α does lie on the cyclic part of any curve, then the vertices of the form $c(t_{\alpha,c,c'})$ lie either on a 'stem' that leads to the cycle or on the cycle itself. In the first case, no cycle can be caused.

In case vertices of the form $c(t_{\alpha,c,c'})$ lie on the cyclic part of the type III computations (note that this cycle is identical for all computations), then we note that all edges connect vertices to immediate successors in a direction towards α and do not pass beyond it. In particular, the edges cannot form a cycle, as α does not have any successor.

Being finite, connected, and cycle-free, proves that T is a tree. □

T_α can be interpreted as the *in-tree* of the computations that lead into α. The paths towards the root represent the trajectories of the computations, and the internal vertices correspond to the action items in A where computations 'meet' for the first time and become confluent, by the operational determinacy. The leaves of T_α are the starting points of the maximal computations that lead to α. All other computations that lead to α are subsumed by the maximal ones, but they are important just the same, by their different starting points. We will make this precise in Theorem 26 below.

If P is (operationally deterministic and) observationally finitary, then one can prove a same result as Lemma 25 for all α with $\delta(\alpha)$ defined. If process P is observationally deterministic in stead of operationally deterministic, then the same arguments can be applied in case P is operationally or observationally discrete. (The main reason for the latter restriction is that it guarantees an analogue of Lemma 22.)

Compactness Let P be operationally deterministic again. We now consider what it means for the generation of knowledge, when P is operationally or observationally finitary. In the analysis, we pursue the intriguing thought that a computation, say c, may be viewed as a *model* for the set E_c of knowledge items that it computes. If we accept this view, then it is natural to ask when a given set of knowledge items has a model, i.e. is generated by a computation of P. Can one prove a kind of *compactness theorem* like in Model Theory?

We first consider sets of action items B with $B \subseteq A$ and ask the similar question. When does B have a model, i.e. when is there a single computation c with $c \in C$ that visits all items of B, in some order?

Lemma 26. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be operationally deterministic and operationally finitary. Let B be a non-empty subset of A with the property that for every two items $\alpha, \beta \in B$, there is a computation $c \in C$ such that c visits both α and β. Then there is a computation $c \in C$ that visits all items of B.*

Proof. Let α be an arbitrary element of B. By assumption, there is a computation $c \in C$ that visits α. If all items of B are visited by c, then we are done. Otherwise there is an item α' of B that is not visited by c.

By assumption on B, there must be a computation $c' \in C$ such that α' and α are both visited by c'. As P is operationally deterministic and α' does not lie on c, it means that α' is visited before α and hence, that $c' \in C_\alpha$. By determinism, all items that were visited by c before, are visited by c'.

Now repeat the argument for α'. If all items of B are visited by c', then we are done. Otherwise there is an item $\alpha'' \in B$ that is not visited by c'. By assumption on B, there must be a computation $c'' \in C$ such that α'' and α' are both visited by c''. Here c'' is necessarily different from c and c' and, as before, it also means that α'' is visited by c'' before α'. It follows that $c'' \in C_{\alpha'}$ and thus, by determinism, that $c'' \in C_\alpha$. Also, all items that were visited by c' before, are now visited by c''.

In every repetition of the argument, either a computation is found that visits all items of B or a new computation of C_α is uncovered that visits a larger part of B. As P is operationally finitary and thus C_α is finite, this repetition can continue at most finitely many times. Upon termination a computation of C_α, and thus of C, is found that, necessarily, visits all items of the set B. $\square$

The following variant of Lemma 26 can be shown as well, by only a minor modification of the proof.

Lemma 27. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be observationally deterministic and observationally finitary. Let B be a non-empty subset of A with the property that for every two items $\alpha, \beta \in B$, there is a computation $c \in C$ such that c visits both α and β. Then there is a computation $c \in C$ that visits all items of B.*

Proof. The same argument as in the proof of Lemma 26 applies, because all items $\alpha, \alpha', \cdots$ in the proof now have defined δ-values. In this case, we only need that P is observationally deterministic and observationally finitary to derive the same conclusion. $\square$

We can now address the problem of characterizing when a set of knowledge items has a 'model'. The assumptions on P are modified to a form that is appropriate, and applicable, to the case of knowledge sets.

Theorem 24. *Let $P = \langle A, E, \delta, E_0, C\rangle$ be observationally deterministic and observationally finitary. Let F be a non-empty subset of E with the property that for every two items $\chi, \psi \in F$ there is a computation $c \in C$ such that c computes both χ and ψ. Then there is a computation $c \in C$ such that c computes all items of F.*

Proof. It follows from the assumption of F that all its items χ must be computable. As P is observationally deterministic, Proposition 5 implies that for every $\chi \in F$ there is a unique $\alpha_\chi \in A$ with the property that $\delta(\alpha_\chi) = \chi$. Let $B = \{\alpha_\chi \mid \chi \in F\}$.

