

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

**UMI**<sup>®</sup>  
800-521-0600



# THE MISSING LINK

IMPLEMENTATION AND REALIZATION OF COMPUTATIONS  
IN COMPUTER AND COGNITIVE SCIENCE

Matthias Scheutz

Submitted to the faculty of the University Graduate School  
in partial fulfillment of the requirements  
for the degree  
Doctor of Philosophy  
in the Departments of Cognitive Science and Computer Science,  
Indiana University

September 1999

UMI Number: 9950828

**UMI<sup>®</sup>**

---

**UMI Microform 9950828**

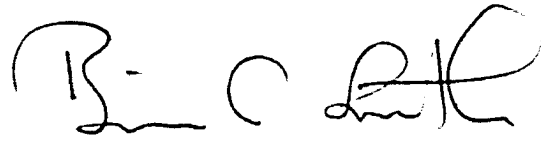
**Copyright 2000 by Bell & Howell Information and Learning Company.**

**All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.**

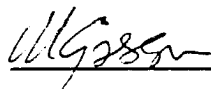
---

**Bell & Howell Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346**

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.




Brian C. Smith, Ph.D.

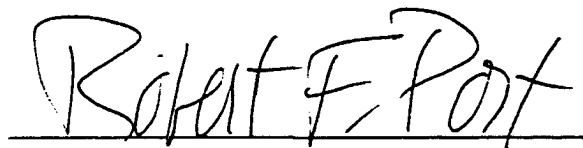


Michael Gasser, Ph.D.

Doctoral  
Committee



Larry Moss, Ph.D.



Robert Port, Ph.D.

September 22<sup>nd</sup>, 1999

© 1999

Matthias Scheutz

ALL RIGHTS RESERVED

iii

*For Colleen*

## Acknowledgments

I am greatly indebted to several people for their help and support during the preparation of this dissertation. First and foremost, I would like to thank Brian Smith for his suggestion to write a dissertation on the “implementation issue” and his offer to supervise it. Without him, this dissertation would literally not have come into being. Not only was he my most faithful reader, as he read several versions of this project at various times, but he was also my most sincere critic insisting many a time that the ideas expressed in the text could be formulated more precisely and more succinctly. His critical mind and his fascination with the philosophy of computing served as a continuous source of inspiration for me to finish what began as a mere “working paper” on some aspects of the usage of “implementation” in computer and cognitive science. Without Brian, this dissertation would not have become a united, coherent whole. Of course, he is not responsible for any remaining shortcomings or flaws, nor are the other committee members, i.e., Michael Gasser, Robert Port, and Larry Moss, with whom I had many inspiring conversations about fundamental issues in cognitive science. In particular, Bob’s strong stance on dynamical systems as the right explanatory formalism in cognitive science enabled me to see what is lacking in computationalism, but also what is good about it. Last, but not least, I would like to thank my wife Colleen Ryan for her patience and support throughout the preparation of this dissertation. She was the major source of motivation to continue this work, especially when I thought I had lost my impetus



completely. Her caring, yet persistent reminders of my self-imposed deadlines helped me to stay focused and eventually complete what I had set out to do. Thank you all!

## **Abstract**

The notion of computation has attracted researchers from a wide range of areas, cognitive psychology being one of them. The analogy underlying the (metaphorical) usage of “computer” in cognitive psychology can be succinctly summarized by saying that the mind is to the brain as the program is to the hardware. Two main assumptions are buried in this analogy: 1) that the mind can somehow be understood computationally, and 2) that the same kind of relation—the implementation relation—that obtains between programs and computer hardware obtains between minds and brains too. While the first assumption has led to fertile research, the second remained mainly at the level of an assumption.

Recently our understanding of the implementation relation has been challenged by claims of Putnam and Searle (both of which are examined in detail) that every physical system can be viewed as implementing every computation. If this were true, then computation would hardly be an appropriate notion for describing mental processes.

Many people have attacked these claims by finding flaws in the respective arguments, some have even attempted positive accounts of what it means for a physical system to implement a computation (e.g., Chalmers or Copeland). While these positive accounts can avoid most of the problems with the intuitive notion of implementation (as pointed out by Searle and Putnam), it is argued that they still do not get at the heart of their criticisms, namely the formation of physical state types. To overcome these problems, a positive account of implementation and physical realization of computations is developed which does not depend on the notion of physical state, but rather develops the notion of

implementation as a series of abstractions over the physical peculiarities of the implementing system.

---

Brian C. Smith, Ph.D.

---

Michael Gasser, Ph.D.

---

Larry Moss, Ph.D.

---

Robert Port, Ph.D.

## Table of Contents

Acknowledgments.....	v
Abstract.....	vii
Table of Contents.....	ix
Table of Figures.....	x
Chapter 1: Introduction.....	1
Chapter 2: Computation, Implementation, and the Computational Claim on Mind.....	11
2.1 Computationalism or the Computational Claim on Mind.....	11
2.2 Computation and the Significance of Turing Machines.....	19
2.3 The Standard Account of Implementation.....	30
2.4 Physical States and Physical Descriptions.....	44
2.5 Computational States and Computational Descriptions.....	55
Chapter 3: CCM's Convinced Critics.....	58
3.1 Attacking Computationalism: Everything Computes!.....	58
3.2 Putnam's Realization Theorem.....	60
3.3 Weak Spots in Putnam's Construction.....	68
3.4 Searle's New Argument.....	76
3.5 Copeland's (Re)construction of Searle's Theorem.....	80
3.6 Implications of Putnam's and Searle's Theorems.....	93
Chapter 4: The Alleged Rehabilitation of Computationalism.....	97
4.1 The Stakes are High: Two Ways to Rescue Computationalism.....	97
4.2 An Analysis of Copeland's Solution of the Implementation Problem.....	101
4.3 The Universal Realization of Psychological Theories.....	109
4.4 Possible Objections to the Extended Version of Putnam's Construction.....	113
4.5 The State-to-State Correspondence View of Implementation.....	118
4.6 An Analysis of Chalmers' Definition of Implementation.....	120
4.7 A Modification of Chalmers' Notion of Implementation.....	133
4.8 A Pitfall for Correspondence Theories: The Slicing Theorems.....	142
4.9 General Problems with the State-to-State Correspondence View of Implementation.....	156
Chapter 5: When Physical Systems Realize Functions.....	163
5.1 Taking the Physical Seriously.....	163
5.2 Setting the Stage: Electromagnetic Fields and Circuit Theory.....	168
5.3 What it Means for a Physical System to Realize a Function.....	171
5.4 Analog Electric Circuits.....	179
5.5 Digital Electronic Circuits.....	187
5.6 From Circuits to Digital Systems.....	198
5.7 Digital Systems and the Theory of Computation.....	203
Chapter 6: Conclusions and Prospects.....	211
References.....	228

## Table of Figures

<b>Figure 2.1</b> The standard account of physical realization of a function: $i$ and $o$ are physical states of the physical system $S$ , $I$ is the interpretation function that maps these states onto the union of the range and the domain of $f$ .	32
<b>Figure 3.1</b> Physical states are defined as sets of maximal states of a system $S$ for a given interval of real-time in such a way that they are in correspondence with automata states in a “run” of the automaton.	64
<b>Figure 3.2</b> Correspondence states are defined as sets of maximal states of all physical states which correspond to the same automaton state resulting in an isomorphic mapping between correspondence and automata state types.	65
<b>Figure 3.3</b> Regions in the wall are singled out to become bearers of labels (a,b,c), which in turn will become the interpretation (via the interpretation function $I$ ) of constants in SPEC (A,B,C).	88
<b>Figure 3.4</b> Computation in a real “von Neumann” CPU: the table shows the values of all registers at the respective time-steps (values reflect the states of the registers after the command has been executed). ‘xxx’ indicates that the actual value does not matter.	88
<b>Figure 3.5</b> Computation in the “wall CPU”: the table shows the values of all registers under the interpretation function $I$ for each interval that is mapped onto computational time-steps.	90
<b>Fig. 4.1</b> The relation between computational and physical states (where the time interval considered is between 8 a.m. and 12 a.m. on March 22, 1998).	107
<b>Fig. 4.2</b> The relation between grouped physical states (i.e., interval states) and automata states.	108
<b>Figure 4.3</b> The simple physical system $P_1$ consisting of a battery, a switch, and a light bulb.	126
<b>Figure 4.4</b> The automaton $M_1$ with inputs from $\{a,b\}$ and outputs from $\{0,1\}$ .	127
<b>Figure 4.5</b> The physical system $P_2$ consisting of a battery, two switches, and a light bulb.	128
<b>Figure 4.6</b> The causal structure of physical system $P_2$ , which defines the isomorphic automaton $M_2$ .	129
<b>Figure 4.7</b> A graph of the states and transitions in the two-switch system $P_2$ after inputs, outputs, and states have been redefined. States encircled by a dashed line are mapped onto the same automaton state.	130
<b>Figure 4.8</b> A graph of automaton $M_3$ , which is bisimilar to $M_1$ .	139
<b>Figure 4.9</b> A graph of automaton $M_4$ , which is bisimilar to $M_1$ .	141
<b>Figure 4.10</b> A graph depicting the causal structure of the physical system $P_{1,4}$ .	144
<b>Figure 4.11</b> A graph depicting the physical system $P_{1,2k}$ (dotted lines indicate states that lie within the same time interval).	145
<b>Figure 4.12</b> Transitions in $P_{1,2k}$ that are mapped onto all $k$ automata transitions.	146

<b>Figure 4.13</b> An automaton which could be used to argue against the validity of the Slicing Theorem.....	147
<b>Figure 4.14</b> Transitions in the 3-switch system that are mapped onto all $n$ automata transitions.....	151
<b>Figure 5.1</b> The relation between the resistor and the function it realizes: given a certain <i>Real</i> $r$ , the value $f(r)$ is then obtained by taking the <i>encoding of the input</i> $I(r)$ , applying it to the resistor, and then decoding the output $F(I^1(r))$ , using the <i>output encoding</i> , resulting in $O^1(F(I^1(r)))$ , which is equal to $f(r)$ . .....	176
<b>Figure 5.2</b> The oscillator circuit: once the input is changed from 0 to 1, the output will oscillate between 0 and 1 as long as the input is 1.....	197
<b>Figure 6.1</b> The difference between the state-to-state correspondence view (left) and the “functional realization model” (right): in the latter, functions serve as mediators between physical systems and computations. ....	216
<b>Figure 6.2</b> An automaton, which computes the function realized by the oscillator circuit from the previous chapter. ....	217
<b>Figure 6.3</b> Relocating the alleged gap between computations and implementations as a gap between the world and our physical description of it.....	225

## Chapter 1:

### Introduction

The notion of computation has attracted researchers from a wide range of disciplines. Some of them, for instance cognitive psychology, are very different from and seemingly unrelated to computer science. While the reasons for these various attractions to the notion of computation differ from science to science, in the case of psychology it might be due to its ability to describe real-world processes in an abstract, formal, syntactically specified manner. In particular, the capacity of computers to process information and the fact that programs can specify sequences of actions of and in computers inspired some psychologists to think of cognitive functions in terms of programs and of the brain as a computer (running these programs). The analogy underlying the (metaphorical) usage of “computer” and “program” in cognitive psychology can be succinctly summarized by saying that “the mind is to the brain as the program is to the [computer] hardware” (Johnson-Laird, 1988, the remark in brackets is mine, or Searle, 1984).<sup>1</sup> Eventually, the guiding ideas of this analogy became so prominent, originally in psychology, later in artificial intelligence, as to establish a genuine research paradigm, commonly called “computationalism” or “the computational claim on mind” (abbreviated as “CCM”).<sup>2</sup>

---

<sup>1</sup>Sometimes the term “software” is instead of the term “program” in this analogy so as to not be committed to any particular programming language, and I will follow this idea by using the more general term “computation”, which is thought to encompass any computational formalism.

<sup>2</sup> I am not necessarily committed to the term “research paradigm” if one wants to understand it in a strict Kuhnian sense, in which case one should read it as synonymous with “widely accepted guiding idea for a research direction”.

Two main assumptions are buried in the “mind:brain=program:hardware”-analogy, also called the “computer metaphor”: 1) that the mind can somehow be “understood as a computation” or be “described by a program” (this will require the adoption of a notion of computation or program, respectively), and 2) that *the same kind* of relation—(a version of) the *implementation relation*—that obtains between programs and computer hardware obtains between minds and brains too. Assumption 1) has led to fertile research in psychology as well as artificial intelligence (AI) collecting evidence for its truth, while assumption 2) by and large remained at the level of an assumption, presumably because neither AI researchers nor psychologists need to pay attention to it. AI researchers, who build computational models, implicitly deal with the implementation relation of software on computer hardware on a daily basis, but are not concerned with the implementation relation of minds on “brain hardware”; nor are psychological studies, which remain at the level of “program description”.<sup>3</sup> Having established that *some* notion of implementation plays, or to put it more forcefully, *has to play some* role in the enterprise of computationalism, it is crucial to explicate the relevant notions of implementation and assess their role for the computational claim on mind.

One common belief, I take it, that underlies all computationalist explanations is the claim that programs, or more generally computations, somehow “mirror” the causal structure of computers (=hardware) in particular, and physical systems in general. Motivated by computational practice, it is widely held that this mirroring is established by setting up a functional correspondence between computational and physical states (the

---

<sup>3</sup> Neuroscience would probably be the closest discipline concerned with implementation issues of brains. Yet, neuroscientists do not attempt to relate program-like descriptions to brain areas, rather they attempt to study the



“implementation relation”). For example, there is a tight correspondence between parts of the architecture of a von Neumann CPU and expressions of the assembly language for that CPU. Another example would be a logic gate (e.g., an AND gate), whose “computational capacity” is described by a Boolean function, the values of which in turn can be related to physical magnitudes in the physically realized circuit.

While we can more or less easily establish such a functional correspondence between physical and computational states for certain artifacts (i.e., devices we have designed to allow for computational descriptions), it unclear that this can be done for natural kinds (such as brains) too. In particular, one has to be aware that any such correspondence crucially depends on “adequate” physical states. In the case of electronic devices the appropriate physical states can be determined either by asking the engineers who designed the devices or by looking at the blueprint and comparing it to the computational description of the device. In the case of biological systems, however, such states are not clearly defined. Consider, for example, physical states of a pyramidal cell: which of those states could correspond to computational states such that the respective computation captures essential parts of the causal structure of these cells? It has been suggested that “firing” vs. “not firing” would be “natural” candidates (e.g., by McCulloch and Pitts, 1943)<sup>4</sup>, but it turns out that this computational model is too reductive as essential temporal processes (such as temporal integration of signals, maximal firing rates, etc.) are completely neglected. Hence, more factors about pyramidal cells need to

---

functional role of these areas (with respect to the rest of the brain) directly by virtue of their physiological functions.

<sup>4</sup> Interestingly, McCulloch and Pitts (1943) define a notion of realization in their paper, which holds between a logical description (a “temporal propositional expression”) and a neural net. This is, to my knowledge, the first time that this notion has been used to indicate that a “computational description” is *implemented* in a physical system

be taken into account, yielding more physical states that have to correspond to computational ones, etc. Artificial neural networks seem to be promising candidates for such computational descriptions, but the issue has to my knowledge not been resolved yet. It might well be that in the end the complex behavior of pyramidal cells defies a computational description, but this is obviously an empirical issue.

To summarize: computations “mirror” the causal structure of a system under a given correspondence function between computational and physical states *only relative to the choice of the physical states*. Computational explanations of the behavior of a given physical system, therefore, depend essentially on those physical states of the system that can be set in correspondence to some computation. This dependence, *per se*, is not problematic as long as one can assume “appropriate physical states” of a system (e.g., as in the case of electronic devices). If, however, it could be shown that for any computational description one can find “physical states” of a given system, which can be set in correspondence with the computational ones and are, furthermore, *appropriate* (in a certain sense of “appropriate” which will depend on the underlying physical theory), then computational explanations would be in danger: every system could then be seen to compute! In other words, computationalism would be vacuous if every physical system could be viewed as implementing every computation.

Indeed, it has been argued that rocks, for example, can be seen to implement any finite state automaton, or that walls to implement the WORDSTAR program (as has been suggested by Putnam, 1988, and Searle, 1992). If these claims are true, then something clearly must have gone wrong with our computational descriptions: *pan-computation*

(i.e., every physical system is computing everything) is not a tenable view, neither from the theoretician's nor the practitioner's perspectives.

Many people have attacked Putnam and Searle's fatalistic views and argued that there is no reason for such pessimism (e.g., Chalmers 1996, Chrisley 1994, Melnyk 1996, Endicott 1996, et al.), mostly by pointing to flaws in Putnam and Searle's argumentation (such as wrong notion of causality, wrong notion of state transition, etc.). Some have even attempted a positive account of what it means for a physical system to implement a computation (Chalmers 1994, Copeland 1996), which in their view does not lead to pan-computation. If successful, such an account of implementation, would not only rescue computationalism from being vacuous, but also provide a means to assess the computational capacities of physical systems not of interest to cognitive scientists, but possibly to engineers. It is part of the purpose of this work to assess and criticize the tenability of these (positive) accounts of what it means to implement a computation.

Both positive accounts of implementation investigated in this work are in my view ultimately based on the idea of a state-to-state correspondence, that is, of a tight correspondence between physical and computational states. While both are intended to overcome any negative implementation result (such as Putnam's and Searle's arguments), it seems to me that there are intrinsic problems connected with any approach to implementation that is directly based on the notion of physical state. To argue that this dependence is indeed the weak spot of any such definition of implementation, it would suffice to show that "legitimate physical states" (i.e., states that are in some sense "natural") of a particular physical system (*not necessarily any arbitrary system*, as Putnam and Searle claimed) can be defined and set in correspondence to states of a

computation, which we would not be willing to attribute to the system. Such an argument would establish that the notion of physical state is too permissive to serve as a basis for a *general theory* of implementation.<sup>5</sup>

In particular, it would follow that correspondence views of implementation (i.e., any view that is based on establishing a correspondence between physical and computational states) need to be revised or have to be given up completely.<sup>6</sup> In favor of giving them up altogether counts the intuition that an account of implementation should not have to be concerned with (the formation of) physical states at all, in favor of keeping the conviction that computations have to be somehow tied to the physical world.

These two seemingly contradictory ideas, that 1) what a system implements should not depend on how we choose and combine physical states, and 2) what a system implements should depend on the physical description of the system, are the guidelines for my own positive account of implementation—the other part of this work.

The basic structure of this investigation is fourfold. First, the intellectual territory is laid out. Second, the attacks on computationalism, that is, on the notion of implementation used in computationalism are presented in detail. Then reactions to these attacks, in particular, two kinds of rescue attempts of the notion of implementation by Chalmers and Copeland are analyzed in detail, followed by a presentation of counterexamples to these approaches. Finally, a new account of implementation is

---

<sup>5</sup> Note that it is *not* claimed that state-to-state correspondence views fail to establish implementation relations in every case. That would be clearly false as simple examples like an AND-gate and its Boolean function demonstrate. Rather it is claimed that there are systems for which these views on implementation deliver unwanted implementation results, which are, in my view, mainly due to problems with the notion of physical state (and not so much the notion of correspondence).

<sup>6</sup> If the argument in this work against state-to-state correspondence views is indeed successful, then it is hard for me to imagine that any revision could salvage such views...

suggested that does not suffer from the deficiencies of the other approaches (as it is not based on the notion of physical state).

Mapping this outline onto chapters, the following gives a more detailed summary of the project:

*Chapter 2* first spells out briefly some of the crucial assumptions behind the term “computationalism” with respect to the role the notion of implementation plays in it. Then the role of the classical notion of computation for computationalist explanations, that of “Turing computability”, is analyzed and it is argued that from the concept of Turing machine alone it is not clear when a physical system implements one of these “often-imagined, but seldom seen devices” (B. Smith). A formal analysis of the standard account of the “implementation of a computation” (as presented by Stabler) and some of its pitfalls is followed by a digression into the nature of “physical and computational states”, two notions that are an essential part of every theory of implementation.

In *Chapter 3*, I locate one of the most severe shortcomings of the computationalist position in a particular kind of explanatory gap: that the acceptance of standard notions of computation (such as “Turing-computability”) leaves open exactly how abstract “computations” are linked to concrete physical systems realizing them. This lack of a reasonable theory of implementation, which specifies the link between computations and “computers”, is then made apparent in the analysis of two provocative claims: Searle’s claim that (under the received notion of implementation) walls “implement” the Wordstar program and Putnam’s that every ordinary open system “implements” every finite state machine.

In *Chapter 4*, I first argue that Searle and Putnam's arguments against computationalism are based on the same criticism: as long as physical states (of a given physical system) can be chosen freely, one can always relate these physical states to computational states (of an arbitrary computation) in such a way that the physical system can be viewed as implementing that computation. In particular, it follows that the problems of the "logical view of implementation" (as exhibited by Copeland) are the same as those of the "state-to-state correspondence view of implementation" as held by Chalmers. A thorough examination of Chalmers' view then shows that contrary to what it promises, it does not force the mapping between physical and automata state types to be isomorphic. Hence, computation together with Chalmers' notion of implementation cannot be used to describe the causal structure of physical systems in computational terms. I argue, furthermore, that even if this deficiency is accounted for, the problem of physical state type formation common to all "state-to-state correspondence views" on implementation still remains unsolved. The lack of general criteria for the definition of physical states and their formation into physical state types, so I claim, is what ultimately admits unwanted implementations of the sort that Searle and Putnam have pointed out. To illustrate this point, a very simple physical system is studied, which (according to Chalmers' view) implements an absurdly large class of computations, once a certain kind of physical state formation is introduced. The resulting state types, although somewhat unusual, are theoretically and pragmatically acceptable. Moreover, state transitions between tokens of those types do not lack the necessary counterfactual support for which Putnam and Searle's approaches have been criticized. From these considerations it is concluded that one is left with two non-exclusive options: either formation rules for

physical states (and state types) of arbitrary physical systems can be provided in general without recourse to a particular physical system, or the so-called “semantic” as well as “state-to-state correspondence” views of implementation have to be abandoned as plausible candidates for a *general* theory of implementation—they might work to assess the computation implemented by a system in cases where physical states for the system are given, but fail for systems, for which no notion of physical state is available.

*Chapter 5* attempts to tackle the notion “computation” from a practical point of view (by looking at the devices that humans design, build, and use). Physical descriptions of these devices lend themselves to the very general notion “realization of a function”, a *precursor and amalgam* of the notions “computation” and “implementation”. The notion of realization of a function, in turn, becomes more and more restricted as practical and/or physical constraints (such as measurability, feasibility, etc.) are incorporated. Although this account is far from being a full-fledged theory of implementation, it can describe relations between abstract and concrete systems in terms general enough to subsume standard computational systems. These systems are viewed as being realized by special kinds of physical systems, called “digitality supporting systems”. The main virtue of this approach for any resulting theory of computation is, then, that the respective notions of “computation” and “implementation” are not, as otherwise commonly maintained, defined at a set-theoretic level. Rather, they are mathematical extractions obtained from behavioral descriptions of concrete systems, developed from a progression through various levels of abstraction, thereby never abandoning—and, thus, retaining up to the highest level—their close ties to the concrete world.

Finally, *Chapter 6* concludes by summarizing the achievements of this new account of implementation and putting them in perspective with Searle and Putnam style criticisms. Besides avoiding the shortcomings of other attempts to rescue the notion of implementation, and thus computationalism, the notion of “realization of a function” sheds new light on the notion of computation itself: computations *cannot* be viewed as independent of physics anymore. Alas, their intrinsic ties to the mathematical descriptions of physical objects are revealed. They are only observer-relative to the extent that physical descriptions of physical systems are observer-relative. Furthermore, the Church-Turing thesis does not necessarily have to apply to computers (i.e., physical systems that compute) anymore—there might be systems that can be described in certain physical theories, which can compute non-recursive functions.

The last chapter also briefly points to some of the research directions that might develop out of the view on implementation suggested in this work (e.g., the possibility of digital systems to allow for representations).



## **Chapter 2:**

# **Computation, Implementation, and the Computational Claim on Mind**

## **2.1 Computationalism or the Computational Claim on Mind**

To pinpoint what is subsumed under the notion of computationalism would be a genuine research project in its own right given the multitude of different proposals and analyses of it (e.g., see Dennett, Sterelny, et. al.). It is not uncommon to find slogan-like phrases such as “the brain is a computer” (Shapiro, 1995, p. 523), “the mind is the program of the brain” (Ned Block, 1995), “computation is cognition” (Sterelny, 1989, et. al.), or “the mind is a computer” (Smith, 1995) just to name a few, which all hint at the main credo of computationalists. But note that these statements are necessarily condensed and cannot be taken at face value; all taken together would equivocate essentially distinct notions. Turning them into substantial claims requires a careful analysis (e.g. see Smith’s analysis of the claim “the mind is a computer”).

Some people have attempted to spell out principles that, in their view, underwrite the computationalist position. Dietrich’s “computational manifesto” (Dietrich, 1990) is an example of such a text, which is in particular interesting to the current project, as it attempts to explicate what it means for a system to compute a function.

Dietrich defines computationalism as “the hypothesis that cognition is the computation of functions. If computationalism is correct, then scientific theories of

cognition will explain it as the computation of functions.” (Dietrich, 1990, p. 135) Obviously, in interpreting such a statement, much hinges on the involved notion of computation, that is, what it means to compute a function. For example, does computing a function imply “following rules”, or “executing an algorithm”, or would non-algorithmic methods that arrive at the same results count as well? The importance of these issues will become particularly apparent, if one attempts to provide a formal definition of the notion “computation of a function” (see section 3).

The project of computationalism, according to Dietrich, is to determine “[...] which functions cognition is, i.e., which specific functions explain specific cognitive phenomena”, construing computationalism as foundational and methodological. Thus, “by itself, it makes no claims about which functions are computed, except to say that they are all Turing-computable (computationalists accept the Church-Turing thesis), nor does it make any specific claims as to how they got computed, except to say that the functions are systematic, productive, and interpretable in a certain way” (Dietrich, 1990, p. 135).<sup>7</sup>

Dietrich isolates four properties of computational explanations:

1. They explain an ability or capacity of the system to exhibit certain behavior
2. They must be systematic (exhibit the system *as a system*)
3. They are interpretive (which is logically entailed by the first property)<sup>8</sup>

---

<sup>7</sup> While it might be true that most computationalists are committed to the “Church-Turing-Thesis”, it certainly not true that *all* computationalists feel that they need to take a stance on this issue (quote commentary...). Even more controversial is the requirement of the “productivity of computation”, but I will not address this issue here.

<sup>8</sup>With this Dietrich intends that computations can be interpreted to have content, i.e., that computations are manipulations of representations.

4. They require computational functions to be productive (to “compute a value” as opposed to “look it up in a table”, for example)

These four properties, according to Dietrich (1990, p. 140) “work together to form an explanatory strategy which is scientifically respectable and robust”. It is the goal of every computationalist to show *that* and *how* certain physical systems compute certain functions.

Dietrich distinguishes his “computationalism” from what he calls “computerism” and “cognitivism”. The former is the thesis

“that explanations of cognition will look like procedures for our current (late 20<sup>th</sup> century) computers. Computerism is thus tied to a specific computer hardware architecture, in particular a serial architecture. Note that the computerist is interested in more than the functions which get computed; she is interested in how they are computed.” (Dietrich 1990, *ibid.*)

Dietrich does not fail to add that it is very unlikely that anybody seriously holds this view. However, as he points out quite correctly, computationalists have often been wrongly accused of taking such a stand.

Cognitivism, according to Dietrich is yet another restrictive version of computationalism, which purports that

“the functions which explain cognition are rational functions defined over propositions (or sentences). [...] Cognition is the production of output propositions which are rationally related to input propositions. For the cognitivist, inference is the paradigmatic cognitive function.” (Dietrich 1990, p. 141)

Construing “cognitivism” this way, computationalism does not entail cognitivism, since computational processes are not restricted to propositional or sentential structures. As a consequence, cognitivism could be false, if it turned out that the most important functions are not inferences, according to Dietrich, while computationalism could still be true, if these functions were computational. In short, computerism is a claim about *how* functions are computed, whereas cognitivism is a claim about *which* functions are computed.

Not everybody would agree with Dietrich’s proposed categorization (as is apparent from the numerous commentaries to Dietrich’s target article). Searle, for example, distinguishes three kinds of questions, the answers to which give rise to three claims on mind differing in strength according to what they claim (Searle, 1984, 1992):

1. Is the mind a computer program?
2. Is the brain a digital computer?
3. Can the operations of the brain be simulated on a digital computer?

The three related views, which answer the respective question affirmatively, are then:

1. Strong AI
2. Computationalism/Cognitivism
3. Weak AI

Searle's stance is that questions 1 and 2 must be answered negatively, whereas a positive answer can be given to question 3. Without going into detail, it is *prima facie* not clear at all how Dietrich's notions of "computationalism", "computerism", and "cognitivism" would figure in Searle's categorization.

This apparent terminological discrepancy is indicative of the ongoing foundational debate in cognitive science and the philosophy of mind about the nature and extent of computationalist positions. Because of different aims, terminologies and analyses, computationalist positions differ from author to author, and it is important to keep these different claims on mind (i.e., different versions of computationalism) separate.

There are, for example, other descriptions of computationalism that emphasize the information processing capabilities of computers (such as Dennett's who views computationalism as the conjunction of three theses: "thinking is information processing", "information processing is computing (is symbol manipulation)", "the semantics of those symbols connect mind and world"). Again others underscore essential distinctions (such as the "analog/digital" distinction, etc.) maintained by computationalists (e.g., see Demopoulos, 1987).

To avoid confusion, I will use the term "computationalism" in this work as standing for any position that is committed to the claim that mental states *are* computational states. Note that this notion is "parameterized" by the notion of *computation*: since what counts as computational with respect to one account of computation, might not count as computational with respect to another, what are mental states, then, with respect to one account of computation, might not be mental states with respect to another. Fortunately, computationalists have traditionally been committed to a particular notion of

computation, the classical notion of Turing machine computability (see the next section), which justifies the above terminological “sweep”. Another, more important reason, why the peculiarities of the various positions on computationalism do not matter for the current endeavor is that the notion of implementation is directly connected to the notion of computation. Hence, as long as the notion of computation is given, one can study the related implementation issues (that is, when a computation of a particular kind is implemented) without having to pay attention to extraneous factors such as the representational capacities of the symbols involved in the computation, for example.

Computationalism, so construed, belongs to the wider class of different functionalist positions (e.g. Turing-machine functionalism, psychological functionalism, homuncular functionalism, causal functionalism, etc.—see Garfield, 1995). Put crudely, functionalism is a commitment to the independence of a functional level of description of a “system” (e.g., a cognitive system)—this level is viewed as a genuine level in its own right, which may or may not be reducible to lower levels of description (e.g. see Block 1996). As an explanatory device for theories of mind in cognitive science, it combines the advantages of behaviorism and type-identity theory without also inheriting their disadvantages (e.g., Kim, 1996). In particular, functional accounts honor the potential “multiple realizability” of functional architectures, that there might be different physical realizations for one and the same functional organization.

Functional accounts of the mental differ, among other things, on their ideas about the relation between the functional architecture and its (potential) physical realization. The difficulty with any such explanation is to steer between the Scylla and Charybdis of being either too vague (so as to not have a theory of realization at all—the “free-floating”

minds) or too rigid (thus reducing functionalism to a version of the type identity theory with all of its flaws). This unbridged gap between functional and physical architectures is still haunting today's versions of functionalism.

This is where computationalism can play its major trump card: by claiming that functional states *really are* computational states, functional states can be viewed as states of a computation that can run on *any hardware supporting the abstract computational architecture of the computation*.

For example, since any computation can be simulated by a universal Turing machine (see the next section), it follows that this versatile “machine” can *implement* any functional architecture (“functional” in the sense of “computational”)—or so it is claimed (see Fodor, 1981). Note that hidden in the last argument is the assumption that it is clear how Turing machines can be physically realized, that is, how they can be built (and not any Turing machine, but a universal one!).<sup>9</sup>

It seems that what distinguishes computationalism from all other versions of functionalism is that by pointing to technology, to computational practice, to the kinds of machine we have built, it provides a (tenable) notion of implementation (something that cannot be said about its fellow contenders: if functional states *are not* computational states, what exactly are they? And how are they related to states of a physical system?). Classical artificial intelligence (or “strong AI” in Searle’s terminology), in particular, is built upon the assumption that we *know* how to implement programs on computers (that we built):

---

<sup>9</sup> In a sense, it suffices to know how to realize a universal Turing machine, as it can “simulate” any other Turing machine.

“Artificial intelligence is about programs rather than machines only because the process of organizing information and inputs and outputs into an information system has been largely solved by digital computers. Therefore, the program is the only step in the process left to worry about.” (Bridgeman, 1980)

It is this notion of implementation of a program, stemming from computational practice, which, by analogy, is stretched to make it fit the notion of physical realization of minds in “wetware”—whether this computational girdle is too tight for biological systems is still an open and highly debated issue.

This short and sketchy exposition of the role of implementation in computationalism will be fleshed out in the following, not only to substantiate computationalist claims about 1) the notion of computation and 2) the relation of computational and physical states, but also to be able to analyze various criticisms of computationalism later, which attack this very notion of implementation. The consequences of these attacks for functionalist explanations, if justified, are quite significant: if computationalism *fails at what is thought to be its major strength and advantage compared to other functionalist accounts*, what should we make of functionalism in general? Can functionalist theories of the mental be taken seriously if one cannot explain how functional states obtain their causal powers from the underlying physical system?<sup>10</sup> If functional states cannot be individuated at a lower level (i.e., a level lower than the level of functional description),

---

<sup>10</sup> Of course, not all functionalist accounts revert to computational states in order to explain how functional states obtain their causal powers. Yet, the ones that do not involve computational states wrestle with a convincing explanation of how functional states can be causally efficacious. Simply viewing functional states as physical states, which explains how these states get their causal powers, leads immediately to the question what kind of relationship the functional states bear to the class of their physical realizers...



that is, if functional states are truly independent from their physical manifestations, then—unfortunately—Cartesian dualism might be on the up and up again.

## **2.2 Computation and the Significance of Turing Machines**

Most computationalists adhere to “standard notions” of computation inherited from formal logic, which were conceived by the forefathers of the theory of computation (Gödel, Church, Kleene, Post, and Turing) before computers the way we know them today even existed.<sup>11</sup> The probably most crucial insight of the 30ies with respect to the meaning of the “intuitive notion of computation” (then called “effective calculability”) was that three different attempts to characterize it formally could be proven to be equivalent: the class of recursive functions equals the class of  $\lambda$ -computable functions equals the class of Turing machine computable functions. These equivalence results are possible, because what “computing” means with respect to any of the suggested formalisms (i.e., what a “computation computes”) is expressed in terms of functions from inputs to outputs; and using functions as mediators, the different computational formalisms can be compared according to the class of functions they compute.

Later, other formalisms such as Markov algorithms, Post systems, universal grammars, PASCAL programs, as well as various kinds of automata were also shown to “compute” the same class of functions, referred to as “recursive functions”. Note that an identification of “computation” with “computing functions” is a necessary prerequisite

---

<sup>11</sup> This is not to say that people have not thought about how to build computers. The idea of a mechanical calculator goes back at least to Pascal and quite sophisticated machines have been conceived and some even built. One of the most extraordinary developments was Babbage’s “difference engine”, which he himself unfortunately could never complete.

for these equivalence results. The extensional identity of all these formalisms supports a famous thesis formulated by Church as a definition:

“We now define the notion [...] of an effectively calculable function of positive integers by identifying it with the notion of a recursive function on positive integers (or of a  $\lambda$ -definable function of positive integers). This definition is thought to be justified by the considerations which follow, so far as positive justification can ever be obtained for the selection of a formal definition to correspond to an intuitive notion”. (Church, 1936, p. 356 in Davis 1965, p.100)

Using the various equivalent results it follows from “Church’s Thesis” that any of the above mentioned formalisms captures our intuitive notion of computation, that is, *what it means to compute* (Note that this presupposes the construal of computation as “computation of a function”). Although this thesis cannot be proved in principle as mentioned by Church himself, it became more and more plausible as newly conceived computational formalisms were shown to give rise to the same class of “computable” functions.

What is common to all these computational formalisms, besides their attempt to specify formally our intuitive notion of “computation”, is their property of being independent from the physical. In other words, computations in any of these formalisms are defined *without* recourse to the nature of physical systems that (potentially) realize them. Even the Turing machine model, which is often taken to be the prototype of a “mechanical device”, does not incorporate physical descriptions of its inner workings, but abstracts over the mechanical details of a physical realization (this issue will be addressed in more detail later).

It was this prominent feature, the independence of computations from their physical realizations, that allowed logicians to study classes of functions that could be “computed” according to any one formalism. Limiting results such as the “Halting problem” for computers can be derived just by virtue of the specification of “computations” as “recursive” without invoking any facts about the physical details of their realizers.

The independence of computations from the physical realizers also made the notion of “computation” an attractive candidate for functional explanations of mental processes in cognitive science, because psychological functions could be studied independent of the underlying system by solely paying attention to the role they played in the whole network of different psychological functions. The “computer metaphor” helped to establish cognitive psychology as a research area in its own right in the late 50ies, thus overcoming then still dominating behavioristic positions.

Among the various computational formalisms, the Turing machine model has gained most attention in philosophy and cognitive science for various reasons, but certainly also because of its property of being an abstract model of a hypothetical “mechanical” machine. Especially for “mechanical explanations of mind” this model seemed to be well-suited because of its dialectic nature of being in some sense separated from *and* at the same time connected to the physical.

It is often tacitly assumed—and I will argue mistakenly so—that the Turing machine links computations to a mechanical device, to something physical. Certainly none of the other formalisms provides such a link, they simply assume that such a link is possible (e.g., to explain human computational behavior in terms of  $\lambda$ -computable functions).

Computations according to the other formalisms are only indirectly linked to systems that could perform and/or realize them *via* the computational equivalence to the Turing machine model. So the Turing machine model works as a mediator between abstract computational descriptions and possible systems that could realize these descriptions. This means, however, that if the Turing machine model fails to withstand attacks regarding its implicit notion of implementation, then all the other formalisms are “cut off from the concrete” as well. It will thus be valuable for understanding the importance that computationalists attribute to the Turing machine model to consider Turing’s original motivation that led to his development of “Turing machines”.

Interestingly enough, Turing (1936) invented his machine model of “computation” in order to capture the human activity of “computing”, i.e., the processes a person (the “computer”) goes through while performing a calculation or computation using paper and pencil. He was not concerned with digital computers at all (although he uses the term “computer”, but for a human person doing computations—at that time digital computers did not yet exist) or the foundations of computing so much as with the problem of analyzing and modeling what the possible processes are that people go through when they “blindly” follow rules to do computations. In his analysis of the limitations of the human sensory and mental apparatus five major constraints for doing “automatic computations” crystallize (here I follow Gandy’s presentation):<sup>12</sup>

---

<sup>12</sup> Note that “Turing’s account of the limitations of our sensory and mental apparatus is concerned with perceptions and thoughts, not with neural mechanisms. And there is no suggestion that our brains act like Turing machines.” (Gandy, 1988, p. 87)

- Only a finite number of symbols can be written down and used in any computation;
- There is a fixed bound on the amount of scratch paper (and the symbols on it) that a human can “take in” at a time in order to decide what to do next;<sup>13</sup>
- At any time a symbol can be written down or erased (in a certain area on the scratch paper called “cell”);
- There is an upper limit to the distance between cells that can be considered in two consecutive computational steps; and
- There is an upper bound to the number of “states of mind” a human can be in and the current state of mind together with the last symbol written or erased determine what to do next.

Although there are certainly some steps in Turing’s analysis of an abstract human being performing calculations that seemed rather quick and not too well supported (e.g, the transition from writing symbols on a two-dimensional sheet of paper to writing symbols on a one-dimensional “tape”),<sup>14</sup> one can summarize the above in Gandy’s words as follows:

---

<sup>13</sup> This requirement does not exclude an arbitrary amount of scratch paper. It just delimits the range of perception, i.e., the amount of information the human “computer” can use at any given time to determine the next step in the computation.

<sup>14</sup> While it can be shown that a “two-dimensional tape” does not increase the power of a Turing machine, it might change things for human computers significantly. One could argue that by presenting values of a function in a two-dimensional arrangement we might be able to see “patterns” that could not be appreciated in a one-dimensional representation. Assuming then that humans have a capability of “seeing the Gestalt of something” (call it “mathematical intuition”, “higher level comprehension”, etc. as held by Penrose, Gödel, et al.), they might be able to outperform Turing machines on certain problems (e.g. diagonal arguments)—this ability basically would correspond to some kind of “non-local” operation, see also Gandy 1980.

“The computation proceeds by discrete steps and produces a record consisting of a finite (but unbounded) number of cells, each of which is blank or contains a symbol from a finite alphabet. At each step the action is local and is locally determined, according to a finite table of instructions.” (Gandy, 1988, p. 81)

In other words, by “abstracting away” from persons, scratch paper, etc., Turing (1939) claimed that all “computational steps” a human could possibly perform (only following rules and making notes) could also be done by his machine. That way the Turing machine became a model of human computing, an *idealized* model, to be precise, since it can process and store arbitrarily long, finite strings of characters.<sup>15</sup> It is worth pointing out that Turing as opposed to Church did not only state a “thesis” regarding the intuitive notion of computation, but he actually proved a *theorem* (see also Gandy, 1988, p. 83, who restates Church’s Thesis as Turing’s Theorem):

*Theorem 2.1:* [Turing] “Any function that can be computed by a human being following fixed rules, can be computed by a Turing machine”.

This much stronger result, namely that of having proved that humans beings (following fixed rules as defined in Turing’s analysis) cannot compute more functions than Turing machines, is often unappreciated and furthermore obscured by calling Church’s Thesis the “Church-Turing Thesis” (e.g., see the Stanford Encyclopedia of Philosophy or

---

<sup>15</sup> Note that the level at which the mechanism of a Turing machine is described lies above the “mechanical level of description of physical bodies”. It is rather the same at which we describe the behavior of a person when he or she performs a computation. If one would like to put it provocatively, then Turing machines are described “intentionally”.

Cleland, 1993).<sup>16</sup> Turing also believed the converse, that every function computed by a Turing machine can also be computed by a human computer (although this, again, does not take time and space restrictions seriously, but rather assumes an abstract human computer not subject to these worldly limitations). In particular, Turing was convinced that “the discrete-state-machine model is the relevant description of one aspect of the material world—namely the operation of brains”. (Hodges, 1988, p. 9) The origins of Turing’s claim can be found in the intrinsic connection between the notion of “computability” and Gödel’s notion of “demonstrability” (of a proof in a formal system): that which can be “demonstrated” using “definite methods” amounts to what can be done by a Turing machine (see Turing, 1936). By relating the limitations of formal systems as pointed out by Gödel to the limitations of his machine model, Turing

“[...] perceived a link between what to anyone else would have appeared the quite unrelated questions of the foundations of mathematics, and the physical description of mind. The link was a scientific, rather than philosophical view; what he arrived at was a new materialism, a new level of description based on the idea of discrete states, and an argument that this level (rather than that of atoms and electrons, or indeed that of the physiology of brain tissue) was the correct one in which to couch the description of mental phenomena” (Hodges, 1988, p.6)

Turing’s analysis of the human computer was essentially based on the idea that there can be only finitely many different mental states:

“[...] because these states must somehow be stored in the mind, in order that they can all be ready to be entered upon. An alternative way of defending this application of the

---

<sup>16</sup> A very thorough analysis of the different claims hidden in Church’s and Turing’s theses can be found in

principle of finiteness is to remark that since the brain *as a physical object is finite*, to store infinitely many different states, some of *the physical phenomena which represent them* must be 'arbitrarily' close to each other and similar to each other in structure. These items would require an infinite discerning power, contrary to the fundamental physical principles of today." (Wang, 1974, p.93, italics are mine)

The emphasized phrases clearly indicate that Turing used implementation considerations to make his claim that there could be only finitely many mental states. While it is not clear what kind of notions of implementation he had in mind, it seems safe to assume that he believed that the human brain could only have finitely many different states, which in turn had to be related to states of the mind. Given these assumptions regarding mental states it follows that his machine model can indeed perform and/or emulate any operation of a real brain:

"Turing wished from the beginning to promote and exploit the thesis that all mental processes—not just the processes that could be explicitly described by "notes of instructions"—could be faithfully emulated by logical machinery." (Hodges, p.8)

In his famous 1950 paper on machine intelligence, in which he introduced the "imitation game", nowadays better known as the "Turing test", Turing discusses and rebuts various arguments against his machine model and its capabilities at length. The original intention to capture human computational activity in an abstract model had given way to the idea of potential intelligent discrete-state machines that succeed in tricking and outsmarting human players in the imitation game. This shift in emphasis from the abstract model to concrete realizations of such machines is particularly apparent from the predictions of

---

Kearns (1997).



how many states a machine would need to model human memory (see Turing, 1950). And even before that Turing takes implementation issues seriously when he considers methods of constructing machines and whether machines so constructed should be allowed to participate “as machines” in the imitation game.

So, at the core of the concept “Turing machine” lies the idea of a *mechanical* device that can be actually *built* (the infinite tape being substituted by an arbitrarily extendable tape, for example), or that, when taken as an idealized entity, *exists* in Plato’s heaven. Either way, this real/ideal machine is described or modeled by a mathematical structure, i.e., the quintuple that is usually taken to be *the* Turing machine. To distinguish between the Turing machine *qua* machine (real or ideal) and the mathematical structure, I will introduce subscripts ‘*P*’ and ‘*M*’ standing for “physical (system)” and “mathematical (structure)”, respectively.

The Turing machine<sub>*M*</sub> serves a twofold purpose: it is not only thought to be a mathematical model of the Turing machine<sub>*P*</sub> but also an adequate representation of a human person performing computations (with paper and pencil following rules). Turing machine<sub>*M*</sub> somehow is an *abstraction* of Turing machine<sub>*P*</sub> (abstracting over “most” physical peculiarities, yet retaining computationally relevant features such as the tape, the read/write head, etc.). At this abstract level of description it shares “computational restrictions”, which are captured by the mathematical structure, with abstract human computers (such as “finitely many different symbols involved in every computation”, “finitely many different states”, etc.). But what *exactly* are the relationships between the formal structure (i.e., Turing machine<sub>*M*</sub>) and the human computer on one hand, and the formal structure and the Turing machine<sub>*P*</sub> on the other? Is the relationship between

mathematical structure and physical machine that of implementation? And if so, is that also true of the relation between mathematical structure and human computer?

The answer to these problems regarding the role and relationship of Turing machines  $M$  with respect to their physical realization and human computational capacities is non-trivial. Even the relationship between the mathematical structure and the device “Turing machine” is not all that clear. How are “computational states” of a Turing machine  $M$  related to states of a Turing machine  $P$ ? The latter, being a physical system, would have to be described in some physical language, e.g., the language of mechanics; yet it is not clear that we would find any physical states in that description that correspond (even if only slightly) to those defined in the abstract machine; in fact, it seems very unlikely to me. Consider, for example, a Turing machine  $P$ , which moves the tape head at a continuous speed in one direction until “its machine table” (however it is realized physically) tells it to move the head in the other direction. Because of inertia, etc. the head does not stop on the current tape square, but only over the next one, then reverses direction and speeds up again. If the tape area under the tape head (i.e., the current position of the tape head) is taken to correspond to “abstract square”, then in this case Turing machine  $P$  would fail to realize Turing machine  $M$ . If this, however, is not taken to correspond to the abstract square, what is? The speed of the tape head plus position? Obviously the answer is not straightforward. And this is only one of the many problems that one could imagine would make it difficult to find a simple state-to-state correspondence between physical states of Turing machine  $P$  and state of the machine table of Turing machine  $M$ .

Therefore, in order to be able to establish a correspondence between both kinds of machines, one would have to abstract over most of the physical properties of a particular Turing machine $\mathcal{P}$  (such as speed, materials, what corresponds to the tape head if neither tape nor head is used, etc.). Note that nothing in the specification of Turing machine $\mathcal{M}$  forces us to be committed to a particular realization of Turing machine $\mathcal{M}$ . In fact, nothing in its specification forces us to be committed to its being a machine in the first place; all there is to Turing machine $\mathcal{M}$  is a mathematical structure of a particular kind (five-tuple, finite sets, etc.). The whole talk about “tape”, “head”, etc. is a way to visualize or motivate what we have already specified at the necessary abstraction (i.e., the mathematical structure). Hence, if a Turing machine $\mathcal{P}$  is to be compared to a Turing machine $\mathcal{M}$ , one can compare them only at exactly this mathematical level of abstraction. This, in turn, requires a way of abstracting from Turing machine $\mathcal{P}$  to get to Turing machine $\mathcal{M}$ , but this is what we were trying to do in the first place.

The situation is not any better for establishing a correspondence between a human computer and a Turing machine $\mathcal{M}$ . Even comparing humans to Turing machines $\mathcal{P}$  presents serious difficulties: “How exactly do tape head movement and printing on tape relate to hand movements and scribbling on scratch paper?” is probably not answerable at all at the level of classical fields. What are “hands” and “scribblings” and “scratch paper” in terms of fields anyway?

Thus, only by involving meta-theoretical considerations about the nature of mechanical possibilities could Turing argue that humans (following rules) and Turing

machine $M$  have the same “computational” limitations.<sup>17</sup> But what about systems that do not obviously behave in such a way as to give rise to mechanical descriptions? Could those systems *implement* Turing machines $M$ ? And what about the physical system “brain”? Does the brain with its various electrical inputs and outputs *implement* a Turing machine $M$ ? Even though the problem of what humans can compute with paper and pencil following rules seems decided, the general problem of what a physical system can compute according to the notion “Turing computable” will be left open if there is no theory that explains how to relate a physical system to a Turing machine $M$ . In other words, as long as there are no criteria that determine what it means to *implement* a Turing machine $M$ ,<sup>18</sup> it will not be clear what a physical system can or does compute. It is perfectly imaginable that there could be physical devices that utilize non-recursive processes if such processes existed (see the first section of chapter 5).

### 2.3 The Standard Account of Implementation

Although there is no general account of what it is to implement a Turing machine $M$ , there are a few explicit, very general suggestions of what it means to implement a computation, where “computation” is defined as anything specified by any of the different equivalent computational formalisms (e.g., see Dietrich 1990, Chalmers 1996, Copeland 1996, et al.). Often, this definition is cast in even more general terms so as to not be forced to make any commitment to a particular notion of computation (or the notion of

---

<sup>17</sup> Note that physical models of Turing machine $M$  are subject to the same kinds of practical constraints that humans are—neither do they have arbitrary amounts of scratch paper/tape nor time at their disposal. Hence, the abstract mathematical structure is an idealization of and theoretical limit for both.

<sup>18</sup> We do not even have such criteria for a Turing machine $P$ —for one, because to my knowledge nobody has attempted to give a rigorous physical description of it.

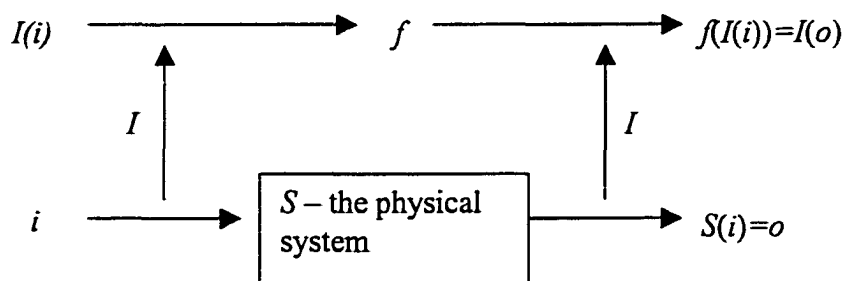
computation at all, for that matter): instead of “the computation implemented by a physical system” one focuses on “the function implemented/realized by a physical system”. Note that the terms “implemented” and “realized” are often used synonymously in this context. Stabler (1987) presents, what could be called the “standard account of what it is to *realize* a function”:

“We require first that the states of the system can be interpreted as representing the elements of the domain and range of the function, and we require that (in certain circumstances) if the system is in a state representing an element of the domain of the function, physical laws guarantee that it will go into a state representing the corresponding element of the range of the function.” (Stabler, 1987)

Formally, this can be written as follows (note that I have renamed variables so as to be consistent with other presentations):

*Definition 2.2:* Let  $S$  be a physical system and  $f$  a function.  $S$  computes  $f$  if, and only if,

1. there is an “interpretation” or “realization” function  $I$  from a set  $PS$  of physical states of the system onto the union of the domain and range of  $f$ , such that
2. physical laws guarantee that (in certain specifiable circumstances) if the system is in a state  $i$  in  $PS$ , then the system will go into state  $o$  (in  $PS$ ) such that  $I(o)=f(I(i))$ .