Clearly B is non-empty. Consider any two items $\alpha_\chi, \alpha_\psi \in B$. Because χ and ψ are items of F, there must be a computation $c \in C$ that computes both χ

and ψ. By observational determinacy, this means that c must visit both α_χ and α_ψ. This means that B satisfies the requirements of Lemma 27. The result now follows directly by applying the very Lemma. $\square$

Maximality Finally, we briefly sketch a different perspective on Lemma 26 and its consequences for sets of computations. As before, let $P = \langle A, E, \delta, E_0, C\rangle$ be a computational process. Define the binary relation $\leq_A$ on A as follows.

Definition 61. *For any $\alpha, \beta \in A$, let $\alpha \leq_A \beta$ if either $\alpha = \beta$, or there is a computation $c \in C$ such that c visits α and β and α before β.*

Proposition 12. *Let P be operationally deterministic. Then $\leq_A$ is a quasi-order on A.*

Proof. By definition, $\leq_A$ is reflexive. In order to prove transitivity, let $\alpha, \beta, \gamma \in A$ be such that $\alpha \leq_A \beta$ and $\beta \leq_A \gamma$. Without loss of generality, we may assume that α, β, and γ are all distinct.

As $\alpha \leq_A \beta$ and $\beta \leq_A \gamma$, there are computations c and c' such that c visits α and then β, and c' visits β and then γ. By operational determinism, c follows the same trajectory as c' after the visit to β and, by Proposition 11, both have the same type. If both c and c' are of type I, then clearly c visits γ after it visits α (and β) on its trajectory and we have $\alpha \leq_A \gamma$. This is easily argued also in case both curves are of type II.

Suppose c and c' are both of type III. Let t_1, t_2, t_3, t_4 with $t_1 < t_2$ and $t_3 < t_4$ be such that $c(t_1) = \alpha$, $c(t_2) = \beta$, $c'(t_3) = \beta$ and $c'(t_4) = \gamma$. If $t_2 \leq t_3$ then, clearly, the curve of c extends to γ and we have $\alpha \leq_A \gamma$ again. In case $t_2 > t_3$, we have that $c(t_2) = c'(t_3)$ and thus, by operational determinism, that $c(t_2 + t) = c'(t_3 + t)$, for all $t \geq 0$. Taking $t = t_4 - t_3$ (> 0), we obtain $c(t_2 + t_4 - t_3) = c'(t_3 + t_4 - t_3) = c'(t_4) = \gamma$ and thus, again, c visits γ after it visits α (and β) and we have $\alpha \leq_A \gamma$. $\square$

We now consider the quasi-order $\leq_A$ on A in more detail. Note that a subset B of A is a *chain* if B is non-empty and for every $\alpha, \beta \in B$ we have $\alpha \leq_A \beta$ or $\beta \leq_A \alpha$ (or both).

Theorem 25. *Let P be operationally deterministic and operationally finitary. Then every chain in A has a lowerbound.*

Proof. Let B be a chain in A according to $\leq_A$. The condition that for all $\alpha, \beta \in B$ we have $\alpha \leq_A \beta$ or $\beta \leq_A \alpha$ is equivalent to the condition that for all $\alpha, \beta \in B$ there is a computation c such that c visits both α and β. By Lemma 26 we conclude that there is a single computation $c \in C$ such that c visits all items of B and, consequently, that $c^{init} \leq_A \alpha$ for all $\alpha \in B$. Thus c^{init} is a lowerbound of B. $\square$

It may be noted that Theorem 25 is essentially *equivalent* to Lemma 26. Thus, the condition that P be operationally finitary appears as a natural structural requirement on the set of computations of P.

Clearly, if P is operationally deterministic and for some computation $c \in C$ and $\alpha \in A$ we have $c^{init} \leq_A \alpha$, then c visits α. If $d \in C$, then $c^{init} \leq_A d^{init}$ if and only if d is a subcomputation of c. This leads to the following concept.

Definition 62. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be operationally deterministic. A computation $c \in C$ is said to be* maximal *if for all computations $d \in C$ such that c is a subcomputation of d, d is also a subcomputation of c.*

Theorem 26. *Let $P = \langle A, E, \delta, E_0, C \rangle$ be operationally deterministic and operationally finitary. Then every computation $c \in C$ is a subcomputation of a maximal one.*

Proof. Let $c \in C$ be given. An immediate proof of the theorem is obtained by considering the graph $T_{c^{init}}$. By Lemma 25, $T_{c^{init}}$ is a finite tree with root α. Let $d \in C$ be a computation whose d^{init} is a leaf of $T_{c^{init}}$. Then d is maximal, and it clearly subsumes c.

For a direct proof one may argue as follows. If c is maximal, then we are done. Otherwise, there must be a computation $d_1 \not\equiv c$ such that $d_1^{init} \leq_A c^{init}$ but $c^{init} \not\leq_A d_1^{init}$. More generally, suppose that for some $i \geq 1$ there are computations $d_1, \cdots, d_i$ such that $c, d_1, \cdots, d_i$ are all different, $d_i^{init} \leq_A \cdots \leq_A d_1^{init} \leq_A c^{init}$ and $c^{init} \not\leq d_1^{init} \not\leq_A \cdots \not\leq_A d_i^{init}$.