**Figure 2.1** The standard account of physical realization of a function:  $i$  and  $o$  are physical states of the physical system  $S$ ,  $f$  is the interpretation function that maps these states onto the union of the range and the domain of  $f$ .

This definition is very general in that it includes various other accounts as special cases. Unfortunately, as it stands, the standard account of implementation does not quite work. It has been pointed out that it is “too liberal” as it does not put any restriction on the interpretation function: without any restriction *every system can be viewed as implementing every computation* (the next chapter will explore two arguments in that direction, one by Putnam and one by Searle, in detail to analyze what exactly needs to be constrained). Finding these constraints is crucial to computationalism, because otherwise computationalism loses its explanatory force if everything can be viewed as computing every function.

One *prima facie* difficulty of the standard account is that “terms like ‘state’ and even ‘physical state’ tend to be used very loosely in this sort of context” (Stabler, 1987, p. 3). Stabler demonstrates the potential to “abuse” the standard account by defining a special kind of physical state: assume the behavior  $F$  of a given physical system  $S$  can be described in a physical theory  $P$  (as long as certain background conditions  $C$  obtain that make this description applicable). Suppose further that an infinite sequence of times  $t_1, t_2, t_3, \dots$  is specified. Infinitely many “physical states” can then be specified by stipulating that the system is in state  $p_i$  if and only if it satisfies its description  $F$  at time  $t_i$ . If the  $p_i$  are then taken to be *the computationally relevant states*, the system will “compute” any

function  $f$  over the natural number. Define the interpretation  $I$  (for an arbitrary function  $f$  over the natural numbers) to be

$$I(p_i) = \begin{cases} i/2 & \text{if } i \text{ is 0 or even} \\ f((i-1)/2) & \text{otherwise} \end{cases}$$

To put what happens here into plain English:  $S$  computes  $f$  by going through a sequence of states that are states by virtue of its description  $F$  being true of  $S$  at the respective time (under conditions  $C$ ). In a sense,  $S$  does not really “compute  $f$ ” but rather “enumerates” the pairs  $\langle i, f(i) \rangle$  at any two successive points in time:

“The trick used in this example is obviously to get the object to compute the function by somehow building the function *into* the interpretation function. We could equally well *build the function into* the specification of the computationally relevant states themselves, or into the specification of the circumstances in which the computation takes place. It would be nice to explain exactly how these sorts of tricks can be excluded, but this problem is hard and fortunately beyond the scope of this paper.” (Stabler, 1987, p. 4, emphases are mine)

To see what needs to be done in order to “exclude tricks of this sort”, one needs to analyze different kinds of “tricks” and hope to be able to detect “common patterns”—this will be the task of the two following chapters. It is clear that there must be “some restrictions on the interpretation function used in any empirically substantial computational claim” (Stabler, p. 4). For now it suffices to point to one obvious problem with the above definition, namely that states are picked out by particular times (since it was a straightforward way to obtain infinitely many computational states that could

correspond to the infinitely many pairs of natural numbers that define  $f$ ). What if the computationally relevant states have to be “somehow extracted” from a physical description of the system?

“This raises a problem for the idea that an ordinary calculator might realize the addition function on the natural numbers. The problem is that the addition function is infinite (in the sense that it has an infinite domain): any pair of numbers can be added, and there are infinitely many different pairs. To realize the addition function, then, a physical system would have to have infinitely many computationally relevant physical states (since our interpretation  $I$  maps a set of states *onto* the union of the domain and range of the function). Obviously, the states which are commonly regarded as the relevant states of the calculator are not infinite in number.” (Stabler, 1987, p. 5)

Besides the question, whether physical systems can realize infinite functions at all (e.g., using infinitely many states), it seems that one has to account at least for cases like the calculator and revise the standard account to allow it to realize an infinite function *using finitely many states*. The idea is that infinitely many computational states are not needed to realize an infinite function if each argument of the function corresponds to a finite sequence of computational states. In other words, each natural number (that is an argument of  $f$ ) would be “represented” by an input sequence of a finite number of computational states, and by the same token, each value of  $f$  for a given argument would correspond to an output sequence of finitely many states. The revised standard account, which allows a system with only finitely many (computationally relevant) states to realize an infinite function is presented by Stabler as follows:



*Definition 2.3:* [Revised standard account] Let  $S$  be a physical system and  $f$  a function.  $S$  computes  $f$  if, and only if,

1. there is an “interpretation” or “realization” function  $In$ , which maps a set of finite sequences of physical “input” states of the system onto the domain of  $f$ , and an interpretation function  $Out$ , which maps a set of finite sequences of physical “output” states onto the range of  $f$
2. physical laws guarantee that (in certain circumstances  $C$ ) if the system goes successively through the states of an input sequence  $seq_i$ , it will go successively through the states of the corresponding output sequence  $seq_o$  where  $Out(seq_o) = f(In(seq_i))$ .

Note that the revised standard account has made two crucial transitions: an explicit transition from (potentially) infinite sets of physical states to finite such sets, and another *tacit* transition from “physical states of the system” to sets of “physical input states of the system” and “physical output states of the system”. This is especially noteworthy as the latter account implicitly excludes so-called “inner states” of the system (which the former implicitly included): only the input-output mapping matters in the revised standard account as succinctly expressed by the formula  $Out(seq_o) = f(In(seq_i))$ . Put differently, the physical system is considered a black box, whose “inner mechanisms/workings” (as described by a physical theory) are abstracted over. Thus, while the revised standard account can answer the “what” question (i.e., “what function does a physical system realize”), it will not be able to answer the “how” question (i.e., “how does a physical system realize the function”), because it only took input and output states from the

physical theory (in which the system is described) ignoring the rest. So, if one wants to account for the “how” question as well, more of the physical description of the system (than merely sets of input and output states) needs to be retained in the definition of “physical realization of a function”.

It is interesting to compare definition 2.3 to Dietrich’s account of what it means to compute a function:

*Definition 2.4:* [Dietrich] Let  $S$  be a physical system and  $f$  a function. The computation of  $f$  can be attributed to  $S$  successfully when it is possible to explain  $S$ ’s computation of  $f$  in terms of a sequence of functions  $\langle g_1, \dots, g_n \rangle$  ( $n > 1$ ) such that:

1.  $f = g_n \circ g_{n-1} \circ \dots \circ g_1$
2. the sequence  $\langle g_1, \dots, g_n \rangle$  is productive
3.  $S$  passes through a sequence of states each of which corresponds via an interpretation function  $I$  to either the domain or the range of one of the  $g_i$ ’s, and each state between the first and final states is the range of some  $g_i$  and the domain of some  $g_{i+1}$
4. the  $g_i$ ’s are antecedently understood

And Dietrich adds that “when  $f = g_n \circ g_{n-1} \circ \dots \circ g_1$  and the  $g_i$ ’s are non-trivial, it is natural to say that the sequence of functions  $\langle g_1, \dots, g_n \rangle$  analyzes the computation of  $f$  by  $S$  and explains the capacity of  $S$  to compute  $f$ .”

While both definitions involve “finite sequences of computational states”, they involve the notion for different purposes. In definition 2.3 sequences of computational states are used to “represent” numbers (from a countably infinite domain). These sequences are the input and output to the physical system, while in definition 2.4 it is not clear what would count as input and output, since neither of these terms is explicitly mentioned. It seems implied, however, that inputs and outputs are simply taken to be “computational states”, since “*S* passes through a sequence of states *each of which corresponds via an interpretation function I to either the domain or the range of one of the  $g_i$ 's*”. If this is so, then Dietrich’s account is susceptible to the same kind of objection as the standard account of definition 2.2. The only difference seems to be, then, that while the standard account was silent about the “internal structure” of the computation, about “how the system realized the function”, Dietrich’s version requires the system to go through a finite sequence of steps, each of which is again determined by some function. There seems to be a similarity between what Dietrich had in mind and Cummins’ idea of step satisfaction (see Cummins, 1989), as each function  $g_i$  could correspond to an “atomic step” in the algorithm defined by  $\langle g_1, \dots, g_n \rangle$  and executed by the system (which reflects his requirement of the productivity of computations).

Assuming that there are only finitely many relevant computational states in a system (as in definition 2.4), an additional criterion has to be added to Dietrich’s definition, namely that all  $g_i$ ’s be finite. This, in turn, implies that  $f$  has to be finite too. Therefore, Dietrich’s definition would have to be altered in order to allow for sequences of input and output states as in the standard account of implementation (which very likely leads to

modifications of other parts of his definition, in particular the sequence of functions the system has to go through).<sup>19</sup>

There are other modifications to definition 2.3 that become necessary upon further analysis. It has been pointed out by Kripke (1981), for example, that a physical machine can only “approximately” or “imperfectly” realize an infinite function, for one, because time is limited (“it will run out of time”) and also because it will make errors (“nothing is perfect”):

“ [...] the machine is a finite object, accepting only finitely many numbers as inputs and yielding only finitely many as outputs—others are simply too big. Indefinitely many programs extend the actual behavior of the machine. Usually, this is just ignored because the designer of the machine intended it to fulfill just one program [...] Second, in practice it is hardly likely that I really intend to entrust the values of a function to the operation of a physical machine, even for that part of the function for which the machine can operate. Actual machines can malfunction: through melting wires or slipping gears they may give the wrong answer. How is it determined when a malfunction occurs? By reference to the program of the machine, as intended by its designer, not simply by reference to the machine itself.” (Kripke, 1981, p. 33-35)

Kripke’s objection to the general idea of a physical system realizing an infinite function rests on the claim that “indefinitely many programs extend the actual behavior of the machine” and that reference to intention of the designer (of a computing device) is needed to fix one particular program. To counter objections of that sort, Stabler suggests

---

<sup>19</sup> While it seems to me that Dietrich might have wanted to reflect the idea of functional composition of “recursive functions” in his definition of implementation, it is not at all clear how one would translate such a sequence of formally defined functions into the processes that happen (possibly simultaneously) in a real computer. In particular, I fail to see the reason why one would impose such a structure on a computational device (as there are certainly many different ways of implementing a function obtained by functional composition).

modifying part 2 of definition 2.3 by adding the counterfactual clause “if the system satisfied conditions of normal operation  $N$  for long enough”:

2'. Physical laws guarantee that (in certain circumstances  $C$  and *if the system satisfied conditions of normal operation  $N$  for long enough*) if the system goes successively through the states of an input sequence  $seq_i$ , it will go successively through the states of the corresponding output sequence  $seq_o$  where  $Out(seq_o) = f(In(seq_i))$ .

By adding this clause, reference to the designer's intentions is replaced by conditions of normal operation, thus avoiding the problems that the former raises for the functionalist (see Stabler, 1987, p. 19). Stabler, being aware that introducing counterfactuals can cause more (additional) problems than they can solve (e.g., see Putnam's discussion of Lewis' notion of causation in his 1988, and also chapter 3), argues that the kinds of counterfactuals involved in the conditions of normal operation are supported by physical laws:

“The counterfactuals needed by such accounts [e.g. realization of simple computations such as the identity function realized by a wire] are of a sort that can be supported by our understanding of physical laws. There is no reason to suppose that the more complicated counterfactuals needed to support claims about the functioning of organisms will be different in kind. [...] we need only consider what would be the case if the antecedents of our counterfactuals held, and as in our examples [e.g., the wire example mentioned above] sometimes our understanding of physical laws can guide us quite clearly in that consideration.” (Stabler, 1987, p.18, remarks in brackets are mine)

Although it would be preferable to dispense with counterfactuals in an account of physical realization altogether, it seems that counterfactuals will always enter the picture if a notion of “*normal operation*” is involved (e.g., “if the system had been in condition  $C$  at time  $t$ , then...”). And such a notion of normal operation seems to take care of problems regarding potential malfunctions of a system (i.e., malfunctions with respect to the program that was supposed to describe the systems behavior, as noted by Kripke).

It is not quite clear, however, if adding conditions of normal operations can actually solve Kripke’s first objection, namely that finite systems cannot realize infinite functions. Stabler’s strategy to determine which infinite function a system realizes is to “counterfactually extend the life-time of a physical system”: if the system satisfied conditions of normal operation  $N$  for long enough, then it would be able to compute all pairs of a (particular) infinite function  $f$ . This seems to me to miss the point, because it seems to conflate and equivocate the notions of “computing a function” and what I will later call “realizing a function”: consider all input and output sequences of length 1 in clause 2’ above. These sequence will define a function  $g$  from input states to output states, call it “the function realized by the system”. This function is *necessarily finite*, as the set of input states  $In$  and the set of output states  $Out$  are finite. According to clause 2’ it is this function realized by the system that can be used to define another infinite function  $f$ , whose arguments are sequences of arguments of  $f$ , called “the function computed by the system”. While Stabler argues that the system, according to 2’, can be seen to compute  $f$ , I believe that Kripke’s objection actually concerns the function “realized by the system”  $g$ : in his view, I take it, of no such function  $g$  can it be said that it is infinite or that it can be extended to a particular infinite function (without making

reference to design issues, etc.). If this is so, is a tricky question that hinges upon one's view of how physical systems are best described. I will return to this issue in chapter 5.<sup>20</sup>

In any case, the modified standard account of implementation 2.3. + 2', as it stands, seem to be a good entree into the discussion of what it means for a physical system to realize a computation. To get an idea of the theoretical weight of its ingredients, I shall list a summary of the entities involved in and used by the standard account:

1. the function  $f$  (to be realized)
2. physical input and physical output states
3. the input mapping  $In$  (from finite sequences of input states *onto* the domain of  $f$ )
4. the output mapping  $Out$  (from finite sequences of output states *onto* the range of  $f$ )
5. the circumstances  $C$
6. the conditions of normal operation  $N$

Each of the six items has its own virtues and "flaws" (that is, difficulties connected with it). While I will have to say more about the last two (i.e., the circumstances  $C$  and the conditions of normal operations  $N$ ) in chapter 5, I shall briefly point out the reason why the remaining four ingredients add a lot of spice to the discussion.

First of all, take the function  $f$  and ask what class of functions "possible"  $f$ s belong to. In other words, is there an analytic definition of the class of functions realized by physical systems (i.e. *the class of physically realizable functions*)? Not only would such a

---

<sup>20</sup>In chapter 5, I will start with descriptions of physical systems that views them as realizing infinite functions and then, by incorporating practical constraints end up with descriptions that view physical systems as realizing only finite

characterization be theoretical desirable, but practically it would probably be one of the strongest “limiting results” in computer science. Obviously this question cannot be decided at a mathematical level alone as it intrinsically involves descriptions of (all possible) physical systems in (actual and potential) physical theories.<sup>21</sup> However, as already pointed out in section 2, it is commonly believed that the class of “physically realizable functions” coincides with the class of recursive functions (if it is not a proper subset altogether). Yet, there are no arguments for this claim (to my knowledge) that involve notions of “actual or potential physical theory” (even Gandy’s fourth principle requiring “local effects” has at best “a touch of relativity”)<sup>22</sup>.

Regardless of whether this question could ever be decided, it will still be necessary to add an additional parameter  $P$  (for physical theory) to the standard account (i.e., “Let  $S$  be a physical system *described in*  $P$  and  $f$  a function...”) to account for the physical theory in which input and output states are (and, more generally, the system itself is) described.

The next three items constitute the heart of most views of implementation as physical realization: physical states are connected to more abstract states by input/output mappings, where these states could either be computational states, or as in the above case simply numbers (or any abstract object, for that matter). It is particularly important to understand how we arrived at this basic structure of the notion of “realization of a function” in order to be able to understand the arguments raised against it. What has led

---

functions.

<sup>21</sup> I used the term potential physical theory in the sense of “physical theory that does explanatory work” (i.e., not any arbitrary physical theory that might not be able to predict anything).

<sup>22</sup>Gandy’s (1980) fourth principle of “computational systems” has a touch of relativity: it is supposed to express the fact that effects only propagate at certain speeds (at most at the speed of light). Thus, effects in any system are restricted to certain “causal neighborhoods” for any given time (those to which they can propagate). See also the first section of chapter 5.



us to modify the (first version of the) standard account (definition 2.2) was the notion of “(relevant) computational states of the system”: the counterexample to definition 2.2 exploited the fact that no constraints were imposed on either physical states or the interpretation of them for that matter. We avoided such unconstrained formations of “physical states” and interpretations thereof, by switching from “physical states” (that is, something like “inner state of the system”) to “input and output states” and replacing the one mapping in the previous account by two *finite* input and output mappings. What the counterexample to the first version of the standard account, thus, hinted at are two potential sources of danger that allow for arbitrary computations:

1. Unconstrained physical states
2. Unconstrained mappings (*In* and *Out*) from physical to computational states

Although the revised account avoids the obvious problems of its predecessor, there are still ways to ridicule it (as will be shown in the next chapter). Accordingly, there are the following questions that need to be answered by any such view of implementation: What is the nature of these input/output mappings? How are these mappings constrained? What physical states are (relevant) computational states? These questions, in turn, presuppose answers to question about the nature of input/output states, or more generally, *the nature of physical states*. After all this is what “the bridge” is all about—connecting computational and physical states. Depending on the physical theory under scrutiny, the term “physical state” can denote very different concepts (e.g. physical states in classical physics are different from those in quantum physics). The same is true of

computational states; being states of computations, their nature is dependent on the respective notion of computation (thus, computational states of a C-program, for example, are different from states in a MARKOV algorithm). Furthermore, computational states seem to be very different from physical states, as the former can be specified quite abstractly *without* reference to real-time and/or physical dimensions.<sup>23</sup>

Since the notion of physical state will play a crucial role in determining the computation realized in a physical system, and because the two main arguments against the “standard account of implementation” under scrutiny in the next chapters in my view essentially challenge the notion of physical state (in addition to notions of computation), I will attend to the notion of physical state first, and then briefly turn to computational states (the role of the mappings between computational states and physical states will be studied in detail in the next chapters).

## 2.4 Physical States and Physical Descriptions

I shall start defining the notion of “physical state” by pointing out that one needs to distinguish between “classical” and “non-classical” physical theories, as the latter ones, such as quantum mechanics for example, use a different notion of state from that of the former ones (see, for example, Messiah, 1961). I will restrict myself to classical physics, as the effects that require quantization (of waves, for example) can only be obtained at

---

<sup>23</sup> Compare this to yet another kind of state that *prima facie* differs from both of the previous ones, mental states, that is, states of minds that can have content and can be “about” something—whether these states are or correspond to computational states, and whether computational states are or correspond to physical states is exactly what computationalism seeks to answer.

very small scales, and I will simply assume that these levels of description are not relevant for cognition.<sup>24</sup>

Assuming classical physics, thus leaves us with the “standard definition of physical state of a system” as the value of each variable of a mathematical system of equations at a given time: dependent variables range over different physical dimensions and time is the only independent variable. Put differently, the values of all other variables are always considered with respect to changes over time. In classical mechanics, for example, there would be two kinds of variables, one kind for location and one for impulse, for each spatial dimension.

The kind of “mathematical system”, *the system of equations* (better: set or list of equations) used to describe the behavior of a physical system, is called *dynamical system*. It is a collection of mathematical equations, either differential or difference equations, depending on whether time is taken to be discrete or real-valued. Independent of the assumptions on time, the values of the variables for each relevant physical dimension can also either be discrete or real-valued. So there is a total of four combinations, all of which are possible: all variables can be either discrete or continuous, but it is also possible to have a system with discrete variables for physical dimensions, yet a real variable for time. And the fourth option is to have time discrete, but all variables for physical dimensions continuous. Which option is used depends solely on the kind of

---

<sup>24</sup> I do not want to exclude these quantum level of description *a priori*, as there is a possibility that they might actually play a crucial role in cognition (see for example Penrose 90, 94). However, it would seem rather absurd to me that any fault-tolerant biological system would rely on quantum effects that could be easily disturbed. In other words, it seems to me that in order to be a robust, reliable, fault-tolerant system extremely small effects that are even minimal compared to regular signal noise would have to be ignored and dependence on them avoided, at least that is what engineers attempt when they try to design reliable systems—but maybe nature came up with a robust and reliable way

physical system the behavior of which one wants to describe as well as pragmatic assumptions and constraints (such as “exactness of measurements”, “required degree of precision”, etc.).

It is the mathematical apparatus of differential equations that is used to describe the behavior of most physical systems. For example, classical mechanics requires its variables to be real-valued. It is important to note, however, that nothing intrinsically hinges upon it. A dynamical system can be a discrete system, if difference equations are used instead of differential equations, and for many systems of differential equations one can generate a system of difference equations and vice versa (e.g., see Yorke et al., 1996).

So, to describe a physical system, one needs to introduce a variable for each relevant physical dimension and consider it as a function of time. The simplest way to specify the behavior of the physical system would be to provide graphs of each such variable over time, that is, to have a set of functions  $X_1(t), X_2(t), \dots, X_n(t)$  where  $X_i(t)$  yields the “state” of the relevant physical dimension  $X_i$  at time  $t$ . This set of functions will determine the behavior of the system for all times. However, it does not reveal the possible dependencies of the  $X_i$  on each other. This is where differential equations come in handy. They provide a way of specifying the various interdependencies of different variables in such a way that graphs of each variable can be obtained from them, yet the interdependencies are also brought to the open. The nature of these interdependencies will become a crucial factor in an explanation of the behavior of the system.<sup>25</sup>

---

of utilizing quantum effects, such as certain non-locality effects, quantum superposition, etc.... in that case the following discussion would have to be augmented by a non-classical notion of physical state.

<sup>25</sup> The mathematical theory of dynamical systems seems well-suited to describe quantitatively systems that exhibit what Clark (1997, 1998) calls “continuous reciprocal causation”.

Every real-world system that involves change can potentially be modeled by a dynamical system—this is what dynamic systems have been designed to do. According to the respective system, this will happen at different levels of description, at the very low level of fields (take Maxwell’s equations), or the very high level of human decision making (take Busemeyer and Townsend’s decision field theory). Dynamic systems are not committed to any particular physical quality either (the same way they are not committed to the discrete-continuous distinction). They are not committed to particular notions of physical states, nor are they committed to a realist or instrumentalist interpretation of a theory’s entities. Whatever changes over time, can be modeled, and it does not even have to be time, because in case of difference equations all that matters is order!

Although dynamic systems are part of a mathematical theory, their usefulness and applicability is not decided within mathematics, but by engineers who use the mathematical apparatus to describe and model the behavior of physical systems; or as Kutz (1998, p. 796) put it: “A mathematical model [of a physical system] is a description of a system in terms of equations”.

The following is a synopsis of Kutz’ (1998) article on “mathematical models of dynamical physical systems” in the *Mechanical engineer’s handbook*. It will shed light on two issues that are of crucial importance to the current enterprise: 1) the nature of physical states the way engineers see them, and 2) the nature of descriptions of physical systems used by people that need to accomplish tasks with these systems (i.e., who need to achieve certain practical results). The reason why this is so crucial is that I will try to defend the position that in relating computations to physical systems we need to

concentrate on notions actually used to achieve practical goals, i.e., on the engineers' notions of physical state (instead of some removed philosophical notions).

First and foremost to modeling physical systems is the idea that what is common to all of them is that they transform, store, and/or consume energy over time (a fact often neglected in philosophical discussions):

“Differential equations describing the dynamic behavior of a physical system are derived by applying the appropriate physical laws. *These laws reflect the ways in which energy can be stored and transferred in the system.* Because of the common physical basis provided by the concept of energy, a general approach to deriving differential equations models is possible. This approach applies equally well to mechanical, electrical, fluid, and thermal systems, and is particularly useful for systems that are combinations of these physical types.” (Kutz, 1998, p. 796, emphasis is mine)

We assume that there are basically two kinds of physical systems, those with only one terminal, and those with at least two terminals (where a terminal is a place where energy can enter and/or leave the system in whatever form). Accordingly, two kinds of variables (in the mathematical model) have to be distinguished: *through* (for one terminal systems) and *across* variables (for two terminal systems). Across variables stand for “the difference in state” (to be explained shortly) between two terminals, through variables stand for the change of physical magnitude. Examples for through variables are the force through a spring or the current flow through a resistor. Examples for across variables are the velocity difference of the two terminals in a spring or the voltage drop in a resistor. Given this distinction, one can compute the flow of power  $P(t)$  into a physical system through both terminals as the product of the through variable  $f(t)$  and the difference

between across variables  $v_1(t)$  and  $v_2(t)$ , i.e.,  $P=f(v_1-v_2)$  (negative values mean that power flows out of the system). The energy transferred to the element between terminal  $t_a$  and  $t_b$  is thus:

$$E = \int_{t_a}^{t_b} P dt = \int_{t_a}^{t_b} f(v_1 - v_2) dt.$$

“Physical devices are *represented by idealized system elements*, or by combinations of these elements. The behavior of a one-port (two terminal) element expresses the relationship between the physical variables for that element and is defined mathematically by a *constitutive relationship (which is derived empirically by experimentation rather than from any more fundamental principles)*. The element law, derived from the constitutive relationship, describes the behavior of an element in terms of across and through variables and is the form most commonly used to derive mathematical models.” (Kutz, 1998, p. 798, emphases mine)

So, (mathematical) dynamical models are used to model simple and complex physical systems. They assume “ideal system elements”, whose “element laws are defined by a constitutive relationship derived empirically by experimentation rather than from any more fundamental principles”. This statement cannot be stressed enough: it is the engineer who by virtue of measurements relates different physical magnitudes and dimensions, and then formulates laws that govern the behavior of idealized elements. It is important to keep this abstraction step in mind. The mathematical model, which describes the physical system, has already built into it an “idealization”, which might or might not lead to a misrepresentation of the behavior/functionality of the system. At this

point the “first implementation problem” already pops up, as one could ask how exactly the physical system “implements” the “idealized element”...<sup>26</sup>

A special form of the equations of a dynamic system, the so-called “input-output form” (or I/O form for short), is particularly useful to describing the interaction of physical systems with their environments. In these systems,

“inputs correspond to [energy] sources and *are assumed to be known functions of time.*

Outputs correspond to physical variables that are to be measured or calculated. [...] I/O differential equations are obtained by combining element laws and continuity and compatibility equations in order to eliminate all variables except input and output” (op. cit. p. 808, emphasis is mine).