Arguing inductively, if d_i is maximal, we are done. Otherwise, there must be a computation d_{i+1} different from $c, d_1, \cdots, d_i$ such that $d_{i+1}^{init} \leq_A d_i^{init}$ but $d_i^{init} \not\leq_A d_{i+1}^{init}$. Note that $c, d_1, \cdots d_{i+1}$ all belong to $\{d \in C \mid d \textit{ visits } c^{init}\}$.

As P is operationally finitary, the latter set is finite. As a consequence, the inductive argument must terminate after finitely many iterations. Upon termination, a computation d is obtained that is both maximal and subsumes the given computation c. □

We omit a discussion of the results in the 'infinitary' case, i.e. when C_α is countable or even uncountable, for some or all $\alpha \in A$.

11.4 Computational complexity

Consider computations as they are generated by computational processes P, in our framework. It is at least intuitive that some computations generate the 'knowledge' we want from them more easily or quickly than other computations do. This raises the question whether there is any evidence of *computational complexity* that might be observable, without knowing precisely how the computations are generated. Are there ways of defining complexity in topological terms, at the very general level of our theory? And, what precisely are we defining the complexity of?

Let $P = \langle A, E, \delta, E_0, C \rangle$ be a computational process, and let e be a *computable* knowledge item of E, i.e. for which there is at least one computation $c \in C$ that visits an item $\alpha \in A$ with $\delta(\alpha) = e$. The complexity of computing e will depend on the 'complexity' of the segment of c that leads up to α, possibly minimized over all computations $c \in C$ and items $\alpha \in A$ with $\delta(\alpha) = e$ that they visit. One simple measure would be the *earliest moment* $t \in [0, \infty)$ such that $\delta(c(t)) = e$, for some $c \in C$.

Quantified over a suitable class of allowable processes, the time measure gives us a well-defined notion of *time complexity* for knowledge items, very similar to the notion as we know it in classical computability theory [24]. In order to say more, independent of the 'complexity' of the generative machinery 'inside' any of the processes P, whatever this may mean, we need some sort of measure for the (segments of the) curves that define computations. In this subsection we develop a number of concepts towards this goal, to show that this can be done in the present framework.

General properties First we argue that the framework leads to some general properties that should hold for all computational processes. Let P be a process, and let $c : [0, \infty) \to A$ be a computation of P. We recall notation 8.

Notation 10 *For any $T \in [0, \infty)$, let $c_T : [0, T] \to A$ be the* segment *of c up to time T, and let $Ran\{c_T\}$ be the* range *of c_T in A.*

Thus, for any T, the map c_T is the curve of c as it unfolds from time 0 up to time T, and $Ran\{c_T\}$ is the set of action items visited by c during this time period.

We first observe that $c_T : [0, T] \to A$ is a continuous function. In fact, by the Heine-Cantor theorem for metric spaces, we may assert that c_T is even uniformly continuous on $[0, T]$. A first simple consequence is that the range of c_T in A is a compact, thus dense, subset of A. It gives us the following property, almost by definition.

Lemma 28. *For any $T > 0$, $Ran\{c_T\}$ has the Bolzano-Weierstraß property, i.e. every infinite subset of $Ran\{c_T\}$ has at least one accumulation point in $Ran\{c_T\}$.*

Proof. As $[0, T]$ is compact in $\mathbb{R}$, its image under c_T is compact in A. It is well-known from topology that compact spaces are Bolzano-Weierstraß spaces. This gives the result. □

The compactness of $Ran\{c_T\}$ also implies that it is a bounded subset of A. It leads to the following observation, for all computations. Recall that d was the metric of A.

Lemma 29. *For every $c \in C$ and $T > 0$, there is a constant $b_{c,T}$ such that for all t_1, t_2 with $0 \leq t_1 \leq t_2 \leq L$ one has: $d(c(t_1), c(t_2)) \leq b_{c,T}$.*

Proof. We argued that $Ran\{c_T\}$ is compact in A. As compact subsets of metric spaces are bounded, the result follows. □

Interpretation Lemma 29 has an important philosophical consequence, if we accept the premise that distance between action items relates to computational effort. After all, the lemma implies that, if α and β are two action items on the path of a computation and α is visited before β, then β has only finite distance to α. Hence, if α and β would be represented by symbolic complexes of some kind, with a Hamming-type of distance metric on these complexes, then the representation of β would be obtained after *finitely modifying* the representation of α.

This fact was one of the basic intuitions when we defined computations the way we did to begin with. The intuition is now confirmed by the formal framework, without any special assumption on P except for the symbolic property of A. Note that it also 'proves' the *bounded exploration postulate* in the case of Abstract State Machines [5], in our framework.