Let  $I_1(t), \dots, I_n(t)$  be the different inputs to a system and  $O_1(t), \dots, O_m(t)$  the different outputs of that system. Then the I/O form can be described by  $m$  equations using  $m$  different  $n$ -place functions  $F_1(x_1, \dots, x_n), \dots, F_m(x_1, \dots, x_n)$  (which have to be continuous):

$$\begin{aligned} O_1(t) &= F_1(I_1(t), \dots, I_n(t)) \\ O_2(t) &= F_2(I_1(t), \dots, I_n(t)) \\ &\dots \quad \dots \\ O_m(t) &= F_m(I_1(t), \dots, I_n(t)) \end{aligned}$$

Take, for example, electrons in a copper wire. Their behavior depends on the difference of potential between the two ends of the wire. If the potential differs, electrons will start to move, current will flow. According to Ohm’s law the flow of current will be a function of the resistance of the wire and the potential difference between its terminals. These influences are the inputs to the system (and if the system can affect another system

---

<sup>26</sup> One does not have to look at computational devices and their abstract descriptions (which are even further “removed” from reality) to formulate “physical realization”. In fact, it will turn out (in chapter 5) that the notion of



by virtue of the same or another physical dimension, these will be the outputs). The way inputs and outputs are related is thus given by physical laws, whereas the way the system actually behaves over a period of time is given by functions of the input and output dimensions over time. It is important to appreciate the difference between 1) the statement that certain relations obtain between different physical magnitudes and dimensions in a physical system and 2) a description of the evolution of that system over time: the first kind of statement makes a nomological claim which is independent of actual processes and real-world embodiment as described by the second.

Consider a mass point, for example, for which the forces that apply equal its mass times its acceleration. This equations relates (local) force, acceleration and mass of the mass point. However, this does not say much about the *actual behavior* of a *particular mass point*. An additional function is required to describe the behavior of the mass point over time. Let  $x(t)$  be such a function describing the location of the mass point as a function of time. Then together with the knowledge that the first derivative is the velocity of the mass point and the second derivative is its acceleration, one can obtain a function that describes the forces exerted to the mass point over time (given the mass of the point, too). Or reversed, given the forces over time, one can determine the path of the mass point.

Two components figure, therefore, in the description of the behavior of a physical system: first, equations relating certain physical magnitudes of a physical system (such as the mass point above) in a “timeless” manner—the physical laws—and second, functions

---

(physical) realization of a computation is only at the end of a chain of increasingly abstract descriptions of a physical system, for each of which one can formulate a corresponding “implementation problem”.

that describe the temporal behavior of some of these magnitudes (from which the temporal behavior of the others can be deduced using the previously specified laws).<sup>27</sup>

While the I/O-form of differential equations solely relates inputs to and outputs of a physical system, it is sometimes desirable to consider *the flow of energy within the system* in addition. This can be achieved by introducing so-called “natural inner states”, which correspond to energy sources and sinks of a system. The modified version of the I/O-form looks like this:

$$O_1(t) = F_1(I_1(t), \dots, I_n(t), P_1(t), \dots, P_k(t))$$

$$O_2(t) = F_2(I_1(t), \dots, I_n(t), P_1(t), \dots, P_k(t))$$

... ..

$$O_m(t) = F_m(I_1(t), \dots, I_n(t), P_1(t), \dots, P_k(t))$$

$$P_1(t) = F_{m+1}(I_1(t), \dots, I_n(t), P_1(t), \dots, P_k(t))$$

$$P_2(t) = F_{m+2}(I_1(t), \dots, I_n(t), P_1(t), \dots, P_k(t))$$

... ..

$$P_k(t) = F_{m+k}(I_1(t), \dots, I_n(t), P_1(t), \dots, P_k(t))$$

where the  $P_i(t)$  are “inner states”. Mathematically speaking, there are infinitely many different sets of equations of finitely many different inner states that give rise to the same I/O form (i.e., that describe the same I/O behavior of a physical system).<sup>28</sup> Therefore, it

---

<sup>27</sup> Note that if all relevant physical magnitudes of a system are given as functions over time, no “additional” law is needed to describe the system’s behavior, as all relevant information is already hidden in these complete descriptions. Thus, some physical laws (such as Newton’s second law) are special forms of differential equations that “abstract” over behaviors of particular physical objects. However, it might not be a trivial task (if possible at all) to “reconstruct” laws from these I/O-functions.

<sup>28</sup> Larry Moss in personal communication.

is crucial to support a particular choice of inner states by practical considerations if one is interested in deriving “the flow of energy” within the system, as there will be dynamical systems, which describe the behavior of the system perfectly without having a single inner state correspond to an energy source or sink.

Besides energy sources and energy sinks, another “natural choice” for inner states are the states of the connections between parts of the system. Thus, a division of the system into “interacting parts” result in a natural description of inner states; and vice versa, a set of inner states and their values can partition the system according to the loci which are used to record, measure, etc., in short, *define* these inner states. But again, as pointed out before, these choices depend on pragmatic considerations; theory alone is not enough to justify them, let alone to individuate the so-chosen states. It is crucial to keep this in mind with respect to computationalist explanations: if it is claimed that computation somehow “mirrors” the causal structure of a system, and if this mirroring function is established by setting up a correspondence between computational and physical states, it is crucial to understand that this can only be claimed *relative to the choice of the physical states*. If physical states are given and their choice is (pragmatically) justified (as in the case of energy sources and energy sinks), computations will be able to reflect parts of the causal organization of the physical system (as they only abstract over physical dimensions, while preserving the temporal order and the quantity of spatial locations—this will be explored in chapter 5). However, if physical states are *not* previously defined, as already pointed out in section 3, one can always define inner states of a system such

that the system realizes any computational architecture as long as it preserves the input-output mapping.<sup>29</sup>

To summarize what has been said about physical states: first and foremost, one has to distinguish two different notions, 1) the actual make-up of a physical system at a given time, and 2) the value of all variables of the mathematical model of a physical system (e.g., the values of the variables of a dynamic system) at a given time. The relation between the former and the latter is that of idealization, as the latter is derived from experiments and measurements making certain idealizing assumptions about the elements of the system. A particular form of the mathematical model, the I/O form, made it possible to model energy flow through the system. In particular, modifications to the I/O form allowed for the introduction of “inner states” of the system (e.g., states that correspond to energy source or energy sinks). However, no unique description of “inner states” could be derived from the I/O form itself: infinitely many different dynamical systems with different inner states exhibit the same input-output behavior. Therefore, if inner states are to be a crucial part of a mathematical model of a physical system (e.g., because one wants to relate them to “functional states”), they have to be justified pragmatically.

---

<sup>29</sup> For example, one could argue that a PC computing the factorial function is really a MAC computing the factorial function: the PC gets the input-output function right, and as far as inner states are concerned, one choose the ones relevant for MACs. Note, however, that this reasoning breaks down immediately, once one opens the PC and measures physical magnitudes at particular locations—the notion of “disassembling of a system” seems to play a crucial role in this argument (see Dennett, 1986).

## 2.5 Computational States and Computational Descriptions

Computational states—to put it provocatively—are “nothing more than abstractions” of certain physical states of idealized physical systems from an practical point of view. What I mean by this can be explained using a von Neumann-CPU as an example: to say that the CPU is “updating its address register” is to say that certain conditions obtain in the physical CPU in sequence (that is, over real-time) that allow for a certain change of states in a particular spatial region of the CPU. Because this level of description (if all the physical details were filled in) is not only tedious, but also prevents one from expressing truths about all CPUs of a particular kind (regardless of their spatial locations and physical manifestations), an abstraction step, i.e., the shift to a more abstract description is very useful. At this level of description, one ignores certain physical properties such as voltages, transistors and their chemical make-up, etc. (those properties that are *not relevant* to describing the computation) and talks instead of bits and arrangements of bits. What is retained is a correspondence between the number of bits and the number of physical locations at which a state change can occur. But whereas arrangements of bits are described using  $n$ -tuples of bits, temporo-spatial regions would have to be described using a much more complicated and complex formal apparatus (e.g., using a set of values of all field parameters for the given spatial region during the given time-interval). All of this ballast is not necessary, however, if one only wants to understand the change in the address-register (i.e., the arrangement of bits) in the CPU. Therefore, while the physical description does employ a notion of physical state that is defined by the physical theory used to describe the CPU (e.g., the theory of electrical

fields), the notion of “computational state” is simply the state of the arrangement of bits, i.e., that which has been distilled from the physical description: the values of the components of the vector. Contrary to the values of the variables in the physical theory, which denote certain physical conditions (in that they, for example, express magnitudes and orientations of vectors of a each spatial point in a field), values of bits do not correspond to anything, except that they are abstractions of these other physical values. Computer scientists and computer engineers have introduced these abstractions because the only important property of the underlying physical states with respect to the computational capacity of the system is that a particular spatial region can be in either of two (sets of) states at a given time (the system has been particularly designed to have this property). In short, to simplify matters and to allow one to generalize over different physical realizations, the notion “bit” is used to describe particular physical states of particular physical systems.

While computational states can be seen as abstractions of certain physical states if the physical system is given, it is not clear how one would go from computational states back to physical ones. Thus, if one asks whether a particular system “satisfies” a given computational description (“implements a computation”), one would have to check, if starting with the physical description of the physical system by abstracting over various physical properties one could arrive at the computational description<sup>30</sup>—whether the other route is also possible, i.e., going from computational to physical descriptions directly, will be considered in the next chapters. For now may it suffice to note that some

---

<sup>30</sup> Note that it is not *prima facie* clear that there is a “generic way” of abstracting over physical dimensions in a systematic manner to obtain computational descriptions.

computational states (the ones defined for computers we build) are obviously abstractions of certain physical states. Whether *all computational states* are abstractions of physical states, and whether it is possible to tell if a given physical system implements a given computation *is exactly* the “implementation problem” in the philosophy of mind and cognitive science as well as the foundations of computer science. Before a solution of this problem can be attempted, an analysis of those arguments is necessary that intend to show how any computational state can be obtained from a physical system if physical states are defined in a “malicious” way.

## Chapter 3:

### CCM's Convinced Critics

#### 3.1 Attacking Computationalism: Everything Computes!

Computationalism has recently come under heavy attack from various directions. Some (especially adherents of connectionism and “dynamic systems people”) hold that mind simply cannot be explained in terms of computation. Others believe that crucial aspects of the intuitive notion of computation are still not well understood and, hence, not reflected in (formal) definitions of computation. Whereas the former position excludes computational explanations of mind *a priori*, the latter insist on a *re-evaluation* of the notion of computation and makes value of the notion of computation for an explanation of mind dependent on it.

The reasons for excluding the notion of computation from cognitive explanations vary significantly (e.g., see the various articles in Port and van Gelder, 1995). However, there is a common theme to two of the most significant philosophical attacks advanced by Searle (1992) and Putnam (1988): they locate the most severe shortcomings of the computationalist position in a particular kind of explanatory gap—standard notions of computation (such as “Turing-computability”) leave open exactly *how* abstract “computations” are linked to concrete physical systems realizing them. Searle claims that (under the received notion of implementation) walls “implement” the Wordstar program and Putnam shows that every ordinary open system “realizes” every finite state



automaton. Consequently, the validity of the intuitive account of implementation, in particular once it is taken out of its computational context and applied to a wider class of systems (that is, natural systems), must be seriously doubted.

The strategy pursued by Searle is to argue that the notion of “implementation of a program” is *intrinsically observer-relative* and that one cannot derive a syntactic description from a physical description. Thus, in his polemic, every object can be *assigned* a computational description under which that object can be viewed as implementing any program.

A quite different strategy is taken on by Putnam, who is concerned with the notion of “realization of a FSA”. While Searle’s line of argument presumes a specification of a computational architecture on which the program (to be implemented) can “run”, Putnam only needs to establish a relationship between a physical system and a finite state automaton, which exhibits certain properties (e.g., that state transitions are preserved). *In nuce*, he shows how physical state types can be defined for any physical system such that it realizes every finite state automaton.

Both attacks seem to tackle different aspects of the notion of implementation—the first seems to get at the problem *that* every physical system can be “interpreted” as being a computer, that is, that every physical system can be viewed as supporting a computational architecture (on which the program to be implemented can run), whereas the second seems to show *how* any physical system can implement every instances of a particular computational formalism (that of finite state automata). To put the two lines of argument into slogans, the first attack seems to be a “semantic view”, as it involves the

notion of “interpretation”, whereas the other seems to be a “correspondence view”, as it sets up a correspondence between physical and computational states.

The following sections will first present Putnam’s argument, as this seems to be the less general of the two, then (Copeland’s reconstruction of) Searle’s views will be analyzed.

### **3.2 Putnam’s Realization Theorem**

Hilary Putnam is commonly held to be the originator of functionalism, the view that mental states can be individuated and distinguished according to the functional role that they fulfill (in the overall architecture). His well-known multiple realization argument (Putnam, 1967) showed that type identity (between mental and physical state types) was too strong a requirement: the same mental state might be realized in different physical ways, which could have nothing in common at the physical level besides realizing the same mental state.

These considerations led him to the level of functional description of the mental inspired by the Turing machine formalism. Later, he abandoned his conviction that the psychology of an organism could be understood in terms of Turing machines. Mental states were not viewed as computational, but rather as *terms* in a “psychological theory”, which bear certain relations to their physical manifestations. Given the notion of “functional isomorphism”, a relation that holds between two systems if there is a mapping from states of the one onto states of the other which makes both systems isomorphic models of the psychological theory under consideration, Putnam’s credo became:

"[...] that all mental states (propositional attitudes, experiences, etc.) are preserved under functional isomorphism. It was built into this proposal [...] that something is a 'model' of a psychological theory only if it has *nonpsychological*—physical or computational, or whatever—states which are related as the psychological theory *says* the mental states are related." (Putnam, 1988, p. 99)

In his book *Representation and Reality*, Putnam (1988, especially in chapter 5) launched a severe attack on functionalism arguing that mental states cannot be viewed as mere computational descriptions, not even a combination of physical and computational states would suffice to account for the nature of mental states. Furthermore, his objections extend to "physical realizations of a psychological theory", a notion that can be defined as follows:

*Definition 3.1:* A physical system  $S$  realizes a psychological theory  $T$  (in the language  $L = \langle Q, \rightarrow, \dots \rangle$  where  $Q$  is a finite set of "theoretical terms", call them " $T$ -terms", standing for mental states, ' $\rightarrow$ ' is the theoretical pendant to "causes", and " $\dots$ " indicates additional primitives) if there exists a 1-1 mapping  $f$  from  $T$ -terms onto physical state types of  $S$  such that the following holds: for all  $q, p$  in  $Q$ , if " $q \rightarrow p$ " is a theorem in  $T$  and  $S$  is in state  $f(q)$ , then this will "cause"  $S$  to change into state  $f(p)$ . If  $S$  realizes  $T$ , then  $S$  is said to be a *model of  $T$* .

Given definition 3.1, Putnam claims that "if *any* physical state is allowed as a possible 'realizer' for any ' $T$ -term' in a psychological theory, then [...] psychological theories will just have *too many* realizations" (Putnam 1988, p. 100).

This criticism extends naturally to computational descriptions. The independence from the physical systems realizing them seems to be a strength and a weakness of computations at the same time. Although computational formalisms allow one to specify what function  $f$  is defined by a particular computation (in the sense that the computation takes values from the domain of  $f$  as inputs and delivers values from the range of  $f$  as outputs), they implicitly presuppose a notion of implementation, i.e., that it is understood on what physical systems they can actually “run”.

To put Putnam’s criticism differently from a cognitive science perspective, the assumption that the functionality of mind can be described in computational terms would not be sufficient to single out those physical systems that possess minds, since a functional description true of physical systems with a mind could also be true of physical systems without a mind.

Because abstract finite automata seem to capture essential aspects of the notion of “psychological theory” as used in definition 3.1, Putnam gives particular credence to his claim by proving that *every ordinary open system is a realization of every abstract finite automaton* without input and output (see the appendix of his 1988, pp. 121-125)—I christened this result *Putnam’s Realization Theorem*. He uses this result to make the further claim that “the assumption that something is a ‘realization’ of a given automaton description (possesses a specified “functional organization”) is equivalent to the statement that it behaves as if it had that description.” (p.124) In other words, functionalism, if it were correct, would imply behaviorism (ibid.)—this is truly not a welcome conclusion for functionalists (including adherents of CCM). Given the central

role that finite abstract automata play in computer and cognitive science, this result is more than discomfoting, and, therefore, seems worthwhile a close examination.

First, a particular level of description of a (given) open physical system is chosen, in this case a “field-theoretic” level.<sup>31</sup> Then, a description of the physical system in “the language of fields” consisting of an exact definition of its spatial boundaries during a given interval of real-time is assumed.<sup>32</sup> Finally, an arbitrary finite state automaton (FSA) without input and output is fixed. The goal, then, is to find “physical states” and a mapping, which relates (computational) states in the automaton to those in the system such that the system “obeys” the machine table of the automaton. The definition of physical states becomes necessary, because all (field-theoretic) states of an open physical system are assumed only once by the system, hence they are *different for different times*. But *common* physical states at different times (i.e., state types) are needed to correspond to those automaton states (state types) that are repeated during a “run” of the automaton.

Putnam’s proof of his counterintuitive theorem hinges crucially upon a very “liberal” formation of physical states, namely an arbitrary (possibly infinite) union of “maximal states”, assuming a field theoretic level of description:

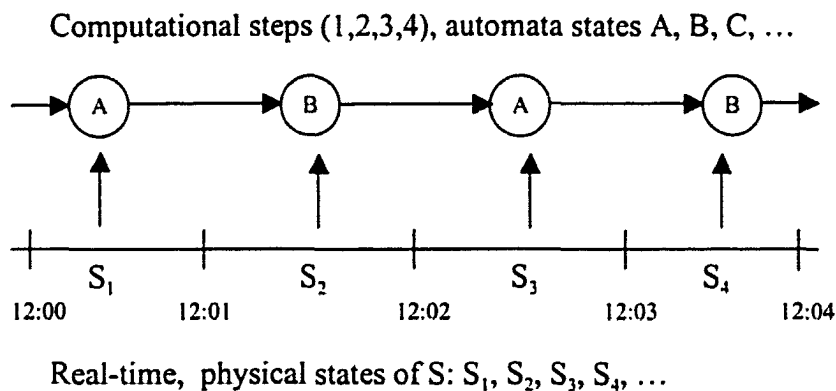
“In physics an arbitrary disjunction (finite or infinite) of so-called ‘maximal states’ counts as a ‘physical state’, where the maximal states (in classical physics) are complete specifications of the values of all the field variables at all the space-time points”.

(Putnam, 1988, p. 95)

---

<sup>31</sup> Open systems are not shielded from environmental influences. This assumption together with the “principle of non-cyclic behavior” is crucial to Putnam’s argument as it implies that the physical system assumes each state only once!

<sup>32</sup> Note that the physical system need not form or correspond to a “legitimate”, i.e., empirically possible object. For the argument, any description of a spatial region together with its spatial boundaries, “closed set of points in space” topologically speaking, suffices.

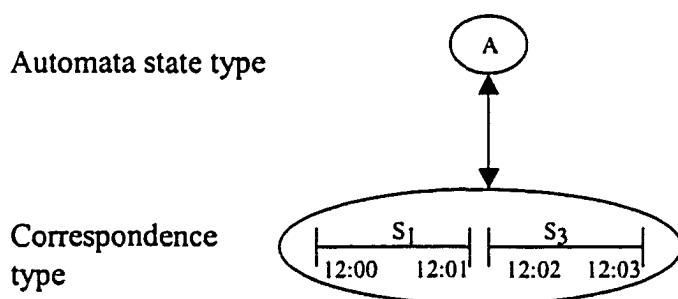


**Figure 3.1** Physical states are defined as sets of maximal states of a system  $S$  for a given interval of real-time in such a way that they are in correspondence with automata states in a “run” of the automaton.

Putnam considers regions in phase-space, that is, real-time intervals during the “life-time” of a physical system  $S$ , e.g., all maximal states of  $S$  from 12:00 to 12:04 on October 22, 1997. The set of values of all field parameters at all points within the boundary of  $S$  in such a real-time interval he calls an “interval state”. Interval states relate real-time of the physical system to “computation time” (=computational steps) of the automaton. They are taken to be *physical states* that correspond to sequences of abstract states determined by the machine table of the automaton—see figure 3.1. By construction, transitions between interval states are “causal” in the sense that they can be predicted from the laws of physics given the physical conditions on the boundary of  $S$  throughout its life-time (I will say more about this shortly).

In a second step, the fact that the automaton might be in the same computational states at different times in a particular run has to be accounted for. This is achieved by defining yet another kind of state, call them “correspondence states”, by taking (possibly

infinite) disjunctions over interval states according to the machine table.<sup>33</sup> Being types of interval states, correspondence states—as their name suggests—can then be mapped 1-1 onto automaton states such that the following holds: if the physical system is in correspondence state  $A$  and the machine table contains a transition from automaton state  $A$  to automaton state  $B$ , the physical system will transit (according to the laws of physics) into correspondence state  $B$ , i.e., it will “realize” the machine table of the automaton. Note that correspondence states are considered “legitimate physical states” by the physical theory.



**Figure 3.2** Correspondence states are defined as sets of maximal states of all physical states which correspond to the same automaton state resulting in an isomorphic mapping between correspondence and automata state types.

Although Putnam gives no explicit definition of what it means to “realize” (the machine table of) an automaton, a definition can be directly extracted from his proof:

*Definition 3.2:* A physical system  $S$  realizes  $n$  computational steps of a FSA without input and output within a given interval  $Int$  of real-time if for each state in the FSA there exists

---

<sup>33</sup> The details of how correspondence states are defined depends on the particular machine table. Roughly, the  $n$ -th interval state in the sequence of real-time intervals will become part of correspondence state  $A$ , if the automaton is in

one correspondence state of  $S$  and a division of  $Int$  into  $n$  subintervals such that: if the FSA at the  $k$ -th step ( $1 \leq k < n$ ) is in state  $A$  transiting to state  $B$  in the  $k+1$ -th step and the physical system is in correspondence state  $A$  during the  $k$ -th subinterval of  $Int$ , this causes it to transit into correspondence state  $B$  in the  $k+1$ -th subinterval of  $Int$ .

Another way of ensuring the correspondence of correspondence states with automata states is to claim the existence of a bijection (i.e., a reversible one-to-one function) between those two kinds of states (which might actually come closer to what Putnam might have had in mind, see also Chalmers, 1996):

*Definition 3.3:* A physical system  $S$  (described in a theory  $P$ ) realizes  $n$  computational steps of a FSA without input and output within a given interval  $Int$  of real-time if there exists a 1-1 mapping  $f$  from automata state types onto physical state types of  $S$  and a division of  $Int$  into  $n$  subintervals such that for all automata states  $q, p$  the following holds: if  $q \rightarrow p$  is a transition in the automaton from the  $k$ -th to the  $k+1$ -th computational step ( $1 \leq k < n$ ) and  $S$  is in state  $f(q)$  during the  $k$ -th subinterval of  $Int$ , then this will “cause”  $S$  to change into state  $f(p)$  in the  $k+1$ -th subinterval of  $Int$ .

Given this definition,<sup>34</sup> Putnam’s result reads as follows:

---

state  $A$  after  $n$  computational steps.

<sup>34</sup> More interesting than the actual phrasing of the theorem is the striking structural similarity of definitions 3.1 and 3.2. In fact, Putnam notices at some point, that the notion “psychological” theory in his (and Lewis’) account simply *replaces* the computational formalism in the standard version of functionalism (see Putnam 1988, p. 99). And, indeed, it makes no difference to Putnam’s argument if one speaks of the realization of a psychological theory or the realization of a computation as long as both involve states and transitions between them (in the former case “mental states”, in the latter “computational”)—see also chapter 4, section 3.



*Theorem 3.4:* (Putnam's Realization Theorem) There exists a theory  $P$  such that for every ordinary open system  $S$ , for every finite state automaton  $M$  without input and output, for every number  $n$  of computational steps of the automaton  $M$ , and for every real-time interval  $Int$  (divisible into  $n$  subintervals)  $S$  (described in  $P$ ) realizes  $n$  computational steps of  $M$  within  $Int$ .

Putnam's own diagnosis of what went awry to make this construction possible points to the liberal formation of physical states/state types: in order to avoid this kind of counter-intuitive result we "[...] must restrict the class of allowable realizers to disjunctions of basic physical states [...] which really do (in an intuitive sense) have 'something in common'" (Putnam, 1988, p. 100). In restricting the choices of physical states (that are supposed to correspond to computational states) to "natural" states which *really do* have something in common, one must not, however, involve "higher level" properties if the system is to *exhibit these properties* by virtue of its particular physical states; or in Putnam's words (who discusses the same problem for propositional attitudes, which are supposed to be reduced to physical-computational states):

"[...] this 'something in common' must itself be describable at a physical, or at worst a computational level: if the disjuncts in a disjunction of maximal physical states have nothing in common that can be seen at the physical level and nothing that can be seen at the computational level, then to say they 'have in common that they are all realizations of the propositional attitude A', where A is the very propositional attitude that we wish to *reduce*, would just be to cheat." (Putnam, 1988, p. 100)

### 3.3 Weak Spots in Putnam's Construction

The main implication of Putnam's proof for the current enterprise is that without restricting the formation of physical state types a direct correspondence between physical states and machine states can always be found. Implementation viewed as such a mapping, however, cannot single out interesting physical systems as "computers", since the class of finite state automata without input and output coincides with all physical systems describable at the level of physical fields under this notion.<sup>35</sup> As a consequence, this result—if true—would provide strong evidence against the tenability of a theory of implementation that is built upon the definition of physical states.

Intuitively, however, instead of believing in the applicability of Theorem 3.4, it seems safer to assume that something fundamental must have gone wrong. The most conspicuous place to look for a flaw is in the arbitrariness of the definition of physical state types (the correspondence states of the physical system). Since this turns out to be the most refractory and challenging deficiency of a "state-based" theory of implementation, I will address this issue in chapter 4 in great detail. In the meantime, I will try to strengthen the above result to exclude some *prima facie* criticisms.

One obvious problem is that the kind of automaton used in the theorem lacks any input/output interaction, and it is not quite clear what such an automaton is supposed to do at all! Usually, in automata theory, a FSA  $A$  is defined as a quintuple  $\langle Q, \Sigma, \delta, q_0, F \rangle$  where  $Q$  is the set of states,  $\Sigma$  the input alphabet,  $\delta$  the transition function from states and

---

<sup>35</sup> There are various extensions of Putnam's result that limit the range of possible responses: it can be proved for finite state machines with input, for Turing machines with only input, for state machines with (countably) infinitely many

inputs to states,  $q_0$  the start state, and  $F$  the set of final states.<sup>36</sup> The state table, as exhibited by  $\delta$ , defines for each state in  $Q$  all possible transitions to other states depending on the current state and the current input (transitions can be made without reading input, too, or more transitions can be defined for the same input and state—in that case the machine is called “non-deterministic”). Starting in the single start state  $q_0$ , the automaton changes states according to its input and state table until it either “blocks” (i.e., reaches a state where it cannot make any transition according to the state table) or reaches a final state (in which case it is said to “accept the input”). It is obvious that the input plays a crucial role in the behavior of the FSA. What the FSA does (=“computes”), is determined by the set of input strings that the FSA accepts. Now, if input is left out completely, the whole notion of “accepting an input” is taken away from the FSA, every such FSA always accepts only one “input”, i.e., no input—the empty string  $\epsilon$ . Such a FSA can only follow a predetermined trace of transitions through the state table. Depending on whether non-determinism is allowed or not, the FSA will either have multiple paths through the directed graph defined by the state table or only a single path where every step is determined and predictable without any additional knowledge about “external” factors. It seems that Putnam must have had the deterministic case in mind, since his construction would not work in the non-deterministic case: to see this simply assume that there is a computational state that has two different successor states, eventually leading to two different computational paths. Putnam’s mapping, however,

---

states, and, finally, for state machines that can read countably infinite strings of characters, see Scheutz (1997), some of those results I will discuss here and in section 3 of chapter 4.

only maps one computational path on the “temporal evolution” of the physical system. Therefore, a modification would be required to account for non-determinism. Note that the lack of branching is exactly the reason why Putnam’s automata cannot have input or output, as different input states might require branching and thus might lead to different computational paths.

If all we require of a “computational physical system” is to follow a single sequence of inevitable state transitions, then Putnam’s theorem does not come as a surprise: it seems always possible to map a sequence of states (from a finite set) onto integers, where each integer corresponds to a state in a physical system (given certain assumptions on physical systems as pointed out by Putnam); simply take the index of the automata state in the sequence as the value of the mapping (the  $i$ -th state is mapped onto  $i$ )—a similar construction has been already pointed out by Stabler (1987) before (see the last chapter): by going through a natural sequence of physical states (one that is defined and determined by the laws of physics), every physical system realizes every input-outputless automaton.

What can be concluded from this for cognitive science, even if one agrees with Putnam’s understanding of computation? Probably not much more than that input-outputless automata are not an appropriate formalism for describing cognitive systems—but that seemed clear in the first place. In order to get more interesting behavior, Putnam’s construction needs to be extended to allow at least for conditional branching within the state transition graph of the automaton. If transitions of the automaton do not

---

<sup>36</sup> Sometimes, if the automaton is also required to produce output, another component, the output alphabet  $\Sigma'$ , is added and  $\delta$  is defined correspondingly as a function from states and inputs to states and outputs, see Hopcraft and Ullmann (1979).

solely depend on the current state, but also on the current *input* character, one would expect the “causal structure” of the underlying system to be more complex.

Chalmers (1996) argues that Putnam’s construction does not obey the right kind of state-transition conditional. What is required, in his view, is not that “on all those occasions in which the system happens to be in state  $p$  in the given time period, state  $q$  follows.” (p. 312) Rather it is required that “*if* the system were to be in state  $p$ , *then* it would transit into state  $q$ ”(p. 312). In particular, Chalmers mentions multiple start states and transitions between states that are unreachable from a given start state, a possibility which is not reflected in Putnam’s construction. Although in the case of one start state both of the above requirements amount to the same, this is not true for multiple start states, because then Putnam’s construction would only exhibit a mapping between a single “trace” through the state table graph (from one start state) as opposed to a mapping of the whole graph. More generally, it is not true for machines where the computational trace through the state table depends on “external” factors, that is, on *input*. For those machines, it seems, not only a particular computational trace, but the whole internal structure would have to be reflected in the physical system in order to account for all “possible inputs” (whatever “possible inputs” may be—this itself is far from being unproblematic).

Putnam acknowledges the fact that his construction does not work in general for automata with input and/or output (1988, p. 124). But he argues that one could still *interpret* a physical system  $S$  that allows for inputs and outputs as being an abstract machine with input and output using his construction as follows: suppose  $M$  is a FSA with I/O. Then *if*  $I$  happens to be the input to  $S$  at a certain time  $t_n$  and  $O$  its output at a

later time  $t_{n+1}$ , one can set up a mapping between concrete and abstract inputs and outputs *in addition* to specifying all “inner states” of  $S$  (as before) such that the physical system realizes  $M$  (Putnam, 1988, p. 124). In particular, it follows that “behaviorism can be reduced to functionalism”, since “inner states” do not matter: every system with inputs and outputs can be interpreted as having the right kinds of inner states as long as it gets the I/O mapping right.<sup>37</sup>

While this is a valid statement, the statement it follows from (namely that  $S$  realizes  $M$ ) is not true.  $S$  could still fail to realize  $M$  in general, as the above only guarantees that it realizes a *particular run* of the abstract machine. The structure of the abstract machine might not be “mirrored” in the physical, in fact, it might only exhibit the same “input/output”-behavior for this particular input/output pairing. This is Chalmers’ main critique of Putnam’s construction, which he shares with Chrisley (1994). Both argue that Putnam confuses particular runs of computations on certain inputs and realizations thereof with realizations of the “structure of the machine” for all possible inputs.

Necessarily, these arguments involve a notion of “counterfactual” or “modality” to be able to account for non-actual, that is, counter-factual behaviors of the physical system. But this is exactly what Putnam views as highly problematic. The big question, therefore, is whether a counterfactual based notion of “realization of a computation” does create more problems than it can actually solve (I will briefly attend to counterfactuals in section 4 of chapter 4).

One easy way of dismissing merely counterfactual-based charges is to show how Putnam’s Realization Theorem can be strengthened in such a way that it applies not only

---

<sup>37</sup> Compare this to what has been said about “inner states” in dynamical systems in chapter 2, section 4.

to particular runs of an automaton for a given input, but to all possible inputs of the automaton.

Since automata with input and output can be simulated by an automaton with only input using input/output pairs of the former as input for the latter, it suffices to prove the theorem only for automata with input:

*Theorem 3.5:* For every ordinary open system  $S$ , for every finite state automaton  $M$  with input, for every real-time interval  $Int$ , and for every input string  $w$ ,  $S$  realizes  $|w|$  computational steps of  $M$  within  $Int$  (for the given input).

*Sketch of proof:* Consider any ordinary open system  $S$  during an arbitrary interval of real-time  $Int$  and any finite state automaton  $M$  with any input string  $w$ . Let the surface of  $S$  (=its boundary) or part of the surface be the “input region”.<sup>38</sup> Note that the environmental conditions on the boundary throughout  $Int$  specify the input that  $S$  will receive. By the *Principle of Noncyclical Behavior*, “the state of the boundary of such a system is not the same at two different times” (Putnam, 1988, p. 121). Then define interval states not only for the interior, but also for the boundary region in the usual manner (so we get two kinds of interval states, internal and input). For the  $i$ -th input (that is, the alphabet symbol read on a particular run of the automaton after  $i-1$  computational steps), define an input correspondence state for all  $i \leq |w|$ . The same needs to be done, of course, for the interior correspondence states. Finally, for a particular computation, one checks easily that if the automaton transits from  $A$  reading  $a$  to  $B$  in the  $i$ -th step, the

---

<sup>38</sup> There are problems connected with this move. Chrisley (1994), for example, argues why the boundary of a system cannot be taken as input region, see section 4 of chapter 4 for a more detailed discussion.

system  $S$  will be in interior correspondence state  $A$  with input correspondence state  $a$  in the  $i$ -th interval state transiting into interior correspondence state  $B$  (according to the definition of the correspondence states). So, every physical system will realize every finite state automaton with input (where the input device of the automaton is not specified physically—compare Putnam’s comment on p. 124).

There is a sense in which the above theorem still only covers a particular trace of the automaton, since the logical structure of the argument is:  $\forall(\text{inputs}) \exists(\text{a realization})$ , i.e., given a input string a correspondence can be set up such that the system realizes the automaton. Hence, *in nuce* this is the same as Putnam’s argument: *realization depends on inputs*. The order of the quantifiers does not capture the structure of the automaton independent from its inputs. It treats *automaton cum input* more like multiple (inputless) automata each of which follows one trace (in a way, this is exactly what has been done in the proof). To be a genuine extension of Putnam’s construction, the order of the quantifiers needs to be reversed,  $\exists(\text{a realization})\forall(\text{inputs})$ :

*Theorem 3.6:* For every ordinary open system  $S$ , for every finite state automaton  $M$  with input, for every number  $n$ , and for every real-time interval  $Int$ :  $S$  realizes  $n$  computational steps of  $M$  within  $Int$  for every input string  $w$  such that  $|w|=n$ .

The main difference between theorems 3.5 and 3.6 is that we cannot use information about the  $i$ -th input character in establishing the mapping from physical states to automata



states: we do not know what the  $i$ -th input is going to be. Notice, however, that in the previous sketch of the proof, when we set up a correspondence between the  $i$ -th input character and the  $i$ -th input state, we did not care *what the particular nature of the  $i$ -th input was either*. To set up the required mapping, it was sufficient to *refer* to the  $i$ -th input by using the term “ $i$ -th input”. In other words: whatever state the physical system is going to be in the  $i$ -th interval of  $Int$  (that is, the  $i$ -th interval state), this state will get mapped onto whatever  $i$ -th input the automaton will received. Therefore, the proof of the last theorem was something stronger than required—it was actually a proof of this stronger theorem. All that is needed to prove the stronger version is to fix the reference to times and inputs at certain times—if the  $i$ -th input character is “a”, “a” and “ $i$ -th input” are co-referential. And this was already done in the previous case.

It is very likely that people like Chalmers or Chrisley would still not be convinced by the above argument. They would probably find flaws in the way the mapping is established, they might disagree with the way reference to the  $i$ -th input is fixed, or criticize the involved notion of causality. And surely, nobody would claim that the above result *really* does give a satisfactory account of the realization of a computation. However, what the whole exercise of extending Putnam’s construction was thought to show is a very simple point: contrary to the beliefs of some who dismiss the problem of “state formation” as not significant (see, for example, Chalmers 1996) the cause of “universal realization” might not be the nature of the state transition conditional, or the nature of the involved notion of causality, because the above extension of Putnam’s

construction, I would claim, does not share this deficit.<sup>39</sup> A flaw retained in the above, however, is the definition of rather “arbitrary” physical states by *fiat*. Therefore, it might well be the nature of these bewildering physical states that is responsible for all the intellectual turmoil.

### 3.4 Searle’s New Argument

Searle has been criticizing computationalism since the beginning of the Eighties. His first argument was aimed directly at *strong AI*, the view that “the mind is to the brain, as the program is to the computer hardware” (Searle, 1984, p. 28). Searle, being a firm opponent of strong AI, presented various rebuttals of this doctrine, the most famous of which is his heavily debated “Chinese room” thought experiment (see, e.g., Searle, 1980 or Searle, 1984). The argument rests on the assumptions that 1) programs are formal (syntactical), 2) minds have content (semantic content), and 3) that syntax itself is neither identical with nor sufficient by itself for semantics. According to Searle, it clearly follows from these premises that programs are neither sufficient for nor identical with minds, thereby refuting strong AI.

As conclusive as this may sound at first glance, that is, even under the assumption that the reasoning is valid, it can still be doubted that all the premises are true. And, as it turns out, much of the truth of the first assumption depends on how one interprets the notion “program” (see, e.g., Melnyk, 1996). But one does not even have to go that far: to conflate the notions of “program” and “process”, i.e., not to distinguish between

---

<sup>39</sup> In chapter 4, a formally precise version of the above proof sketch will be presented together with an analysis of potential points of critique.

computer programs and the computational processes they give rise to, is to commit a category mistake! It means to view the mind, something that is clearly a real-world process (presupposing one is not committed to any form of substance dualism), *as a program* and not as *specified by a program*. Nobody, I think, would make the claim that mind *is* a program (“is” read as “is identical to”): if minds were to be on a par with programs, i.e., if the term “mind” could be legitimately compared to the term “program”, then minds would be static, formal objects too—which seems totally absurd. Rather, people in strong AI would claim that minds can be specified by programs (i.e., “is” in the sense of “give rise to”). Thus, the first premise of Searle’s argument should read “computational processes are formal”, but that does not seem right anymore (e.g., Smith, 1996, p. 33-34).

While programs specify processes and are thus abstractions, processes in turn are concrete, real-world phenomena by virtue of which they can be “about” something. In other words, programs and their components (such as variable symbols, constant symbols, procedure names, comments, etc.) derive their intentionality from that of the programmer, for whom they are meaningful, whereas computational processes *qua* process, I would argue, can possess *original* intentionality (in Haugeland’s terms). Just to give a quick example, a printing process does print  $n$  copies of my paper and not ‘ $n$ ’ copies; or email sent to my address gets delivered to `mscheutz@indiana.edu` and not to ‘`mscheutz@indiana.edu`’.

While computational processes *need not* have semantics (although it is hard for me to imagine a process being entirely syntactical—it seems to me that syntax is confined to descriptions of such processes), they certainly have *the possibility and potential* for

original intentionality (in the sense that it can be directly *about* something without the need of a human mind as mediator). Note that the claim that a process has original intentionality is stronger than saying it has semantics as the latter might just be attributed or derived.

Much has been written on Searle's Chinese room argument and its problems (see Hofstadter, Dennett, Fodor, Pylyshyn, Lycan, Haugeland et. al. in *Brain and Behavioral Science, 1980*), but in my view most of these criticisms are secondary. The first and foremost fallacy, I would claim, is the conflation between the specification of a process and the process proper. Instead of "Is the mind a computer program?" Searle should have asked "Is the mind a computational process?" (there are indications in his most recent writings that Searle might be more aware of this distinction—see his 1998).

Naturally the above reply is very brief and would need further, much more detailed discussion. However, I will not attempt such an elaboration of my critique of Searle's first argument here, since Searle has advanced another argument more recently, which is more pertinent to the present enterprise. In his book *The Rediscovery of Mind*, Searle (1992, p. 210) augments the "Chinese room" thought experiment by the claim that a physical system's physical properties do not suffice to determine its syntactic properties. Syntax has to be assigned to a physical system, and this assignment, according to Searle, is arbitrary: "If computation is defined in terms of the assignment of syntax then everything would be a digital computer, because any object whatever could have syntactical ascriptions made to it" (Searle, 1992, p. 207). In other words, whether or not a physical system is "implementing" (or better, *realizing*) a program depends solely on one's interpretation of that system:

“On the standard definition [...] of computation it is hard to see how to avoid the following results:

1. For any object there is some description of that object such that under that description the object is a digital computer.
2. For any program and for any sufficiently complex object, there is some description of the object under which it is implementing the program.

Thus for example the wall behind my back is right now implementing the Wordstar program, because there is some pattern of molecule movements that is isomorphic with the formal structure of Wordstar. But if the wall is implementing Wordstar then if it is a big enough wall it is implementing any program, including any program implemented in the brain” (Searle, 1992, p. 208-209)

The core of Searle’s view is thus: the standard notion of computation (i.e., Turing computability) is not suitable for describing minds, since one can always find a suitable interpretation of a computation under which a given system realizes this computation. This has not only fatal consequences for strong AI, but for *any view* maintaining that minds can be described computationally; in particular, computationalism, cognitivism (in Searle’s terms), and various forms of functionalism are at stake. It, therefore, does not come as a surprise that Searle’s attack aimed against main stream cognitive science created hefty reactions, most of which tried to find faults in his reasoning.

Unfortunately, Searle himself does not flesh out his intuitions about the “arbitrariness” of the “physical realization of computations”. Only a few remarks show how he intended to go about proving that every physical system “implements every computation”, an argument structure similar to that of Putnam: assume the “received

view on physical realization of a computation” and show that it leads to unacceptable results. Therefore, the received view needs to be revised if it is to be of any use for computer and/or cognitive science.

Fortunately, a quite succinct formal (*re*)construction of Searle’s ideas is provided by Copeland (1996), a defender of the “traditional theory of computation”. Sharing with Searle the commitment to the received view of computation, Copeland in his *What is Computation?* not only formalizes Searle’s intuitions, but also points out their shortcomings, an argument that will be reviewed in the next section.

### **3.5 Copeland’s (Re)construction of Searle’s Theorem**

Copeland’s strategy to rebut Searle’s view of “universal computation” is two-fold: first he reformulates Searle’s two above-mentioned theses (Searle, 1992, p. 208-209) as a theorem, which he baptizes “Searle’s Theorem”. From the (sketch of the) proof of Searle’s theorem it is then apparent where the intuitive notion of implementation has to be altered, since a “naïve view” on program implementation does indeed make Searle’s theorem true. The presentation of Searle’s theorem is followed in a second step by Copeland’s own analysis of what went wrong in the construction. He localizes the shortcomings in a non-standard interpretation of theoretical terms that are thought to describe the architecture of a computational system. Excluding non-standard interpretations of theoretical terms formally describing “algorithms and their corresponding architectures” (in a sense to be specified in the following) is what, in Copeland’s view, ultimately can block unwanted conclusions such as Searle’s “the wall behind my back is right now implementing the Wordstar program”.

The driving force behind Copeland's analysis of what it means for an entity  $e$  to *compute* a function  $f$  is his conviction that "[...]to compute is to execute an algorithm" (Copeland, 1996, p. 335). While this may seem too severe a restriction of the notion of computation to some (if not from a theoretical, then at least from a practical point of view), in the context of our evaluation of Searle's theorem it does not matter, as Searle would not be opposed to it. What does matter rather is Copeland's formal clarification and definition of the notion " $e$  computes  $f$ ":

"To say that a device or organ computes is to say that there exists a modelling relationship of a certain kind between it and a formal specification of an algorithm and supporting architecture. The key issue is to delimit the phrase 'of a certain kind'" (Copeland, 1996, p. 335).

Note that Copeland attempts a comprehensive account of implementation, where "entity  $e$ " is not co-extensive with "physical system". Rather, entity  $e$  could be "real or conceptual, artifact or natural" (Copeland 1996, p. 336). What matter is that it can be formally described by a specification SPEC as an architecture together with the formal specification of an algorithm  $\alpha$  for that architecture. For example, the blueprint of a PC together with an addition program written in 486 assembly language would constitute a formal specification SPEC in Copeland's sense.

Copeland then asks: "So on the one hand we have SPEC, a description of a machine, and on the other we have an entity  $e$ . How do we bridge the gap and say that  $e$  is such a machine (at the time in question)?" (p. 338) And he immediately answers: "The bridge is effected by means of a system of *labelling* for  $e$ ", which is a way of assigning labels (to

parts of  $e$ ) “that constitute a ‘code’ such that spatial or temporal sequences of labels have semantical interpretations” (ibid.).<sup>40</sup>

*Definition 3.7:* A labeling scheme  $L$  for an entity  $e$  consists of two parts:

1. the designation of certain parts of  $e$  as label-bearers
2. the method for specifying the label borne by each label-bearing part of  $e$  at any given time

Note that nothing is said about the nature of the relation between labels and parts of  $e$ . One would assume that it has to be restricted to some sort of functional correspondence, otherwise the term “label” would not be appropriate. Furthermore, the set of methods of specifying labels for each label bearer needs to be confined to *effective* methods, otherwise this would defeat the purpose of using a labeling scheme in the first place.<sup>41</sup> This last deficit could be eliminated by explicitly requiring that there be only finitely many label-bearing parts (as finitely many labels can always be assigned to finitely many parts effectively).

Obviously, there is a minimal requirement that each labeling scheme for a given formal specification SPEC has to satisfy, if it is to “model” the formal specification correctly: it must have enough different labels for all (non-identical) constants of SPEC

---

<sup>40</sup> Note that introducing a notion like “labeling scheme” for parts of the system is required if one wants to establish a correspondence between formal, theoretical terms (describing parts in the computational architecture) and “regions” (in real, physical space-time or another abstract topology).

<sup>41</sup> If non-effective labelings were permitted and the system had “infinitely many parts”, then it seems possible to label parts in such a way that the entity  $e$  can compute “non-computable” functions (to provide an exact answer, however, one would have to first pick a system with infinitely many parts together with their causal relationships....).



such that the latter can be mapped onto the former (this, in turn, requires the system to have “enough parts” to be labeled). Such a mapping between the constants of SPEC and the labels is the prerequisite for the definition of general truth-conditions relating SPEC and  $e$  such that it is meaningful to ask if  $e$  is a model of SPEC. Note that a truth-definition for SPEC crucially depends on the formal language of SPEC (in particular, on the logical and non-logical symbols, the way of combining them, etc.). Copeland downplays the importance of this issue somewhat by cryptically remarking “For definiteness, let SPEC take the form of a set of axioms, although nothing in what follows turns on the use of the axiomatic method as opposed to some other style of formalisation” (Copeland, 1996, p. 337-338). In fact, he does not provide any details regarding the nature of the minimal logical requirements of SPEC, although his critique of Searle’s Theorem assumes that a connective like “ACTION-IS”, which indicates a state change in the computational architecture and has to be interpreted in a very particular way, be part of SPEC.<sup>42</sup> While a serious truth-definition can only be provided if the language at hand is clearly defined and the interpretation of all involved predicates and connectives (such as “ACTION-IS”) is completely determined, I will simply assume with Copeland that this can be done for a given formal specification SPEC. Again, the minimal logical structure of SPEC would be crucial to a general theory of implementation. For our purposes, however, this incompleteness of Copeland’s sketch is a minor glitch, as we will focus in the following on a very specific formal system.

---

<sup>42</sup>In fact, Copeland argues that ACTION-IS, if it is supposed to capture the notion of state change or state transition, cannot be defined as material implication (in the sense, that “if the system is in state  $q$ , then it will go in state  $p$ ”). State transitions, in his view, have “modal force”, see also the end of this section.

If the truth-conditions for sentences of the language of SPEC with respect to the labeling scheme  $L$  (for an entity  $e$ ) can be given, i.e., if the notion “ $\phi$  is true of  $e$  under  $L$ ” is defined for all sentences  $\phi$  in SPEC, then the notion “model of SPEC” can be formally expressed:

*Definition 3.8:* Let SPEC be a formal specification,  $e$  be an entity, and  $L$  be a labeling scheme for  $e$ . Then the pair  $\langle e, L \rangle$  is a *model of SPEC* iff every sentence of SPEC is true of  $e$  under  $L$ .

Copeland uses the notion “model of SPEC” to define formally what it means for an entity to compute a function:

*Definition 3.9:* An entity  $e$  is *computing* function  $f$  iff there exists a labeling scheme  $L$  and a formal specification SPEC (of an architecture and an algorithm specific to that architecture, which takes arguments of  $f$  as inputs and delivers values of  $f$  as outputs) such that  $\langle e, L \rangle$  is a model of SPEC.

Unfortunately, the above definition conflates “computation” and “implementation” subsuming both under the notion “is computing” (see also chapter 4, section 2). Instead, it should and can be broken down into two parts: the relation between  $e$  and SPEC (which is established by virtue of the labeling scheme  $L$ ) and the relation between the algorithm part  $\alpha$  of SPEC and  $f$ . To see this, we need to make details of the algorithm such as the

relation between its inputs and outputs explicit. The notion “the function computed by an algorithm” can be used to distill the abstract input-output mapping  $f$  from the specific algorithm  $\alpha$  for the given architecture SPEC (obviously it is assumed that  $\alpha$  be deterministic). While the model-relation (definition 3.7) holds between  $\langle e, L \rangle$  and SPEC, the latter accounts for the relation between the  $\alpha$  part of SPEC and  $f$  (would this be the “computation”-relation?). Therefore, to say that  $e$  is computing function  $f$  is to say that  $e$  implements SPEC and the algorithm part  $\alpha$  of SPEC computes function  $f$ . This distinction together with definition 3.8 clearly exposes Copeland’s view of implementation as that of semantic interpretation.

Note that there are other reasons for separating descriptions of computational architecture from those of algorithms that can run on them. One, which I find especially important, is the possibility of identifying two systems as being *the same kind of* computer. In order for such an identification to work, an architecture description is needed that is silent on algorithms that could be implemented. Since there is a many-to-one relationship between architecture and algorithms (the reverse is obviously true too), different algorithms can run on the same architecture. Therefore, using a pairing of architecture and algorithm, one would have to have some means of separating architecture description from algorithm in a given SPEC—this could have been avoided by keeping them separate in the first place. Another is that in order to evaluate Searle’s claim one should not need to know what function the algorithm part specifies, it should suffice to show that the physical system implements the architecture!