We can make one further observation. In Theorem 20 we proved that the computations of operationally discrete processes P that are also operationally or observationally deterministic, can be subdivided into steps that exactly cover the action of P from one observable action item to the next. (This was well-defined.) By Lemma 29, the items visited during any computation have a finite distance to each other, and this holds for all consecutive *observable* items in particular. Assume again that A consists purely of items that are symbolically represented, with a metric corresponding to the symbolic distance between items. We now conclude that every step basically effectuates the finitely many modifications, whatever this amount to, to get from one symbolic complex to the next in a computation, seen through the 'lens' of the observer.

Finite complexity We have seen that 'finite time' amounts to 'finite distance' of the items that are visited in a computation. However, considering Lemma 29 again, note that there 'distance' is measured *in the image space* of computation c, and this is not necessarily the same as the *arc length*, i.e. the 'distance' as measured *along its curve*. It is not even evident that the latter is finite if the former is.

The definition of computation suggests that the 'arc length' of c_T as a function of T is a better, more telling measure for the computational difficulty of c, and thus for the complexity of generating knowledge by c, than just simple metric distance. In order to say more, we need a few more details from the general theory of curves.

Based on the general definition for curves, the *length* of a segment c_T of c can be defined as follows:

$$L(c_T) = \sup_n \Sigma_{i=1}^{n} d(t_{i-1}, d_t)$$

where the supremum is taken over all $n \in \mathbb{N}$ and all sequences $0 = t_1 \leq \cdots \leq t_n = T$ in $[0, T]$. Segments c_T do not necessarily have finite length. When a segment c_T has finite length, it is called *rectifiable*. The theory of rectifiable curves is well-established in metric geometry [11].

It is easily seen that for all $c \in C$, if C_T is rectifiable, then so is every segment $c_{T'}$ with $0 \leq T' \leq T$. This motivates the following definition for the complexity of computations and computational processes, in our framework.

Definition 63. *A computation c is said to have* finite complexity *if c_T is rectifiable for every $T \in [0, \infty)$. A computational process P is said to have* finite complexity *if each computation of P has finite complexity.*

Various results from the theory of rectifiable curves can be brought to bear on the theory of computation. For example, the length of rectifiable curves is known to be a 'well-behaved' function of the curve parameter [11]. For computations, this amounts to the following observation.

Lemma 30. *For all computations c of finite complexity, $L(c_T) : [0, \infty) \to [0, \infty)$ is monotone and continuous in T.*

We may note that many instances of classical state-based models of computation are essentially computational processes of finite complexity. This follows because, in Section 9, we showed that these models can all be viewed as faithful projections of computational processes with computations that are piecewise linear curves. Moreover, the curves have only one bend in between every two points of the discrete point set corresponding to the machine states. Here, 'finite complexity' becomes equivalent to 'finite time' again.

Finally, we note the following examples of general classes of processes of finite complexity.

Lemma 31. *If a computational process is Lipschitz, then the process has finite complexity.*

Proof. It is well-known from metric topology that every curve segment that is Lipschitz, is also rectifiable. The result follows. □

Also, if $A \equiv \mathbb{R}^k$ for some $k \geq 1$ and computation $c : [0, \infty) \to A$ is differentiable as a curve, then known theory implies that c has finite complexity. In this case $L(c_T)$ is even monotone and differentiable.

It is reasonable to assert that *computational processes should be of finite complexity*, if the notion of complexity is to be quantifiable for them.

Measuring computations Let c be a computation, and assume that it has finite complexity. We argued that, in this case, the function $L(c_T)$ is a sensible indicator of the 'complexity' of c. However, the computational effort that goes into producing c may depend in a much more complex way on T, especially for $T \to \infty$.

Already, if we would account for computational effort by charging 'by length' along the curve c, then one would end up measuring complexity as $F(L(c_T))$. where $F : [0, \infty) \to [0, \infty)$ is some monotone and continuous function. F could just add a linear factor to the arc length of c, or accumulate larger amounts for the actual effort or energy dispensed by P over time. Even if one imposes only minor requirements on F, this may lead to undue limitations.

For example, suppose one would want to express that the computational effort over a distance of $x + y$ along the curve of c is never more than that over distances x and y summed together, possibly up to some multiplicative factor. This leads to a *Cauchy-type functional inequality* $F(x + y) \leq K(F(x) + F(y))$, for some constant $K \geq 1$ and measured distances x and y along the curve. Then the following, elementary observation can be made.