Despite mingled terminology, Searle's Theorem can now be stated precisely as follows:

*Theorem 3.10: (Searle's Theorem)* For any entity  $e$  (with a sufficiently large number of discriminable parts) and for any architecture-algorithm specification SPEC there exists a labeling scheme  $L$  such that  $\langle e, L \rangle$  is a model of SPEC.<sup>43</sup>

In more intuitive terms, Searle's theorem states that every object can be "interpreted" as implementing any given computational architecture.<sup>44</sup> To allow for such an interpretation eventually boils down to exhibiting the "right" labeling scheme, which, in turn, requires that the "right" parts of  $S$  be singled out as label bearers in a systematic manner. One general method for finding the "right" parts in any structure (which does not depend on the peculiarities of the system under scrutiny) can be derived from Putnam's construction—I will describe it in the next chapter; another can be found in Copeland's proof sketch of Searle's theorem.

To see how that the wall behind me implements the Wordstar program, assume a formal specification VNC (of a von Neumann computer, for example) is given together with the algorithm of Wordstar (written for that architecture), of which the following algorithm fragment is a short piece:

---

<sup>43</sup> Note that for Searle the entity  $e$  is a physical system.

<sup>44</sup> Whenever I presented a version of Searle's Theorem to an audience, people immediately sensed that something must be wrong with this result, or otherwise—as somebody once put it—"computer dealers would be selling walls". Obviously, the real challenge is to find and explicate the defects in its proof.

LD B, #10

LD C, #20

ADD A, B, C

This simple program loads the values 10 and 20 into registers B and C, respectively, adds them and stores the result in register A. One would like to define a labeling scheme  $L$  such that a particular wall (the one right behind me, for example) under  $L$  is a model of VNC. To do this, one needs to single out parts of the wall that are supposed to correspond to the three registers in VNC, call them “wall states”:

“The first thing to be done is to settle on a way of correlating binary numbers with physical structure. Let’s simply grant Searle [...] a method that enables one to correlate binary numbers with regions of whatever physical object is in question. For instance, if the wall has a high polymer content then the following simple method can be used: when the number of polymer chains that end in a given space S is odd the S tokens 0, and when the number is even S tokens 1”. (Copeland, 1996, p. 343)

Then one records the states of all registers in an actual von Neumann computer while it is running Wordstar for  $n$  computational steps and relates these to labels of wall states (as defined by  $L$ —see Figure 3.3).

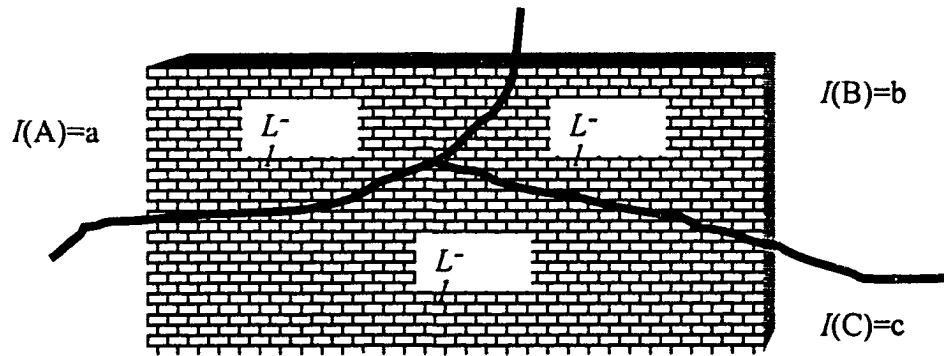


Figure 3.3 Regions in the wall are singled out to become bearers of labels (a,b,c), which in turn will become the interpretation (via the interpretation function  $I$ ) of constants in SPEC (A,B,C).

That is, for any two consecutive computational steps there will be two consecutive intervals of real-time such that the content of a particular register corresponds to a particular wall state (during the respective interval). Figure 3.4 show the value of all registers at each of the three computational steps.

Steps	Command	Register A	Register B	Register C
1	LD B, #10	xxx	10	xxx
2	LD C, #20	xxx	10	20
3	ADD A, B, C	30	10	20

Figure 3.4 Computation in a real “von Neumann” CPU: the table shows the values of all registers at the respective time-steps (values reflect the states of the registers after the command has been executed). ‘xxx’ indicates that the actual value does not matter.

Without going into details about the formal description of the architecture, it will still be helpful to see how one could write formal axioms that describe what the respective

“machine command” is supposed to do. In our case, we will need axioms for “ADD” and “LD” using the connective “ACTION-IS” to describe state transitions. We also assume an “instruction register” whose value indicates the current instruction :

$$\underline{\text{IF}} \text{ instruction}=\text{“LD } r,\#n\text{”} \underline{\text{ACTION-IS}} r \leftarrow n$$

$$\underline{\text{IF}} \text{ instruction}=\text{“ADD } r,s,t\text{”} \underline{\text{ACTION-IS}} r \leftarrow s+t$$

Note that  $r, s, t$  are variables for registers, whereas  $n$  is a variable for natural numbers.

It is easy now to exhibit an interpretation  $I$  such that all formal states will correspond to wall states at any time during the interval  $[t, t']$  ( $t < t'$ ) under consideration (see Figure 3.5). Furthermore, the axioms of VNC that describe “state transitions” will be mirrored by “wall transitions” under a certain interpretation of “transition”: assume the interval under consideration is  $[8:01, 8:03]$  and let  $\text{PLUS} =_{\text{def}} \langle \langle I(s), I(t) \rangle, I(r) \rangle$  and  $\text{VALUE} =_{\text{def}} \langle n, I(r) \rangle$  for all  $t \in \{8:01, 8:02, 8:03\}$ . Then the interpretation of the axioms is:

$$\forall t (\text{instruction}(t) = \text{“ADD } r,s,t\text{”} \supset I(r)_t = \text{PLUS}(\langle I(s), I(t) \rangle))$$

$$\forall t (\text{instruction}(t) = \text{“LD } r,\#n\text{”} \supset I(r)_t = \text{VALUE}(n))$$

Hence, the wall will implement Wordstar during  $[t, t']$ .

Time	Command	$I(A)$	$I(B)$	$I(C)$
8:01	LD B, #10	$I(xxx)$	$I(10)$	$I(xxx)$
8:02	LD C, #20	$I(xxx)$	$I(10)$	$I(20)$
8:03	ADD A, B, C	$I(30)$	$I(10)$	$I(20)$

Figure 3.5 Computation in the “wall CPU”: the table shows the values of all registers under the interpretation function  $I$  for each interval that is mapped onto computational time-steps.

There are obviously quite a few problems associated with this construction, and Copeland himself diagnoses three major shortcomings:

1. All computational activity occurred *outside* of the wall (by recording the activity within a CPU of a machine that *actually* performed the computation), meaning that the labeling scheme is constructed (from this record) *ex post facto*.
2. The labeling scheme involves unwanted temporal specificity (by limiting the wall’s computational capacities to the time interval  $[t, t']$ , a necessary consequence of the *ex post facto* nature of the labeling scheme).
3. The interpretation of “ACTION-IS” fails to support assertions about the counterfactual behavior of a real von Neumann computer.

All three problems point to the discrepancy between the “intended” interpretation of VNC and the “non-intended” one that turned the wall into a von Neumann computer. This discrepancy, according to Copeland, already suggests a criterion to distinguish between



“real computers” and “pseudo-computers” (such as the wall): real computers compute under a standard interpretation of the given architecture specification, pseudo-computers do so only by virtue of non-standard interpretations (they could not be viewed as computing under the standard interpretation). Thus, by requiring that the interpretation of all theoretical terms in the formal specification of the computing architecture plus algorithm be *standard*, one can according to Copeland exclude wall computers and the like. This requirement is easily incorporated into Copeland’s definition of computing: call a model of SPEC “honest” if it is a model under a standard interpretation. Then for an entity  $e$  to compute a function  $f$  means to satisfy definition 3.9 *and* be an honest model.

While one can agree that it is preferable to have models of a formal specification that are models under the standard interpretation if possible—since this is what it is to be the thing described by the specification in the first place—there are cases when it is not even clear what the standard interpretation is. Set theory has an ongoing debate about what the standard interpretation of the Zermelo-Fraenkel axiom system (ZF) is supposed to be (should it include non well-founded sets or not, does it allow for choices or not, etc.), one reason being that the universe of sets is underspecified by the axioms. Thus, there are mutually exclusive alternatives of extending the system giving rise to incompatible kinds of models, which nevertheless are still models of the ZF. In other words, there is an intrinsic problems buried in Copeland’s suggestion of how one could go about excluding “unwanted” computing systems: formal systems notoriously underspecify their model, or to put it differently, there are too many interpretations that might legitimately count as standard. Consider the description of a von Neumann computer. It might be realized in many different ways in different hardware, which already makes it hard to define

common properties of all the different labeling schemes required (in fact, it seems that the only commonality will be that these objects are all von Neumann machines, but this is obviously not helpful). Still, one could argue that whether a certain object “implements” a von Neumann computer can be decided if one is familiar with the von Neumann computers, what the terms in their specification mean, etc. But what about objects we do not understand well? What about objects we have not built? These objects would be *a priori* excluded from being “computers” as we could not decide whether they were honest models of an architecture description—we simply would not know what it means to be a standard interpretation of the terms for them. If the position is acceptable that only artifacts compute, then Copeland’s solution may be satisfactory. However, if one wants to allow the notion of computation to aid other disciplines such as cognitive science in understanding causal relations and laws, the above approach has to be discarded as too narrow and underspecified to be of any use.

There are quite a few open issues concerning the three shortcomings of the interpretation of SPEC Copeland points out in the proof of Searle’s theorem. I will address some of them in the next chapter, when I show that Copeland’s solution does not even work for artifacts: assuming that the notion of “finite state automaton” is an accepted specification of a computational architecture plus algorithm—it is normally not viewed that way, but one can certainly put the notion in this frame—it will turn out that everything implements an FSA, simply because the notion of FSA is too general to confine the set of potential realizers!

### 3.6 Implications of Putnam's and Searle's Theorems

While there have been many attempts to refute Putnam's and Searle's theorems, it seems to me that the methodological point of their criticisms mostly went by unappreciated. Surely, neither Putnam nor Searle really believe that rocks or walls compute. However, they believe, I contend, that the "implementation problem for computations" turns out to be more serious than expected and cannot be dismissed easily. Putnam takes this point even further and raises the question about the relation between psychological theories and their possible realizations. And, indeed, it does not take much to extend this argument to any formal theory that contains certain primitive terms (see the next chapter).

Without really acknowledging the transition, we have stepped foot on different territory—the realm of metaphysics proper. Questions about the nature of formal theories and their relationships to the real world have always been—in different verbal guises depending on the terminology *en vogue* at the time—the center of ontological and epistemological debates. It should not come as a surprise then that computations, too, eventually probe sacrosanct philosophical topics. What is surprising, however, is the fact that computational practice does not seem to be bothered by the lack of foundational clarity that the above theorems reveal. Alas, the "intuitive notions of implementation" used within computer science seems to be understood well enough to master practical challenges. Only when other disciplines borrow the notion of computation and apply it within their own domain, the almost systematic ambiguity and fuzziness of crucial terminology backfires. Thus, neither psychology nor philosophy, and in particular not

cognitive science, should build theories on notions that start to crumble under scrutiny—so the methodological advice!

Putnam's and Searle's theorems show that an "intuitive approach to implementation", one that allows for "arbitrary" assignments of syntax to physics (Searle) or for the establishment of a mapping between arbitrarily chosen physical and computational states (Putnam), gives way to the pressure of analytical inspection. Yet, these somewhat exaggerated results do not do full justice to the intuitive notions of implementation, mainly because these notions are limited to the domain of applied computer science, and it was not the computer practitioners who pulled them out of their context. In a sense, computationalists (i.e., people in cognitive science interested in a computational explanation of mind) are the truly accused, accused of importing blindly a notion into their respective disciplines without reflecting on the implicit assumptions involved in this notion: once "computations" are deprived of their "computers", it is no longer clear what other hardware might be a candidate to "implement them".<sup>45</sup>

The lesson to be learned from Putnam's construction, thus, is that as long as there are no criteria that distinguish "natural" from "unnatural" physical states, the door will be open to "unintended labelings of parts of a physical system" that are not really "parts" (in a common-sense understanding of "part"). It is not clear at all if criteria can be formulated for arbitrary systems. Rather, I venture to claim, it will be necessary for many systems to involve "high level" objects, concepts, and/or properties to describe common

---

<sup>45</sup>Interestingly enough, humans seem to be the appropriate hardware, for simple programs at least, as we can "execute" algorithms, i.e., apply them to inputs and determine their outputs. However, this level of "implementation" would be at best called "virtual machine" and correspond to a "macro interpreter" within WORD, which in turn is

features of states at a “low level” of description of physical systems, for the simple reason that the only thing all these “low-level” states have in common might just be that they realize a higher level state or property. Take, for example, a chair. Most people would agree that what it is to be a chair cannot be determined at a field theoretic level of description. Too many different fields will “implement” chairs, fields that have nothing in common *qua* fields other than that some of their “abstractions” admit of the same higher level description.

If this is so, i.e., that without already assuming common properties at a higher level of description, physical states that realize computational states cannot be shown to have anything in common at the low level of physical description, then every theory of implementation that is built upon establishing a mapping between physical states and computational states would face severe obstacles, to say the least. Deriving computational properties of physical states at a physical level of description seems to present almost insurmountable difficulties--it would mean to solve “reversed multiple realizability”. Analogous to “multiple realizability”, where the physical properties of realization could not be derived from the computational properties (“there are too many different realizers”), the same holds true of “reversed multiple realizability”: computational properties cannot be extracted from physical states, because there are too many possible computational abstractions.

Before concluding this chapter, I would like to point out the intrinsic connection between Putnam’s construction of “state types” and Copeland’s labeling scheme. Above

---

*implemented* by a different program (the WORD program), etc. Computationalists make the specific claim that there is a level “lower than conscious execution of algorithms” at which the brain performs computations.

I alluded to Putnam's strategy of constructing state types by "unintended labelings of parts of a physical system". Note how this way of putting Putnam's strategy blends with Copeland's idea of a labeling scheme, the reason being that "labeling parts" or "defining physical states" are merely two sides of the same coin. To single out space-time regions of a physical system for all times and tag them with a label is really not different from defining space-time regions as states. Both approaches need *physical criteria* to specify space-time regions. Both approaches refer to the so-specified space-time region. They only differ in the way they refer to them. The former uses a name (i.e., the label), whereas the latter uses the mathematical description of the physical theory such as *a set of field values* (i.e., the state). In either case, parts of a physical system are related to a computational description via a correspondence function: in Putnam's case it is the "type-formation", in Searle/Copeland's the "semantic interpretation". The next chapter will make this connection even more apparent.

## Chapter 4:

### The Alleged Rehabilitation of Computationalism

#### 4.1 The Stakes are High: Two Ways to Rescue Computationalism

Computationalism, despite all its weak spots and flaws, still dominates cognitive science. It seems that there is no *better* alternative within sight, even though some dynamic systems people are advancing vehemently and forcefully their new credo (see van Gelder & Port), which has already proved very promising in certain areas of research (see Port, Kelso, Smith & Thelen, Townsend & Busemeyer). Exactly because computationalism has served cognitive science well over the last forty years, in particular the area of artificial intelligence, many people are not willing to give up their fundamental convictions easily; especially not for philosophical arguments which, in their view, *only allegedly* render computationalism untenable (considering the absurd consequences of arguments such as those advanced by Searle and Putnam).

Once one has analyzed the charges held against it, there are basically two non-exclusive ways to defend computationalism: one is to keep standard notions of computation and try to explain how these can be related to the physical in a non-trivial and meaningful way. In fact, every computer practitioner has an intuitive understanding of how computations are related to the real world. This is what a practitioner's job is all about, to establish meaningful relations between abstract specifications and their physical realizations in order to accomplish practical goals. Hence, the first option is to attempt a

revision of the notion of implementation (preferably in accordance with practical experience).

The other, which is also motivated by reflecting on computers and computations in their practical, real world settings is to abandon the idea that standard conceptions of computation (such as Turing computability) are the ones capturing *all crucial aspects* of computation. As a consequence, a new notion of computation will have to be introduced, which takes computation as a *real-world phenomenon* seriously. Turing-computability will then be viewed as capturing only one aspect, namely the theoretical formal aspect of this new notion that embraces daily life routines and computer applications as well as logical, foundational results. Consequently, the reconstruction of computation will also give rise to an appropriate notion of implementation, or so it is hoped. Since I believe that this latter option is the only viable one, I will postpone a discussion of proposals to rescue the notion of computation along these lines for the time being.

Adherents of the former camp have attempted to salvage the classical notion of computation by explaining how one should go about defining an appropriate notion of implementation, which is invulnerable to Searle and Putnam style objections. This endeavor could be approached from two different directions: 1) one can view the relation between computation and computer as that of an interpretation that holds between formal elements of a theory and (formal) logical models of (physical) systems, or 2) one can try to avoid the “semantic route” by establishing some sort of direct correlation between computational and physical states. Note that whereas the former is essentially an observer-dependent view of implementation (as it relies on the assignment of an interpretation function), the latter is *somewhat* objective. It should allow one to give a



definite answer to the question what computation a physical system implements. The former, which I will call the “semantic interpretation view of implementation” (SV), is, for example, suggested by Copeland (1996), whereas Chalmers (1994,1996) is a proponent of the latter, which I will refer to as the “state-to-state correspondence view of implementation” (CV).

While SV and CV might seem similar at first glance, as one could argue that setting up a correspondence between computational and (given) physical states is analogous to interpreting computational states (over a given structure), this similarity disappears when one looks at it through metaphysical glasses (without making commitments as to the nature of the interpretation): SV is essentially an epistemological approach in a pragmatic guise as it is not concerned with the nature of the relation between computational and physical states; rather it emphasizes the way these states *can* be viewed (for whatever practical reasons). CV, however, makes ontological claims about the nature of computational states; for example, that they *are* certain physical states (“up to isomorphism”), or that they share certain structural properties (under the notion of “state transition”) with physical states (“under some notion of causality”).

However, despite their metaphysical difference, I will show in the following that the first glance was not so mistaken after all; except that it is not the notion “mapping” they share, but the notion “physical state” (which is not to say that they share “physical states” themselves—this may or may not be, what they share is the fact that both require the notion of “physical state” in order to develop the notion of implementation). That this is so and that problems connected with the notion of physical state underlie the SV as well as the naïve CV (i.e., the “naïve” state-to-state correspondence view as criticized by

Putnam, see definition 3.3), can be shown by exhibiting a counterexample to Copeland's solution to Searle's theorem, which is essentially based on Putnam's construction. Since it is (naturally) also a counterexample to the standard view on implementation, and *a fortiori* the "(naïve) state-to-state correspondence view", the kinds of physical states that haunt the naïve CV, also bring down the SV. Moreover, this shows—in my view—the common grounds on which Searle's and Putnam's criticisms are based, namely the problem of physical state type formation: if physical state types can be chosen arbitrarily, then literally everything can be viewed as computing (regardless of one's view of implementation).

Chalmers, who is one of the few authors to realize the importance of the notion of implementation for computationalism, attempted to overcome the difficulties resulting from Putnam's construction in his definition of implementation. While his view, which is essentially an instance of CV, does solve some of the problems of the naïve CV (that was attacked by Putnam), I believe he did not succeed in general, because by taking state type formation to be unproblematic, he did not correct the true deficiency.

Obviously, this will require careful attention and I will present a second argument that attacks his state-based notion of implementation by using a special kind of physical state type formation under which simple physical systems (such as a "battery-light bulb-switch"-system) are viewed as implementing arbitrarily complex computations. Although not *every physical system* can be shown to implement every computation under Chalmers' notion of implementation (at least I could not show it), the fact that *some simple physical systems* implement arbitrarily complex computations is still disturbing

and needs to be accounted for, especially since we would not be committed to attribute such computational powers to those systems.

In the end, these constructions are intended to show that despite the obvious attraction of the semantic or the state-to-state correspondence views of implementation, any such view is not tenable as a foundation for a *general theory of implementation*. This is not to say that there are no cases in which some of these views can be employed successfully, but rather that they fail to be general enough to serve as foundational concepts. What, precisely, prevents this generality is their intrinsic reliance on a notion of “physical state”; neither the mapping nor the interpretation function (besides having their own problems connected with them) themselves contribute to this deficit, yet they cannot make up for it either.

#### **4.2 An Analysis of Copeland’s Solution of the Implementation Problem**

Copeland attempted to define very general notions of computation and implementation (reducing implementation to a logical modeling relation and computation to the presence of this relation between a formal specification and a description of an entity) that view various kinds of systems as computers: von Neumann computers, neural networks, Turing machines, or finite state automata, just to name a few.

Not only concrete physical systems, but also abstract systems (such as the fictitious Turing machine) are subsumed under Copeland’s notion “entity  $e$  is computing function  $f$ ”. Although one could argue that in order to truly compute something “material” is required that does the computing (as abstract entities cannot *perform anything!*), which would exclude abstract entities  $e$  from Copeland’s definition, I find the wording “is

computing” more disturbing. For one, the difference between “*f* is *implemented* on *e*” and “*f* is *running* on *e*” (the algorithm specifying *f*, of course) is completely ignored by using the progressive form. In fact, the progressive form is misleading.

It seems that Copeland wanted to capture computational processes “running” on a particular architecture in his definition—this would be the “progressive form reading”—, while at the same time not being restricted to “(real-world) processes”, the latter being the “indicative form reading” (what could it possibly mean to have “a process running” on a abstract computational architecture?).

The tension of achieving these two goals at the same time in *one definition* is what blurs the useful ontological distinction between computation (as a formal object) and (actual as well as potential) computational process on the other. It does not come as a surprise that this conceptual tension finds its formal expression in the very definition of “is computing”: while the wording “is computing” suggests that Copeland wanted to capture the notion of “computational process” (that which is *actually running* on a computer), the existential quantifier in the *definiens* (“there *exists* a labeling scheme *L* and a formal specification SPEC”) hints at the notion “program” instead of “process”; it specifies a relation between a formal entity and parts of a physical entity, yet it does not involve a notion of actuality or process; the possibility of the entity (that is, the system) to run a particular program, suffices (i.e., that it is possible to set up a formal relationship with the computational description and the description of the entity).<sup>46</sup>

---

<sup>46</sup>Note that a correct reading of Copeland’s definition of implementation is crucial to his first and second objection to Searle’s theorem, as they are at best objections under the “program” reading.

There is another distinction to be drawn between “ $f$  is implemented on  $e$ ” and “ $f$  can be implemented on  $e$ ”: while the first stresses a factual, that is, actual state of affairs, the other involves a notion of modality to allow one to distinguish between *architectures cum algorithmic description* and *architectures sine algorithmic descriptions*. While for the latter specifies a class of functions (i.e., the class of functions  $f$  such that  $f$  can be implemented on  $e$ ), the former should be true of only one  $f$  (the one specified by the algorithm part of SPEC).

Another criticism regarding Copeland’s definition of implementation, already mentioned in the previous chapter, is the lack of a clear definition of what the minimal requirements of a potential language for SPEC are. It seems, for example, that the connective “ACTION-IS” or something equivalent would necessarily have to be part of it, and with it the specification of its minimal properties (such as counterfactual support, etc.). But if formal state transitions are to be modeled using counterfactual supporting connectives, the door will be opened to all kinds of criticism about the nature and legitimacy of counterfactuals, a debate that in my opinion should not be part of a theory of implementation.

There are other issues connected to the notion of labeling scheme that are not addressed by Copeland: 1) it is not clear how one would go about identifying parts of a system that qualify as bearers of labels, and 2) how one would go about doing this in a systematic way. For some people (e.g., presumably Rappaport), this might already be an instance of the *implementation problem*: the labeling scheme viewed as an abstraction *implemented* on entity  $e$ ! Also, it seems still justified to talk about systems implementing certain functions while it is not possible to assign labels to all their parts (e.g., take certain

analog chips that “compute” the solution to differential equations by exploiting the diffusion of potentials in silicon crystals).

For the time being, I shall ignore these difficulties that are a consequence of the generality of Copeland’s definition and concentrate rather on two necessary (but not necessarily sufficient) criteria, which Copeland suggests to sort out standard from non-standard interpretations (that is, to distinguish “honest” from other models):

“I suggest two necessary conditions for honesty.

1. The labelling scheme must not be ex post facto. [...]
2. The interpretation associated with the model must secure the truth of appropriate counterfactuals concerning the machine’s behavior.

Either of these two requirements suffices to debunk [...] alleged problem cases.”

(Copeland, 1996, p. 350)

Unfortunately, the generality of Copeland’s approach seems to make it impossible to eliminate non-standard interpretations, even if one applies these two criteria as in the following argument (which extends Putnam’s construction). If the argument goes through and it does indeed meet both criteria for “honesty”, then additional criteria are needed to single out “intended interpretations” (if this is possible at all).

*Theorem 4.1:* Every ordinary open system  $e$  is a model of every finite state automaton.

*Sketch of proof:* At this point I will only sketch the main steps of the proof as it can be generalized to cover not only finite state machines, but also psychological theories (in the sense of Putnam, 1988) in general (and I will fill in the details when I prove the latter

result). First of all, a formal specifications SPEC has to be defined (which itself consists of an “architecture part” and an algorithm part), in this case for a FSA. It is standard to define a FSA (without output) formally by a quintuple  $\langle Q, \Sigma, \delta, q_0, F \rangle$ , where  $Q$  is the set of states,  $\Sigma$  the input alphabet,  $\delta$  the “transition function” from states and inputs to states,  $q_0$  the start state, and  $F$  the set of final states. All triples of  $\delta$  can be viewed as instances of the axiomatic scheme  $\langle q, i \rangle \rightarrow q'$ , where  $q$  and  $q'$  are states,  $i$  is an input, and ‘ $\rightarrow$ ’ is a primitive meaning “transits”. This takes care of the “architecture part” of SPEC. The state table, as exhibited by  $\delta$ , defines for each state in  $Q$  all possible transitions to other states depending on the current state and the current input. Starting in the single start state  $q_0$ , the automaton changes states according to its inputs and state table entries until it either reaches a final state (in which case it is said to “accept the input”) or it ends up in the “trap” state (a state, from which it cannot make any other transition than remaining in this state for every possible input). Notice that  $\delta$  determines what the actual state transitions are in the FSA. These particular transitions can be viewed as the algorithm “implemented” on a more “generic automaton” (i.e., the given FSA without a particular  $\delta$ ). In other words,  $\delta$  is the “program” of the FSA  $\langle Q, \Sigma, q_0, F \rangle$ .<sup>47</sup>

Define a mapping  $f$ , then, from  $\Sigma^*$  into  $Q$  such that  $f(w)=q$  if the FSA is in state  $q$  after having read  $w$ , for all strings  $w$  in  $\Sigma^*$  (this mapping can be obtained inductively from  $\delta$ ). Obviously, the FSA takes arguments of  $f$  as inputs and delivers values of  $f$  as outputs (in

---

<sup>47</sup> One could also exclude  $F$  from the generic FSA, since in a way final states will depend the particular  $\delta$ . However, one can always take another “generic automaton” with a desired set  $F'$  different from  $F$ , if different final states are needed.

the sense that it ends up in the state, which is the output of the function). Hence, the second part of the requirements for the formal specification SPEC is satisfied too.

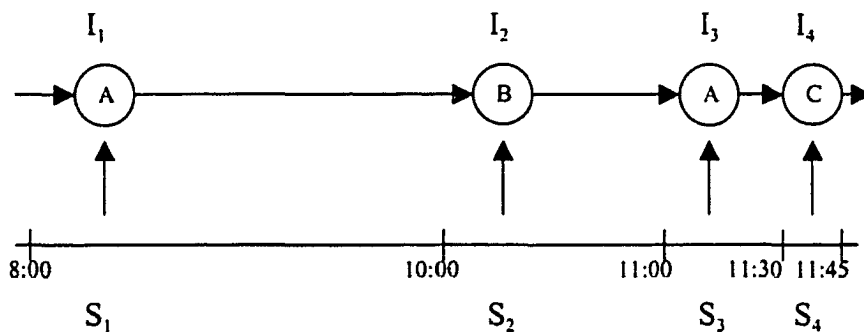
Next define a labeling scheme  $L$  (see definition 3.7) and an interpretation of ' $\rightarrow$ ' such that the formal specification SPEC is true under that scheme for every ordinary open system. Part 1 of the labeling scheme asks us to specify parts of the entity, i.e., of an open system, as label-bearers. In order to account for the fact that the FSA receives input from the "outside", I will treat inputs as far as the "model of the FSA" is concerned as "input states" and call all other states "inner states". Since the only parts about the automaton specified in SPEC are its states (input and inner), we designate the boundary of  $e$  and its "inner" part as bearers of labels (see the next section for a discussion of the legitimacy of this move). For part 2 of the labeling scheme a method has to be exhibited for specifying the label borne by each label-bearing part at any given time—this is where things get tricky.

;



Fig. 4.1 The relation between computational and physical states (where

Computational steps (1,2,3,4), automata states A, B, C, ...



Real-time, physical states of S: S<sub>1</sub>, S<sub>2</sub>, S<sub>3</sub>, S<sub>4</sub>, ...

the time interval considered is between 8 a.m. and 12 a.m. on March

22, 1998).

One has to define “physical states” for  $e$  and the boundary region of  $e$ , which can be related to the abstract states in the automaton. The physical states, call them *interval states*, will be defined (analogous to Putnam) as sets of values of all field parameters at all points within the boundary or at the boundary of  $e$ , respectively, for a given interval of real-time (the basic idea of the construction is shown in figures 4.1 and 4.2). The main difference to Putnam’s construction is that we will have to find a way of partitioning a fixed interval of real-time such that any arbitrarily long finite sequence of computational steps can be seen “as being implemented by the system within this time interval”. Since we do not know the actual length of the computational sequence ahead of time, the construction will have to ensure that we will get enough physical subintervals that can be mapped onto computational states. This can be achieved by letting each interval be only half the length of the previous one (and the first be just half of the total interval under

consideration). As in Putnam's construction, each automata state type corresponds then to the union of different physical states, that is, interval states:

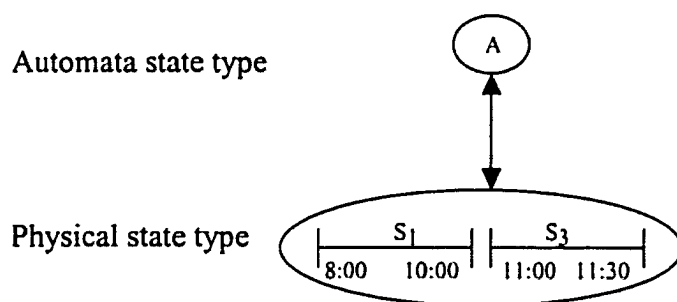


Fig. 4.2 The relation between grouped physical states (i.e., interval states) and automata state.

It follows that  $e$  will *always* finish its “computation” of  $f$  within a finite real-time interval independent of the length of the input. But this means that the FSA could also “compute” infinite strings in a finite amount of time (actually, in an arbitrarily small time interval) adding evidence to the suspicion that such a stipulated correspondence cannot be what implementation is all about.

Finally, one has to verify that  $e$  is an honest model under the given labeling scheme  $L$  for SPEC. This amounts to checking that the labeling scheme is not *ex post facto* and that the interpretation associated with the model must secure the truth of appropriate counterfactuals concerning the machine's behavior. In the above case, however, this seems rather problematic, since it is not clear at all what *the* standard model of a FSA is supposed to be. A FSA specifies a very general “architecture”, a system with an input device (without further details as to the nature and structure of this system *cum* device) both of which can be in different states. Therefore, many entities (in Copeland's sense), using a little imagination, are potential FSAs and in a way this is exactly what the

theorem above shows. In fact, I would claim that every physical system that consists of different states, and exhibits transitions between these states depending on some input to the system, would count as a standard model. This implies that the above result could be strengthened to an “honest model”, if Copeland’s two criteria were met, too. In the next section, after extending the above theorem to psychological theories, I will argue that they are indeed satisfied. Hence, every physical system is an honest model of every FSA.

### 4.3 The Universal Realization of Psychological Theories

Stated as is, Putnam’s Realization Theorem does not present a serious challenge to computationalism. After all, finite state machines without input and output capabilities lack any kind of interaction with their environments. This defect was repaired in the construction of the last theorem, which can be generalized to the realization of psychological theories.

In the following, I will assume a formal theory  $T$  whose language contains a connective ‘ $\rightarrow$ ’ together with a non-empty, finite set of constants  $Q$  and another finite set of constants  $I$  such that if  $i$  is in  $I$  and  $p, q$  are in  $Q$ , then  $(p,i)\rightarrow q$  is a well-formed formula (where  $Q$  can be viewed as the set of mental, or computational, or whatever, states and  $I$  as the set of inputs, for example). Copeland’s definitions have to be reformulated for (psychological) theories:

*Definition 4.2:* Given a formal theory  $T$ , a labeling scheme  $L$ , and a physical system  $S$ , the pair  $\langle S,L \rangle$  is a model of  $T$  iff all sentences of  $T$  are true of  $S$  under  $L$ . A physical system  $S$  that is a model of  $T$  is said to “implement”  $T$ .

*Definition 4.3:* A physical system  $S$  implements a function  $f$  iff there exists a labeling scheme  $L$  and a formal theory  $T$  (e.g., a specification of an architecture and an algorithm specific to the architecture that takes arguments of  $f$  as inputs and delivers values of  $f$  as outputs) such that  $\langle S, L \rangle$  is a model of  $T$ .

*Theorem 4.4:* Every ordinary open system  $S$  is a model of every formal theory  $T$  (in the language  $L = \langle Q, I, \rightarrow, \dots \rangle$  in the above sense).

*Proof:* Again, a labeling scheme  $L$  for  $S$  and an interpretation of ' $\rightarrow$ ' such that  $T$  is true under that scheme for  $S$  has to be defined. For part 1, we designate the boundary of  $S$  (for inputs) and its "inner" part (for the other states) as bearers of labels, to account for the fact that  $T$  has constants for inputs besides constants for states. For part 2, a method has to be exhibited for specifying the label borne by each label-bearing part at any given time:

Consider an arbitrary interval of real-time  $[t, t']$  and let the boundary of  $S$  be the "input region". Note that the environmental conditions on the boundary throughout  $[t, t']$  specify the input that  $S$  will receive. By Putnam's Principle of Noncyclical Behavior, "the state of the boundary of such a system is not the same at two different times" (Putnam 1988, p. 121). We need to define "physical states" for  $S$  and the boundary region of  $S$ , which can serve as designations of the constants from  $I$  and  $Q$ . The physical states, call them *interval states*, will be defined (analogous to Putnam) as sets of values of all field parameters at all points within the boundary or at the boundary of  $S$ , respectively, for a

given interval of real-time. These interval states need to be grouped together (using set union) to form state types such that each state type corresponds exactly to one constant.

We first define a mapping from interval states onto constants and then construct the labeling from this mapping: start by defining inductively an infinite sequence of consecutive intervals  $Int_0, Int_1, Int_2, \dots$ , where  $Int_0$  is  $[t, \frac{t+t'}{2})$  and  $Int_k$  is the open interval

$$[\frac{t + \sum_{j=1}^k \frac{t'-t}{2^j}}{2}, \frac{t + \sum_{j=1}^{k+1} \frac{t'-t}{2^j}}{2}) \text{ of real-time (for } k > 0). \text{ Map the interior of } S \text{ during the interval } Int_0,$$

call it  $P_0$ , onto the constant ( $q_0$  in  $Q$ , say) designating the state in which the system described by  $T$  starts out. Then, for every interval state  $P_k$  (defined by the interior of  $S$  during the interval  $Int_k$ ) corresponding to some  $p$  of  $Q$  do the following: first, define  $I_k$  to be the interval state of the boundary of  $S$  during the interval  $Int_k$ . Let  $I_k$  correspond to input  $i$  from  $I$  after  $k$  steps (\*). If the system described by  $T$  in state  $p$  transits into state  $q$  upon input  $i$ , define the “successor state”  $P_{k+1}$  to be the interval state of the interior of  $S$  during the interval  $Int_{k+1}$ . Now form physical state types by taking the union of all interval states  $P$  that correspond to a single constant (for all constants from  $I$  and  $Q$ ). (\*\*)  
For the resulting physical state types a 1-1 and onto labeling function from constants can be defined, which takes care of the second part of  $L$ . (\*\*\*)

To see that  $S$  is a model under the given labeling scheme  $L$  for  $T$  we need to find an interpretation  $[\ ]_L$  of ‘ $\rightarrow$ ’ such that  $[(p,i) \rightarrow q]_L = true$  in  $S$ . Take  $[\rightarrow]_L$  to mean “causes” (or “follows nomologically” by the laws of physics, i.e., field theory given environmental conditions). (\*\*\*\*)

If  $S$  is in state  $L(p)$  and the input to  $S$  is  $L(i)$ , i.e., during the interval  $Int_k$  the physical make-up of  $S$  is given as well as its boundary conditions, then by the laws of physics it would be possible for a mathematically omniscient being (a Laplacian supermind, see Putnam, 1988, p. 122-123 for details) to determine that  $S$  will be in state  $L(q)$  at the beginning of  $Int_{k+1}$ . Given the boundary conditions during  $Int_{k+1}$  (which correspond to the “next” input state and, thus, have to be provided), it can be determined that  $S$  will stay in state  $L(q)$  throughout  $Int_{k+1}$ . This concludes the proof that  $L$  and  $S$  are a model for  $T$  under  $[\ ]_L$ . ∴

Before I discuss critical points of the construction (indicated by asterisks) as well as consequences of this proof, I would like to point out two interesting features: 1) the interval of real-time  $[t, t']$  was used to fix the reference of constants, and 2) its choice is completely arbitrary, hence any interval will do. This implies that 3) the system can change states arbitrarily “fast”. In fact, physical systems will not only realize finite state automata, but also (countably) infinite state automata with a (countably) infinite input language (if  $T$  is taken to be a description of an automaton and the finiteness restriction on  $Q$  and  $I$  is dropped).

Yet another issue to be considered is the question what kind of function the system  $S$  implements? To answer this, define a function  $f$  from  $I^*$  into  $Q$  such that  $f(w)=q$  if the state  $q$  is reached after having read  $w$  from  $q_0$  for all strings  $w$  in  $I^*$  (this mapping can be

obtained inductively from ‘ $\rightarrow$ ’). Obviously,  $T$  takes arguments of  $f$  as inputs and delivers values of  $f$  as outputs, in Copeland’s sense, hence,  $S$  implements  $f$ .<sup>48</sup>

#### 4.4 Possible Objections to the Extended Version of Putnam’s Construction

I have marked four steps in the above proof that are open to criticism—I will now address them starting with the least difficult.

Step (\*\*\*) would be criticized by Chalmers, since the labeling scheme does not include states of the automaton that are not reached in the particular run (see Chalmers, 1996, p. 315). This requirement, however, can be satisfied by taking the “subinterval”  $[t, t']$  of  $[t, t'']$  (for  $t' > t$ ) and mapping all unreached states onto physical states within this interval.

In step (\*) reference is made to an input state which is undetermined and essentially depends on a particular input (i.e., the  $n$ -th input character). This, of course, is the trick that makes it possible to map the accidental boundary conditions of  $S$  at “run-time” to the  $n$ -th input character. Obviously, there is no *systematic* relation between inputs and boundary states, which a “good” theory of implementation should provide. Note, however, that the reference to this character is uniquely fixed for all possible inputs, and moreover that it is not an *ex post facto* assignment (which is, for example, Copeland’s critique of Searle’s construction, see Copeland 1996, p. 348).

Step (\*\*\*\*) marks a very critical point, the interpretation of ‘ $\rightarrow$ ’ (the equivalent of Copeland’s “ACTION-IS”) as “causes”. This can be considered one of the most

---

<sup>48</sup> Instead of letting outputs be internal states, I also could have designated a particular part of the boundary of  $S$  as “output region” and defined output states accordingly.

problematic questions in general, since there are many different possible interpretations of ' $\rightarrow$ '. I think that Copeland would agree with the claim that all state transitions of  $S$  are *caused*, since they were just so defined (see Copeland, 1996, p. 353). The question is whether counterfactuals are supported, i.e., whether the input had been  $L(i)$  and the current state  $L(q)$  at any time  $t$ ,  $S$  would have changed to state  $L(q')$  (according to the state table). I am not sure if counterfactual support can be directly required as a criterion, since even real systems under “different environmental conditions” will not support counterfactuals (e.g., a PC will stop working correctly if it is exposed to a strong magnetic field). Somehow it has to be specified when transitions obeying counterfactuals can be required and expected. In the end, this boils down to a notion of “normal operation”, which is problematic in its own ways as pragmatic agreements on the physical theory, background assumptions, etc. enter the picture (e.g., see Hardcastle, 1995)—I will say more about this in chapter 5.

As far as the above construction is concerned, one could argue that it does not really make sense to ask for counterfactual support, since  $L$  is defined every possible input string. Hence, if  $L(i)$  and  $L(q)$  were given,  $S$  would have changed to state  $L(q')$  by definition of  $L$ . If we asked, on the other hand, whether  $S$  in state  $P$  with input  $I$  had changed to the successor state  $P'$  (for interval states  $P, P', I$ ) without supplying  $L$ , then it would not even be clear what this question means (despite the fact that, given the boundary conditions  $I$  and the state of the system  $P$ , its next state  $P'$  will be “caused”). So, one could say that the counterfactual requirement is vacuously satisfied.



There are objections, however, to the underlying assumption that input states can be arbitrarily defined. Chrisley (1994, p. 415) points out that “for computational purposes, inputs and outputs are characterized in terms of their intrinsic properties.” Inputs and outputs cannot successfully be defined as in the above construction in a “post-hoc” manner as the boundary of a physical system without at the same time losing *the predictive computational understanding of the system*, because it is not known in advance which input state is going to become which boundary state. According to Chrisley (1994) this predictive nature of computational system is one of its characteristic properties.

There is certainly something true about the predictive power of “genuine” computational descriptions and I will take up this issue again later in this work. For now, one could answer that the system is just so defined that all boundary information has to be available through each interval in order to predict the boundary condition at the beginning of the next interval. Given the environmental influences during that interval in turn, it is possible to determine the next state, but again, only knowing all the environmental conditions. The reverse direction is certainly not true, but it cannot even be asked meaningfully because computational states only have correlates via  $L$ , and  $L$  in turn needs complete information about the development of the environment of  $S$  during the computation period. Thus, the system is lacking the predictive power of computational descriptions if  $L$  is not supplied—but that seems like a platitude, since no system without a labeling, i.e., a correlation between formal and physical descriptions, can be used to make predictions. The question remains, however, if it is possible at all to have this kind predictive power for descriptions at the level of fields. It seems that there would be too many different possible instantiations of one and the same computational state, so that it

would even be difficult to argue that genuine computational systems can “predict” the states of fields by virtue of their future computational states (see also the last section of this chapter).

Another objection concerning the involved notion of causation, brought forward by Chrisley, is that, being a physical notion of causation, it is too “weak” in that it would view events causally connected that “in everyday life and science other than mathematical physics we would not take to be causally related” (Chrisley, 1994, p. 414). Chalmers too rejects the “physical notion of causation” and requires that the transition be “reliable” regardless of environmental influences (see Chalmers 1996, p. 313, where he argues that Putnam’s construction fails to guarantee the reliability of state transitions). Melnyk as well as Copeland argue for a state transition relation that satisfies certain counterfactuals (see Melnyk 1996, p. 398, and Copeland, 1996, p. 351).

Besides the fact that a counterfactual based notion of causation is problematic in its own right, the interpretation of ‘ $\rightarrow$ ’ as a physicist’s notion of causation seems in my view appropriate in the above construction, because at the level of physical fields *no other notion of causality can be successfully employed* (see also Putnam 1988, p. 97). A different notion of causality would also require a different level of description, thus changing the problem at hand. In fact, the ordinary language notion of causation might have even more problems connected to it than the “physical” (e.g. regarding the description of real world systems), since it presupposes an ontological division of the world that physicists as well as philosophers might not be committed to.

Step (\*\*), finally, is the one I find most critical: the formation of physical state types. As I have already mentioned above, these types are considered legitimate physical states by the physical theory used to describe the system. Yet, it seems that something about them is very “unnatural”. One would like to restrict type formations to types that have “something” in common, something other than what is described by the constant in  $T$ , which they realize.

The philosophical strategy of this kind of “disjunctive type formation” is to turn the *multiple realizability argument* against itself: while the multiple realizability argument was invented to show that one and the same computational/psychological description can have very different kinds of physical realizations that need not have anything interesting in common, it is used here to show that *too many* different physical systems can be viewed as sharing the same computational/psychological properties. This problem on one hand, known to philosophers as the “disjunction problem”, and “multiple realizability” on the other are two sides of the same coin, the coin called functionalism. Being dual principles, they can be used to defend and defeat functionalism at the same time; in other words, they are useless to establish either point. Functionalism, and in particular computationalism, could only be defended, if one can find individuation criteria for physical state types in a non-circular manner—this is the challenge for cognitive science.

In a sense, the above construction exhausts possible extensions of Putnam’s idea, yet there are still many open ends left, probably too many for most philosophers to be able to focus on what I believe to be the core of Putnam’s argument. Therefore, a substantial modification of the above construction (or a completely new argument) is needed, which

essentially retains problem (\*\*) while dismissing all other points of critique. Only in that way will it be possible to locate the true source of all complications with the notion of implementation. In the following, I will present such a construction, which is not susceptible to charges such as “wrong notion of causality”, “unreliable state transition”, “unconnected inputs and outputs”, etc. because of the nature of its “physical states”, and pinpoint its crucial deviation from the one above.

#### **4.5 The State-to-State Correspondence View of Implementation**

So far, we can conclude that the SV, the credibility of which essentially depends on choosing the right kind of interpretation function to sort out “honest models” in the sense of Copeland (1996), is on par with the (naïve) CV (as implicit in Putnam 1988, for example). The last theorem showed how these two views are related: both require at some point a mapping between “formal” and “concrete entities”, the former between labels and parts of the machine, the latter between computational and physical states. What the theorem suggests, *in nuce*, is that *there is no essential difference between labeling parts or defining states of a system* (as far as the implementation issue is concerned). Both require that one can discern and define spatio-temporal physical regions of a physical system. Actually, both complement each other; they are two sides of the same coin. Consequently every description requires both: when Copeland makes out parts in a physical system to which he then attaches labels, he needs to take into account different “states” of these parts—this is reflected by the fact that there will be different labels for these states (high voltage vs. low voltage, see Copeland 1996). On the other hand, when Putnam defines physical states of a system, he has to specify which

parts of the system will be considered (in his case the whole region inside the boundary of the system). So, parts and states are intrinsically intertwined and one cannot have one without the other. Of course, this analysis depends crucially on the definitions of “spatial parts” as well as that of “temporal states”. For now, I will simply assume common sense notions of both without delving into philosophical details.

One major problem with the extension of Putnam’s theorem was that the notion of causation it involves is not satisfactory to many people (see Christley, Chalmers, Bringsjord, et al.). The “physical notion of causation”, so it is argued, is certainly not what we use when we describe the behavior of machines, in particular computing machines. We want to be able to say that if the machine *were to be* in state  $q$  and  $q \rightarrow p$  is one of its state transitions, this *would cause* the machine to transit into state  $p$ . In other words, “the state transition has to be actual and counter-factual” (Melnyk, 1996). It is clear that no extension of Putnam’s construction will be able to do this, because *it was just so designed as to not have counterfactual supporting state-transitions*. Instead, the state-transitions were accidental, depending on the environmental influences and conditions.

The crucial question with this strong requirement about computational state transitions is how they could be related to the physical? A widespread idea suggests a functional correspondence between a set of physical states and a set of computational states. The exact properties and restrictions of this correspondence depend on the respective definitions and differ slightly from author to author (e.g., Chalmers, 1996, Melnyk, 1996, Cummins, 1988, Copeland, 1966, McLennan, 1994, Endicott, 1996), but

the general idea is that “the causal structure of the physical system mirrors the formal structure of the computation” (Chalmers, 1994, p. 392) and vice versa. What “mirrors” means in this context, and if this state-to-state correspondence can solve the problems of the naïve CV, will be examined in the following sections.

#### 4.6 An Analysis of Chalmers’ Definition of Implementation

Chalmers (1994, 1996) is among those who provide an *explicit definition* of “implementation.” His basic conception (1994, p. 396) of how a computation is connected to the physical is that of an isomorphism: state transitions in the computation are to be in isomorphic correspondence with reliable causal transitions between physical states— $f(\rightarrow)$ —“reliably causes” (where ‘ $\rightarrow$ ’ is the formal state transition relation in the computation, an automaton, for example). To be precise, Chalmers (1994) actually provides two definitions of implementation, an informal and a (more) formal one, which he holds equivalent. It is worthwhile examining both, since they are not only cast differently, but also differ semantically. Since a precise definition of the mathematical concept of isomorphism is needed to facilitate the later discussion, I shall review it at this point:

*Definition 4.5:* [Isomorphism] Let  $M_1 = \langle D_1, R_1 \rangle$  and  $M_2 = \langle D_2, R_2 \rangle$  be two structures with domains  $D_1$  and  $D_2$ , respectively, where relation  $R_1$  is defined over  $D_1 \times D_1$  and relation  $R_2$  is defined over  $D_2 \times D_2$ . These structures are then said to be *isomorphic* if there exists a bijective function  $f$  from  $D_1$  to  $D_2$  such that for all  $x, y \in D_1$ :  $R_1(x, y) \Leftrightarrow R_2(f(x), f(y))$ .

For easier reference, I will split this last equivalence into the two implications

$$[\text{iso}\Rightarrow] \quad R_1(x,y) \Rightarrow R_2(f(x),f(y))$$

$$[\text{iso}\Leftarrow] \quad R_1(x,y) \Leftarrow R_2(f(x),f(y))$$

Chalmers' first informal definition (which itself consists of two phrasings) reads as follows:

“A physical system implements a given computation when the causal structure of the physical system mirrors the formal structure of the computation.

In a little more detail, this comes to:

A physical system implements a given computation when there exists a grouping of physical states of the system into state-types and a one-to-one mapping from formal states of the computation to physical state-types, such that formal states related by an abstract state-transition relation are mapped onto physical states-types related by a corresponding causal state-transition relation.” (Chalmers, 1994, p. 392)

There is a little ambiguity in the two phrasings as to what the exact meaning of “mirrors” is supposed to be. *Prima facie* one would expect “mirrors” to mean something like “is isomorphic to,” as “mirrors” usually indicates sameness in structure. Yet, the second phrasing does not seem to imply structural sameness, since it requires only [iso $\Rightarrow$ ] (i.e., that “formal states related by an abstract state-transition relation are mapped onto physical states-types related by a corresponding causal state-transition relation”), but not

the other direction [iso $\Leftarrow$ ] (i.e., that physical states-types related by the corresponding causal state-transition relation have to be mapped onto formal states related by an abstract state-transition relation, too). Hence, with “mirrors” Chalmers seems to mean only [iso $\Rightarrow$ ]. At a different place in the text, however, he suggests [iso $\Leftarrow$ ] when he writes “... that the formal state-transitional structure of the computation mirrors the causal state-transitional structure of the physical system” (1994, p. 393). So, it seems that “mirrors” is to be understood as “isomorphic to,” and, indeed, he later writes: “the relation between an implemented computation and an implementing system is one of isomorphism between the formal structure of the former and the causal structure of the latter” (1994, p. 396).

Hence, the mapping between physical state types and computational states has to be bijective (*one-to-one* and *onto*) and preserve the abstract state transition relation in order to give rise to an isomorphism between computational and physical state types. Note that 1) the existence of a *grouping of physical states of the system into state types* is required as a necessary prerequisite for the mapping to work and 2) that there are no restrictions imposed on the grouping; *its mere existence is sufficient*.