Lemma 32. *Let $F : [0, \infty) \to [0, \infty)$ be continuous. Suppose that, for some constant $K \geq 1$, F satisfies the inequality $F(x + y) \leq K(F(x) + F(y))$, for all $x, y \in [0, \infty)$. Then F is polynomially bounded.*

Proof. For all $x \in [0, 2]$ we have $F(x) \leq M$, where $M = \max_{x \in [0,2]} F(x)$. (As F is continuous, M is well-defined.) Now consider any x with $x > 2$. Let $m \geq 1$ be such that $2^m < x \leq 2^{m+1}$. Define 2^m reals $x_0, \cdots, x_{2^m-1}$ by $x_j = \frac{x}{2^m}$. Observe that $x_j \in [0, 2]$ for $0 \leq j \leq 2^m - 1$, and that $F(x) = F(x_0 + \cdots + x_{2^m-1}) \leq K(F(x_0 + \cdots + x_{2^{m-1}-1}) + F(x_{2^{m-1}} + \cdots + x_{2^m-1})) = 2KF(x_0 + \cdots + x_{2^{m-1}-1})$. Then, by induction and noting that $F(x_0) \leq M$, it follows that

$$F(x) \leq (2K)^m \cdot F(x_0) \leq M \cdot (2^m)^{^2\log 2K} \leq M \cdot x^{1+{^2\log K}}$$

Hence, $F(x) \leq M + M \cdot x^{1+2\log K}$ for all $x \in [0, \infty)$, proving that F is polynomially bounded. □

The Lemma shows that, if one accepts its premises, then measuring the computational effort by $F(L(c_T))$ yields at most a polynomial mark-up of the arc length and leaves the measures 'polynomially related'. Note that, for any $k, r \geq 0$, $F(x) = kx^r$ has the property that $F(x + y) \leq 2^r(F(x) + F(y))$ for all $x, y \in [0, \infty)$ and thus satisfies the requirements of the Lemma.

Given the concept of computations as curves, many more notions of complexity could be imagined. Further options arise if items in A are symbolic complexes of some kind and their representations become measurable too. This leads to considerations of representational or space complexity, depending on the characteristics of A.

We leave it as an interesting project to analyze the general notions of complexity in our framework and elaborate on it in detail.

11.5 When are properties knowable

In the actor-spectator approach we have focused on computations as they are, i.e. on the curves of successive action as interpreted by the observer. The semantic map enabled an observer to extract the computed information from them. This gave us a rich perspective on the essence of computation.

In the course of our analysis, we saw that processes and computations could exhibit a variety of behaviours of interest. Examples include operational and observational determinacy, operational and observational discreteness, stabilization, convergence and more. The evident question is whether these properties can be observed, or even detected, with the appropriate means. Informally, the properties that can, will be called *knowable* (by the observer).

We have not assumed any frame of reference for an observer that would enable us to discuss which properties of processes or computations might be knowable, and which might not be. One might even want to distinguish between various ways of knowing a property, e.g. between knowing it 'up front' or recognizing whether it is violated at any moment in time. To say more, one needs to formalize precisely what it means for a property to be knowable for a certain observer.

The analogous question in classical computability theory would be to ask whether a property is *decidable* or not. This type of question has a long history and is well-formalized, e.g. by using Turing machines as the underlying machine model [14]. In Subsection 9.4 we argued that all classical machine models could be obtained as projections of operationally discrete computational processes. Applying this to classical Turing machines, decidability questions for them have appreciable counterparts for the corresponding computational processes. It is reasonable to say that, in this case, decidable properties will be knowable, depending on the powers of the observer. The converse depends on this as well.

Expanding on the latter, many general properties of processes or computations are likely to be unknowable, just like many properties of classical Turing machines and computable functions are undecidable. An example would be stabilization, similar in character to the classical Halting Problem [41]. We leave it as an interesting problem to develop a theory that would allows us to treat knowable properties, in general or e.g. for the class of functional processes analyzed in Section 10 in particular.

12 Some conclusions

In this report we considered the intricate question of characterizing the notion of computation, in the broad sense in which it is understood today. The report expands on the approach in [45].

The notion of computation we use is based an the epistemic philosophy which we have developed in a number of papers since 2013 [48–50]. In this approach, computations are seen as 'acts' of knowledge generation, through the lens of an observer. This characterizes computation in a way that is model-, algorithm- and representation free and no longer bound to the realm of classical computers, in the classical sense.

Computational processes In order to ground computations properly, we have developed a new and novel concept of computational processes. The idea for it derives from a remark by Turing to the effect that *'all machinery can be regarded as continuous'*. Following up on it, we have defined computations as curves in a suitable metric space, with constraints to guarantee that they generate knowledge properly. The definition enabled us to study computation purely, i.e. as a mathematical construct. The formal theory of computational processes is a major result of this report and a first step towards achieving a truly general theory of 'computation'.

As we are ultimately interested in what knowledge computations can bring about, we did *not* delve into any detail of how a process produces the computations that we observe. All underlying machinery is supposed to be hidden in the definition of the action- and knowledge spaces and in the constraints they impose. Thus, the essence of how a process operates, in any mode or context, is assumed to be implicit in the definition of the action items and of how they are 'strung together' by the curves into computations.

Viewing computations as generating knowledge, gives a possible approach to defining (the emergence of) understanding. From a general perspective, understanding is the ability to reveal all causes and objectives of a given subject, as described by a given piece of knowledge. But, understanding a knowledge item would mean that one should be able to find a 'computation' that leads up to it. Thus, understanding can, potentially, be defined in the framework of computational processes as we developed it.

Computation The definition of computation by means of curves then becomes a highly natural one, expressing the intuition that computation is continuous and 'meaningful'. We have seen that it subsumes all current views of computation based on machine models.

We did not assume anything about computational processes besides the very fact that they can generate computations that fulfill the consistency conditions. We did not even assume any explicit connection between the computations of a given process, although we saw that in the case of deterministic, operationally discrete processes a notion of 'program' emerged more or less naturally. Any underlying mechanism is left fully implicit and abstract.

The philosophy of computation we developed, enabled us to concentrate on *what* is really happening dynamically, from the viewpoint of an observer. Moreover, the approach enabled us to resolve Turing's dichotomy and link continuous and discrete computation, as we showed that the latter is merely a projection of the former. In addition, the approach generalizes to notions of computation that need not have a (known) physical realization, such as computations over the reals.

Our ideas provide not only a novel look at computation. The tacit hypothesis that any physically describable process may be viewed as computational, under suitable preconditions, can be recognized in physics, biology and in other sciences. Indeed, in most definitions, a process is viewed as a continuous action, an operation, or a series of 'bounded effect changes' undertaken in order to achieve a desired result. From a philosophical viewpoint this may be seen as a *teleological* definition. It is adhered to by the notions in this report.

We expect to expand on these and other notions in further studies on the epistemic approach to computation.

Some open problems In elaborating on the definitions, we were able to touch on many issues that are in some way characteristic for computations and computational processes. Many more issues remain to be investigated. This is even more so, if one would be willing to sacrifice some of the 'perfect generality' of our philosophy, e.g. by choosing $A \equiv \mathbb{R}^k$ (some $k \geq 1$).

We conclude with a number of typical open questions that arose in the course of our analysis.

- *Action spaces* The only assumption we made for action spaces A was that they are metric. This proved adequate 'for all practical purposes', but it would be interesting to vary on this assumption. For example, one could generalize and let A be a general topological space, or specialize the theory by assuming A to be a finite-dimensional coordinate space, computations to be bound to a manifold of some kind, and the computations of a process to be collectively parameterized. This is all to remain focused on the understanding of computation as a phenomenon, and is not intended to make metric geometry computational.

- *Knowledge spaces* In the present approach, the consistency conditions merely make sure that the knowledge generation that takes place in the course of a computation c remains consistent with the inference rules of the knowledge theory E of the observer. This, again, proved adequate for our purposes, but it may well be realistic to insist on a closer tie between the progress as made by c and the number of steps in the proof system of E. For example, one may well insist that, for all $t_1 < t_2$ with $\delta(c(t_1))$ and $\delta(c(t_2))$ defined, the minimum proof length needed to ascertain $\delta(c(t_1)) \models^* \delta(c(t_2))$ is upper bounded by an increasing function of the arc length from $c(t_1)$ to $c(t_1)$ along c as a curve, if this is defined.

- *Determinism* In our study of computational processes, the notions of operational and observational determinism appeared naturally. We could characterize the computations of both operationally and observationally deterministic processes and both notions played an important role in the analysis of operationally discrete processes. This leaves the question what more can be said about processes that have these attractive properties. Also, can their 'non-determinism' be curtailed in a sensible way, to give a wide class of processes with attractive properties?

- *Simulation* In this report we explored many properties of computational processes that are of interest from a computational viewpoint. We saw that various types of processes could be obtained as a homomorphic image or by means of another type of mapping from processes with other, more agreeable properties (cf. Sections 6 and 9). Notably, we showed that discrete processes are 'projections' of general computational processes. Can these results be shown in an overriding theory of process simulation, for the present framework? What effect can one obtain from bi-simulation, in case computational processes are used for the generation of knowledge?

- *Repeatable functions* In Section 10 we saw how the effect of repeatable processes is formalized in the notion of repeatable functions. It was left open to develop a 'generic theory of repeatable functions', analogous to but presumably much more general than the classical theory of computable functions. The challenge would be to stay within the current framework of computations and their processes, emphasizing only the functional properties that play a role. What can be said about their 'programmability'?

- *Complexity* In Section 11 we made a first attempt to define notions of computational complexity in our framework. We pointed at the connection to the theory of rectilinear curves, but did not digress on the alternative ways in which computations may be measured. Also, we did not digress on how the various notions can be used to classify knowledge items by their computational complexity. Can one develop a sensible theory of, say, polynomial-time computability, in the present framework?

- *Computational processes* We gave a very general definition of computational processes, as a crucial part of the machine-independent theory of computations we developed. We did not digress on computational processes themselves as mathematical objects. What can be said about operations like composition and interaction of processes, in the present framework?

The philosophy of computations, and the mathematical formulation of it as outlined in this report, lead to many more challenging questions. For example, at the most general level it allows us, essentially, to view any natural or artificial process of which we understand its operations as being computational.

This is consistent with the views of Deutsch [16] who proclaimed, even more pointedly, that *'computations are physical processes and every physical process can be regarded as computation'*. However, as he rightly points out, this is also where the interesting questions about the nature of computations begin. The approach in the report gives a framework to address them.

References

1. A.V. Aho, Computation and computational thinking, *The Computer Journal* 55:7 (2012) 832-835.
2. M.A. Arbib, Automata theory and control theory - a rapprochement, *Automatica* 3 (1966) 161-189.
3. M.A. Arbib, Tolerance automata, *Kybernetika* 3 (1967) 223-233. Also in: R.E. Kalman, P.L. Falb, M.A. Arbib, *Topics in Mathematical System Theory*, McGraw-Hill, New York, 1968, Chapter 6.4, pp. 179-184
4. L.W. Beck, *The actor and the spectator - Foundations of the theory of human action*, Yale Univ press, 1975 (reprinted: Key Texts, Thoemmes Press, 1998).
5. A. Blass, Y. Gurevich, Algorithms: A quest for absolute definitions, *Bulletin EATCS* 81, 2003, pp. 195-225.
6. M. Blum, A machine-independent theory of the complexity of recursive functions. *J.ACM* 14:2 (1967) 322-336.
7. L. Blum, F. Cucker, M. Shub, S. Smale, *Complexity and real computation*, Springer, New York, 1998.
8. O. Bournez, M. Cosnard, On the computational power of dynamical systems and hybrid systems, *Theor. Comp. Sci.* 168 (1996) 417-459.
9. O. Bournez, M. Lameiras Campagnolo, D.S. Graça, E. Hainry, Polynomial differential equations compute all real computable functions on computable compact intervals, *Journal of Complexity* 23i (2007) 317-335.
10. M.E. Bratman, *Intentions, plans, and practical reasoning*, Harvard University Press, Cambridge (MA), 1987.
11. D. Burago, Y. Burago, S. Ivanov, *A course in metric geometry*, Graduate Studies in Mathematics, Volume 33, AMS, Providence RI, 2001.
12. R.L. Chrisley, Why everything doesn't realize every computation, *Minds and Machines* 4 (1995) 403-420.
13. R.F. Cohen, P. Eades, T. Lin, F. Ruskey, Three-dimensional graph drawing, in: R. Tamassia, I.G. Tollis (Eds),*Graph Drawing*, DIMACS Int. Workshop, GD '94, Lecture Notes in Computer Science 894, Springer, pp. 1-11.
14. M. Davis (Ed.), *The undecidable: basic papers on undecidable propositions, unsolvable problems and computable functions*, Raven Press Books, Hewlett NY, 1965, reprinted: Dover Books, 2004.
15. P.J. Denning, What is computation?, Editor's introduction, ACM Ubiquity Symposium *What is Computation* , Ubiquity, October 2010. See also: *Computer J.* 55:7 (2012) 805-810.
16. D.E. Deutsch, What is computation?(How) does nature compute?, talk, 2008 Midwest NKS Conference, Bloomington IN, transcript by A. German, 2008, `https://www.cs.indiana.edu/~dgerman/hector/deutsch.pdf`.
17. D.J. Frailey, Computation is process, in: ACM Ubiquity Symposium *What is Computation?*, Ubiquity, November 2010, pp. 1-6. See also: *Computer J.* 55:7 (2012) 817-819.
18. H. Friedman, R. Mansfield, Algorithmic procedures, *Trans. Amer. Math. Soc.* 332:1 (1992) 297-312.
19. J-L. Giavitto, O. Michel, Data structure as topological spaces, in: C.S. Calude et al. (Eds.), *Unconventional Models of Computation*, Third Int. Conference, UMC 2002, Lecture Notes in Computer Science 2509, Springer, 2002, pp. 137-150.
20. Y. Gurevich, Sequential abstract state machines capture sequential algorithms, *ACM Trans. Comput. Logic* 1 (2000) 77-111.
21. Y. Gurevich, Foundational analyses of computation, in: S.B. Cooper, A. Dawar, B. Löwe (Eds.), *How the World Computes*, Proc. CiE 2012, Lecture Notes in Computer Science 7318, Springer, 2012, pp. 264-275.

22. C. Horsman, S. Stepney, R.C. Wagner, V. Kendon, When does a physical system compute?, Proc. Royal Soc. A 470 (2169), 20140182, 2014.
23. C. Horsman, V. Kendon, S. Stepney, J.P.W. Young, Abstraction and representation in living organisms: when does a biological system compute? In: G. Dodig-Crnkovic, R. Giovagnoli (Eds), *Representation and Reality in Humans, Other Living Organisms and Intelligent Machines*, SAPERE Series Vol. 28, Springer, 2017, pp. 91-116.
24. J.E. Hopcroft, J.D. Ullman, *Formal languages and their relation to automata*, Addison-Wesley Publishing Company, Reading, MA, 1968.
25. B.J. MacLennan. Analog computation, in: R.A. Meyers (Ed.), *Encyclopedia of Complexity and Systems Science*, Springer-Verlag. New York, 2009, pp. 271-294.
26. A. Mazurkiewicz, Concurrent program schemes and their interpretation, Techn. Rep. PB-17, DAIMI, Datalogisk Afdeling, Aarhus University, Aarhus, 1977.
27. G.H. Mealy, A method for synthesizing sequential circuits, *Bell System Tech. J.* 34 (1955) 1045-1079.
28. A.N. Michel, K. Wang, B. Hu, *Qualitative Theory of Dynamical Systems: The Role of Stability Preserving Mappings*, 2nd Ed., M. Dekker Inc, New York, 2001.
29. E.F. Moore, Gedanken-experiments on sequential machines, in: C. E. Shannon and J. McCarthy (Eds), *Automata Studies*, Annals of Mathematics studies no. 34, Princeton University Press, Princeton, 1956, pp. 129-153.
30. J.R. Munkres, *Topology*, 2nd Edition, Prentice Hall Inc, Upper Saddle River, NJ, 2000 (re-published: Pearson).
31. J. Numrich, A metric space for computer programs and the principle of computational least action, *J. Supercomputing* 43 (2008) 281-298.
32. G. Piccinini, Functionalism, computationalism, and mental states, *Stud. Hist. Phil. Sci.* 35 (2004) 811833.
33. G. Piccinini, Computational modelling vs computational explanation: Is everything a Turing machine, and does it matter to the philosophy of mind?, *Australasian Journal of Philosophy* 85:1 (2007) 93115.
34. G. Piccinini, Computation without representation, *Philosophical Studies* 137.2 (2008) 205241.
35. A. Platzer, Analog and hybrid computation: Dynamical systems and programming languages, —*EATCS Bull.* 114, 2014, 49 pages.
36. H. Rogers Jr., An example in mathematical logic, *The American Mathematical Monthly* 70:9 (1963)929-945.
37. H.T. Siegelmann, S. Fishman, Analog computation with dynamical systems, *Physica D* 120 (1998) 214-235.
38. S. Stepney, Nonclassical computation - A dynamical systems perspective. In: G. Rozenberg et al. (Eds.), *Handbook of Natural Computing*, Springer-Verlag, Berlin Heidelberg, 2012, Ch 59, pp. 1979-2025.
39. A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific J. Math.* 5 (1955) 285-309.
40. D. Tong, The unquantum quantum, *Scientific American* 307: December (2012), 46-49.
41. A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math Soc.* 42:1 (publ. 1937) 244-265. Correction: *ibid.*, *Proc. London Math. Soc.* 43:6 (publ. 1938) 544546.
42. A.M. Turing, Intelligent machinery, A report by A.M. Turing, *National Physical laboratory*, Teddington, UK, 1948.
43. L.G. Valiant, A bridging model for parallel computations, *C.ACM* 33:8 (1990) 103-111.
44. J. van Leeuwen, On Floridi's method of levels of abstraction, *Minds and Machines* 24:1 (2014) 5-17.
45. J. van Leeuwen, J. Wiedermann, Knowledge, representation, and the dynamics of computation. In: G. Dodig-Crnkovic, R. Giovagnoli (Eds), *Representation and Reality in Humans, Other Living Organisms and Intelligent Machines*, SAPERE Series Vol. 28, Springer, 2017, pp. 69-89.
46. Wikibooks, *Topology*, `http://en.wikibooks.org/wiki/Topology`.
47. J. Wiedermann, J. van Leeuwen, How we think of computing today. In: A. Beckmann, C. Dimitracopoulos, and B. Löwe (Eds.), *Logic and Theory of Algorithms*, 4th Conference on Computability in Europe (CiE 2008), Proceedings, Lecture Notes in Computer Science, Vol. 5028, Springer-Verlag, Berlin, 2008, pp. 579-593.

48. J. Wiedermann, J. van Leeuwen, Rethinking computations. In: M. Brown and Y. Erden (Eds.), *6th AISB Symp. on Computing and Philosophy: The Scandal of Computation - What is Computation?*, AISB Convention 2013 (Exeter), Proceedings, AISB, 2013, pp. 6-10.
49. J. Wiedermann, J. van Leeuwen, Computation as knowledge generation, with application to the observer-relativity problem. In: M. Brown and Y. Erden (Eds), *7th AISB Symp. on Computing and Philosophy*, AISB 50 Convention 2014 (London), Proceedings, AISB, 2014, pp. 1-8.
50. J. Wiedermann, J. van Leeuwen, What is computation: an epistemic approach. In: G.F. Italiano et al. (Eds.), *SOFSEM 2015: Theory and Practice of Computer Science*, Lecture Notes in Computer Science Vol 8939, Springer-Verlag, Berlin, 2015, pp. 1-13.
51. J. Wiedermann, J. van Leeuwen, Epistemic computation and artificial intelligence. In: V.C. Mller (Ed.), *Philosophy and Theory of Artificial Intelligence 2017*, SAPERE Series Vol. 44, Springer-Verlag, 2018, pp. 215-224.
52. E.C. Zeeman, The topology of the brain and visual perception. In: K.M. Fort (Ed.), *Topology of 3-Manifolds and Selected Topics*, Prentice Hall, Englewood Cliffs, NJ, 1062, pp. 240-256.