Before I present Chalmers’ second (formal) definition, a remark on the use of the term “state” seems appropriate. The term “state” in this context is sometimes used in the sense of “token of a particular state type”, although this is at best ambiguous and misleading. “Automaton state”, for example, could denote a state in the set of states in the definition of the automaton as well as a state in a particular run of the automaton—the former is a



type, the latter a token.<sup>49</sup> In physical systems, “physical state”, as was pointed out already in chapter 2, refers to the particular physical makeup of a system at a time (under certain environmental conditions): if the system is described in terms of a system of differential equations  $O_m(t)=F_m(I_1(t),\dots,I_n(t),P_1(t),\dots,P_k(t))$  (for finitely many  $m$ ), then by fixing the time parameter (e.g., at  $t_{17}$ ) one obtains the state of the system (i.e., by fixing the environmental conditions  $I_1(t_{17}),\dots,I_n(t_{17})$ , one obtains  $P_1(t_{17}),\dots,P_k(t_{17})$  as well as  $F_m(I_1(t_{17}),\dots,I_n(t_{17}),P_1(t_{17}),\dots,P_k(t_{17}))$  for all  $m$ ). For example, physical states in field theory would correspond to the value of all field parameters at a given time. This notion of state is independent of *how often* it is instantiated by the system (if at all).

Yet, some philosophers still use “state” as if it referred to a unique particular physical occurrence, a constellation of a physical system which obtains only once at a particular moment in time, and thus once in the life-time of the system.<sup>50</sup> While nothing can be identical to this particular occurrence (and it, therefore, does not make sense to say things like “this occurrence is the same as  $x$ ”), the above usage of “state” supports a notion of “sameness” (e.g., the system was in the same state at time  $t_{17}$  and  $t_{17}$ ). Thus, a physical state is not a (concrete) token of some physical state type, but rather a type itself.

To avoid terminological (and consequently conceptual) confusion, I will use the term “instantiation of a state” (or maybe “state token”, e.g., see Melnyk, 1996) to refer to a unique physical occurrence, and the term “state” in all other cases. The term “physical

---

<sup>49</sup> Example: “The automaton transits from state  $A$  to state  $B$  on input  $a$  producing output  $b$ ” for the type and “After five inputs the automaton is in state  $A$ ” for the token interpretation.

<sup>50</sup> A reason for this conceptual conflation might be that some physical systems might assume or instantiate any state (type) only at most once. For example, Putnam’s construction was essentially based on a physical principle that guaranteed that (open) physical systems are always in different physical states at different times.

state type” will be used to stress that a particular physical state has been obtained by type formation from “simpler states (state types)”.

After having explicated the overall structure of “implementation” in the first definition, Chalmers spells out the details in a more formal definition in which he uses a finite state automaton (FSA) as a representative for other computational formalisms:

“A physical system  $P$  implements an FSA  $M$  if there is a mapping  $f$  that maps internal states of  $P$  to internal states of  $M$ , inputs to  $P$  to input states of  $M$ , and outputs of  $P$  to output states of  $M$ , such that: for every state transition relation  $(S,I) \rightarrow (S',O')$  of  $M$ , the following conditional holds: if  $P$  is in internal state  $s$  and receiving input  $i$  where  $f(s)=S$  and  $f(i)=I$ , this reliably causes it to enter internal state  $s'$  and produce output  $o'$  such that  $f(s')=S'$  and  $f(o')=O'$ .” (Chalmers, 1994, p. 393)

Note that nothing in this definition requires that the mapping  $f$  have to be one-to-one, the reason being that imposing injectivity on  $f$  does not seem justifiable in the light of multiple realization arguments. Just consider an OR-gate, where the potential, which is supposed to correspond to “1”, fluctuates between 4.5 and 5.5 Volts (where 5 Volts would be the “ideal” voltage). In this case, there is a whole *set of physical states*, which are all different from each other, yet “similar enough” to be rightfully taken to correspond to “1”, as practice shows. Hardware designers do it all the time; they produce functioning machines whose computational description works both as explanation and prediction of the machine’s behavior. Yet, such a correspondence between one computational state and many physical states would be excluded by the restriction that  $f$  be one-to-one.

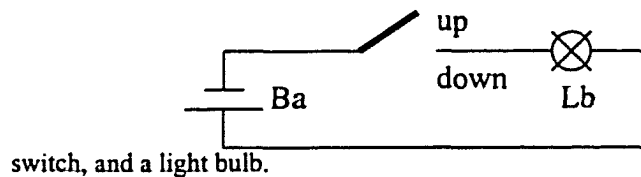
This is where the idea of a grouping of physical states (mentioned in the former definition) comes to reconcile two seemingly incompatible ideas: 1) that no one-to-one

correspondence can be established at the level of physical states (for the above and similar reasons), and 2) that the computational description is supposed to mirror the causal structure of the physical system. By relating not the physical states themselves, but more complex types of these states (formed by the “particular grouping” of states into types), it becomes possible to establish a one-to-one and onto relationship between these types and computational states, which is the prerequisite of an “isomorphism” (the mathematical term expressing structural identity, i.e., “mirroring”).

Chalmers, although never explicitly, seems to suggest that it is possible to obtain an isomorphic mapping  $f^*$  from  $f$  by collecting all those states  $s$  to form a state type  $s_i$  that are mapped onto the same automaton state type  $S_i$  according to  $f$ : “The state-types can be recovered, however: each  $[s_i]$  corresponds to a set  $\{s|f(s)=S_i\}$  for each  $S_i \in M$ . From here we see that the definitions are equivalent. The causal relations between physical state-types will precisely mirror the abstract relations between formal states.” (Chalmers, 1994, p. 393) One would define  $f^*$  from physical state types onto computational state types such that  $f^*(s_i)=S_i$  for each  $S_i \in M$ . This mapping is *one-to-one* because the physical state types have just been defined as such, and it is, furthermore, *onto* (ensured by the “for every state transition relation”-clause in the definition), but it is *not isomorphic* (since  $[\text{iso} \Rightarrow]$  does not hold as I will show in the following).

Let us examine how Chalmers’ definition works in detail, starting with a simple physical system  $P_I$  (e.g., described in circuit theory—see Figure 4.3) for which we can easily specify physical states types: switch  $SwI$  is connected to light bulb  $Lb$  and battery  $Ba$  by copper wires.

Figure 4.3 The simple physical system  $P_I$  consisting of a battery, a



Input to  $P_I$  consists in switching  $SwI$  from either “up” to “down” or vice versa (the states are named ‘1u’ for “ $SwI$  upwards”, ‘1d’ for “ $SwI$  downwards”). The internal states of  $P_I$  are the states of  $SwI$  (‘u’ for “up”, ‘d’ for “down”). Finally, output produced by  $P_I$  are the states of the  $Lb$  which is either lit or not lit (‘+’ for “light on”, ‘-’ for “light off”).

Now consider the following automaton  $M_I = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ , where  $Q = \{A, B\}$  is the set of inner states,  $\Sigma = \{a, b\}$  the input alphabet,  $\Gamma = \{0, 1\}$  the output alphabet,  $\delta = \{ \langle \langle A, a \rangle, \langle B, 1 \rangle \rangle, \langle \langle B, b \rangle, \langle A, 0 \rangle \rangle \}$  the transition function from states and inputs to states and outputs,  $q_0 = A$  the start state, and  $F = \{A, B\}$  the set of final states (which in this case does not really matter). The automaton is depicted (in the standard fashion) as a graph in Figure 4.4, where nodes denote states and edges denote transitions between states, both labeled accordingly (the format for edge labels is “input/output”). For the rest of this section, I will use graphs to represent automata instead of the more tedious mathematical notations.

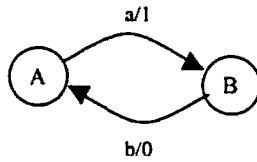


Figure 4.4 The automaton  $M_1$  with inputs from  $\{a,b\}$  and outputs from  $\{0,1\}$ .

Automaton  $M_1$  transits from state 'A' on input 'a' to state 'B' outputting '1', and from state B on input 'b' to state 'A' with output '0'. It follows that  $P_1$  implements  $M_1$  according to Chalmers' definition; just map (every occurrence of) "u" to 'A', (every occurrence of) "d" to 'B', (every occurrence of) "1d" to 'a', (every occurrence of) "1u" to 'b', (every occurrence of) "+" to '1', and finally (every occurrence of) "-" to '0'. The resulting mapping obviously satisfies all conditions of the definition, because it supports counterfactuals, or in Chalmers' terms "strong conditionals": "If a system is in state *A*, then it will transit into state *B* [on input 'a'], however it finds itself in the first state". (Chalmers, 1996, p. 316—capitalization of state names and the remark in brackets are mine). Both of Chalmers' requirements for counterfactual support, that the transition be *lawful* and *reliable*, are satisfied by  $P_1$  according to accepted physical theories (i.e., circuit theory). In particular, I would like to stress the *reliability* of state transitions of *all systems devised in this chapter*, because Chalmers holds that it is the reliability of state transitions which ultimately distinguishes implementation in his sense from Searle's (or Putnam's, for that matter): "The added requirement that the mapped states must satisfy reliable state-transition rules is what does all the work." (Chalmers, 1994, p. 396)

So far, the application of the definition has been straightforward, every automaton state type corresponds to one and only one physical state type in a direct way (there is no need for a complex grouping of physical states types). Consider now the physical system  $P_2$  consisting of two switches  $Sw1$  and  $Sw2$  connected to a light bulb and a battery by a copper wire (as depicted in Figure 4.5). Input to  $P_2$  consists in switching one of  $Sw1$  or  $Sw2$  from either “up” to “down” or vice versa. The four possible input states are, analogous to the previous notation, denoted by ‘1u’, ‘1d’, ‘2u’, and ‘2d’. The internal states of  $P_2$  are the four possible states of the switches (denoted by ‘uu’, ‘dd’, ‘du’, and ‘ud’, where the first letter indicates the state of  $Sw1$  and the second that of  $Sw2$ ). Output of the system is again the state of the light bulb (denoted as before).

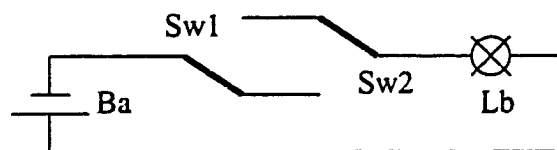
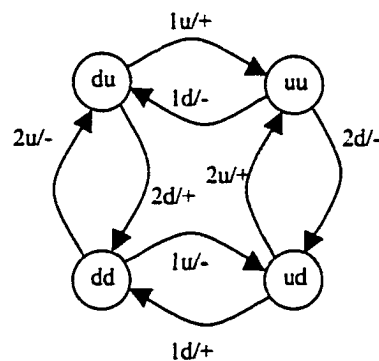


Figure 4.5 The physical system  $P_2$  consisting of a battery, two switches, and a light bulb.

The abstract structure of  $P_2$  for the above-defined input, inner, and output states, can be depicted as state-transition diagram, which also defines an automaton, call it  $M_2$ . The structure of this automaton is isomorphic to the causal structure of  $P_2$  (for the given physical states), hence,  $P_2$  implements  $M_2$ :

Suppose switch  $Sw2$  is never pressed; then it can readily be seen that  $P_2$  implements  $M_1$ , too. Starting in state “du”, the automaton can only transit between “du” and “ud”: ignoring  $Sw2$  simply turns  $M_2$  into  $M_1$ . Once  $Sw2$  is used, however, there is no mapping that can relate the above-defined states in  $M_2$  to states in  $M_1$ , since whether input “1u”

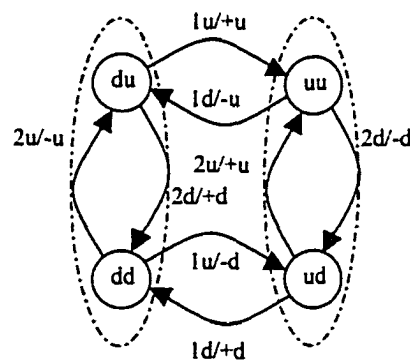
turns the light on or off depends on the state of  $Sw_2$ . I will sketch only part of the argument (since it is rather long and tedious given the number of possible mappings that one has to consider): take “du” to be the start state, which has to be set in correspondence with ‘A’. Then either “1u” or “2d” or both have to be mapped onto ‘a’, and as a consequence, either “uu” or “dd” or both will correspond to ‘B’. Suppose we map “1u” to ‘a’ and “uu” to ‘B’. Then “1d” have to correspond to ‘b’. But this is not possible. To see this, suppose that “ud” will be mapped onto ‘A’ (since both, “ud” and “dd” must correspond to some automaton state). Then “2d” will have to correspond to ‘b’, and consequently “1d” to ‘a’. Contradiction. So, “ud” cannot be mapped onto ‘A’, thus they must be mapped onto ‘B’. But then, the transition “1d/+” turn the light on, as opposed to ‘b/0’ which turns the light off in  $M_1$ . So this is not possible either. Hence, “1d” cannot correspond to ‘b’. It follows that “uu” cannot correspond to ‘B’ and “1u” not to ‘a’. Thus, “2d” must correspond to ‘a’, and so on...



**Figure 4.6** The causal structure of physical system  $P_2$ , which defines the isomorphic automaton  $M_2$ .

In the end, we establish that for the above-defined state types  $P_2$  does not implement  $M_1$ . However, if other state types are considered, then there exists a mapping  $f$  under

which  $P_2$  implements the automaton  $M_1$ : take the output to be the state of the light bulb *together with* the state of  $Sw_2$  (resulting in the four *distates* “+u”, “-u”, “+d”, and “-d”, where the first letter denotes the state of the light bulb and the second the state of  $Sw_2$ ). The idea here is to introduce extra output states so that the effect of pressing switch  $Sw_2$  can be ignored. This combination is physically legitimate, because all states are physically specifiable and support counterfactuals (it should thus be acceptable for Chalmers). In fact, every state will be acceptable as long as it can be specified within the physical theory that is used to describe the physical system (in this case circuit theory). Figure 4.7 depicts the causal structure of the two-switch system for the new states:



**Figure 4.7** A graph of the states and transitions in the two-switch system  $P_2$  after inputs, outputs, and states have been redefined. States encircled by a dashed line are mapped onto the same automaton state.

For the following, assume in addition that the input alphabet of  $M_1$  contains the symbol ‘c’ (this does not change the automaton, since ‘c’ is not used in any transition in  $M_1$ —see also below). Define  $f$  to be the following function:



Inner states	$f(\text{du})=A$	$f(\text{dd})=A$	$f(\text{ud})=B$	$f(\text{uu})=B$
Input states	$f(1u)=a$	$f(1d)=b$	$f(2u)=c$	$f(2d)=c$
Output states	$f(+u)=1$	$f(-d)=1$	$f(-u)=0$	$f(+d)=0$

It can be seen that for “ $(A,a) \rightarrow (B,1)$ ” and “ $(B,b) \rightarrow (A,0)$ ” (i.e., for every state transition) the conditional of Chalmers’ definition of implementation (i.e., “if the system were in state ..., it would transit into ...”) is true: take the first transition “ $(A,a) \rightarrow (B,1)$ ”. Two states of  $P_2$  correspond to state ‘A’, “du” and “dd”. Suppose  $P_2$  were in state “du”. Since the only input corresponding to ‘a’ is “1u”, then  $P_2$  would transit reliably into state “uu” (corresponding to ‘B’) and produce output “+u”, which corresponds to ‘1’. Similarly, if the system were in state “dd”, then  $P_2$  would transit reliably into state “ud” (corresponding to ‘B’) and produce output “-d” (corresponding to ‘1’). The same holds true for the second transition. Hence, according to Chalmers’ definition,  $P_2$  implements  $M_1$  under  $f$  (even if it does this in an admittedly strange way).

This leaves us with a very interesting situation:  $P_2$  implements  $M_1$  (under the above-defined function  $f$ ) and  $P_2$  also implements  $M_2$  (which has a different, more complex transition structure than  $M_1$ ). Assuming that automata are the appropriate formalism to reflect the causal structure of a physical system, one can reach two (not necessarily exclusive) conclusions: 1) physical systems can have *multiple* causal structures, which depend on the grouping of physical states (i.e., the level of description for a given set of physical states), or 2) Chalmers’ definition has to be modified to reflect *the* causal structure of the system determined by the given set of physical states.

The former does correspond to our intuition that the same physical system (i.e., the same spatio-temporal region) can be described at different levels according to different notions of physical state appropriate for that level. Note, however, that one is limited to groupings (that is, combinations) of the given physical states in the above case, which are, of course, limited to groupings that allow for reliable transitions between them (otherwise Putnam's construction could sneak in the backdoor).<sup>51</sup>

The latter conclusion points in another direction: if computational descriptions are to mirror the causal structure of physical systems, then the causal topology of a set of physical states *alone* should be *necessary* and *sufficient* to specify the "corresponding" computational description. Although this approach does not leave any room for groupings of states into state types *within* the definition of implementation, it can still account for all of the potentially different computational descriptions obtained from Chalmers' definition by simply considering the groupings of states (implicit in Chalmers' definition) *as states* of a physical system (e.g., the same system at a different level of description). That way the second leaves open the possibility that physical systems might have more than one causal structure (depending on possible groupings of its states), while removing the burden from the theory of implementation to decide which of these groupings are legitimate and returning it to the theory that delivered these states in the first place.

---

<sup>51</sup> This might not allow one to "jump" to upper levels, where physical states (defined at that level) are not (known to be) combinations of "lower-level" states.

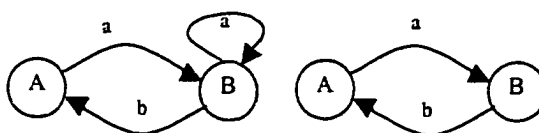
It seems that both conclusions have their own virtues. However, I will show in the following that the first (in my view) also incorporates irreparable flaws, while an extension of the second can (flawlessly) combine the advantages of both.

#### 4.7 A Modification of Chalmers' Notion of Implementation

Recall that Chalmers' conception of implementation is that "the relation between an implemented computation and an implementing system is one of isomorphism between the formal structure of the former and the causal structure of the latter" (Chalmers, 1994, p. 396). An isomorphic correspondence between the computational and the physical structure is required, if computation is to be an "[...] abstract specification of causal organization" (Chalmers, 1994, p. 396). However, nothing in Chalmers' definition ensures that the formal state-structure of  $M_1$  mirrors the causal state-transitional structure of  $P_2$ . The reason for this is that there are transitions within the physical system that do not correspond to any transition in the formal computation, such as " $(du,2d) \rightarrow (dd,+d)$ " (the formal pendent would be " $(A,c) \rightarrow (A,0)$ ").<sup>52</sup> So, although all physical states are mapped onto formal states, not all (reliable) causal relations among these states are captured by the formal system. In short,  $[iso \Rightarrow]$  is not ensured to hold by his definition.

---

<sup>52</sup> One could object that this argument is solely based on a formal trick, namely on adding a new "dummy" character to the alphabet of the automaton to which unused inputs can be mapped. This objection could be countered by pointing out that a new character is not really necessary, since inputs could have been mapped to 'ε', too (ε-transitions are part of the standard equipment in automata theory). A better response, however, is to present two automata with the same alphabet that illustrate the same point (namely the omission of a transition in one automaton, which is supposed to be implemented by the other):



One solution, assuming that all (finitely many) inner physical state types are defined and given, would, therefore, be to define ' $\rightarrow$ ' to be the set of all causal transitions from inner physical state types plus physical input types to other inner physical state types plus physical output types for all possible physical state, input, and output types of the system. The union of the first projection of the first projection and the first projection of the second projection of  $\rightarrow$  would then be the set of computational inner states types, the second projection of the first projection would be the set of computational input types, and the second projection of the second projection the set of computational output types. In this way input, output, and inner states (of an automaton, for example) would be derived directly from a physical description of the system. The structure of the resulting automaton would necessarily be isomorphic to the causal structure of the physical system. Hence, it is always possible to find an isomorphic computation for any physical system (which is unique up to renaming).

Even if one does not want to take computational states to be *sets of physical states*, Chalmers' definition can be modified to reflect the stronger requirements [iso $\Leftarrow$ ] and [iso $\Rightarrow$ ] if physical states are assumed as given—this corresponds to the second conclusion, which does not involve groupings of states into types, but only the states themselves (note that I used the term “realize” instead of “implement” for reasons that will become clear shortly):

*Definition 4.6:* [Realization of an FSA] A physical system  $P$  realizes an FSA  $M$  if there is a bijective mapping  $f$  that maps internal state types of  $P$  to internal states of  $M$ , input types to  $P$  to input states of  $M$ , and output types of  $P$  to output states of  $M$ , such that:

1. For every  $s, s', i, o$  the following conditional holds: if  $P$  is in internal state  $s$  and receiving input  $i$  where  $f(s)=S$  and  $f(i)=I$  and this reliably causes it to enter internal state  $s'$  and produce output  $o'$  such that  $f(s')=S'$  and  $f(o')=O'$ , then  $(S,I) \rightarrow (S',O')$  is a state transition of  $M$ . ([iso $\Rightarrow$ ])
2. For every  $S, I, S', O'$  the following conditional holds: if  $P$  is in internal state  $s$  and receiving input  $i$  where  $f^{-1}(S)=s$  and  $f^{-1}(I)=i$  and  $(S,I) \rightarrow (S',O')$  is a state transition of  $M$ , this reliably causes it to enter internal state  $s'$  and produce output  $o'$  such that  $f^{-1}(S')=s'$  and  $f^{-1}(O')=o'$ . ([iso $\Leftarrow$ ])

This modification comes at a price, however, since  $M_2$  is the *only* automaton (type) up to renaming that *mirrors* the causal structure of  $P_2$  (under this notion of implementation): on this new definition  $P_2$  no longer “realizes” (or in Chalmers’ terminology “implements”)  $M_1$ . In other words, if the causal structure of a physical system is completely described by an automaton for a given set of physical states, then there is no room for the system to realize another automaton, except if the set of physical states is altered (e.g., by grouping them). One should be willing to pay this price if one obtains a computational description of the causal structure of a physical system (for the given states) in exchange.

Chalmers, however, wants to allow physical systems to implement all kinds of *simpler* computations, since he believes that there is “no canonical mapping from a physical object to the computation it is performing.” (1994, p. 397) This is a result of his conflating two conceptually distinct steps or operations: 1) the formation of physical state types in terms of which the implementation mapping will be defined; and 2) the specification or establishment of that mapping itself. Whereas in the second phrasing of his first definition the existential quantifier ranges over mappings *and* groupings of physical state tokens into types, he abandons this separation in his second definition by establishing a mapping from physical states to automata state types. This leads him to accept that “within every physical system there are numerous computational systems. To this limited extent, the notion of implementation is ‘interest-relative’” (1994, p. 397). This conclusion seems to be counterproductive to the computationalist program of describing the causal structure of a physical system computationally—every physical system will now have many different causal structures at the same time! How are we going to find the “right” one, or if there is no “right” one, then one necessary for the possession of a mind, say? I do not want to repeat Chalmers’ rhetorical tactic of using quick estimates to impress the reader, but it seems that the chances of figuring out the correspondence of interesting automaton/automata in the brain that give rise to mind are almost zero, if the formation of physical state types is not given and thus, within limits, completely arbitrary (e.g., take the states of axons of every neuron every millisecond for fifty years to be the physical state tokens which can be arbitrarily combined to form types, for example, the state of neuron\_14554 at 121232 milliseconds and the state of neuron\_9834511 at 4365354 milliseconds after birth form a state type).

My view is that the formation of physical state types should not be allowed to be considered *interest-relative*; otherwise systems like Putnam's will count as computing too. Or to put the same point another way, if one allows the formation of physical state types to be arbitrary and unconstrained, then the conceptual "work" for which we turn to the notion of implementation will simply devalue, without illumination onto the question of how we individuate physical state types. On the contrary, one would like to allow very limited possibilities for the formation of "reasonable physical state types", *derivable within the physical theory* describing the system's behavior: physical state types are theory-relative! Only severe physical restrictions of physical state type formation *together* with the above canonical mapping can rescue the notion of implementation from being observer/interest-relative. I will show below that Chalmers gains nothing by emphasizing the need for "strong, reliable state transition conditionals", if he at the same time ignores the formation of physical state types.

Even if one were to grant Chalmers his "interest-relative" interpretation of implementation that views physical systems as implementing "simpler" computations at the same time, there is problem with his view of "simpler". While it is common in complexity theory to measure the complexity of a computation in terms of "the length of the computation" (i.e., number of computational steps as function of the input size), some people use the number of states of an automaton as a complexity measure (e.g., Chaitin). Although not made explicit in his paper, Chalmers too uses the number of states as a measure of complexity (of computations) when he allows physical systems with  $n$  physical states to implement automata with  $n$  or fewer states.

What is problematic with such a complexity measure, however, is that there are automata with a huge number of states that compute very *simple functions* while other automata with only few states compute very *complex functions* (here the terms “simple” and “complex”, attributing properties to functions, are to be understood in the sense of standard complexity theory). In short, the number of states *per se* is not good measure of the complexity of the function computed by an automaton. This is why usually an additional requirement is added: the complexity of a function is defined to be the number of states of the smallest (in terms of states) automaton that computes this function. Even if the smallest automaton is not unique, there will always be a smallest number of states. What is needed for this kind of definition, however, is a way of comparing automata with respect to the functions they compute. The simplest way of doing this is just to compare the input-output functions of two automata and check if they are the same. However, this is not sufficient in the current case, as we are not only interested in the input-output function, but also in the sequence of steps, i.e., in the transitions involved in computing it—if we were not interested in modeling the causal structure of a physical system, the internal computational structure of an automaton would not matter. Therefore, we need a way of comparing the inputs and outputs as well as the sequence of state transitions needed to compute the function.

To illustrate this problem, consider the following automaton  $M_3$ :



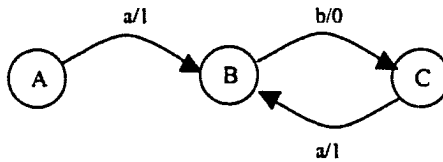


Figure 4.8 A graph of automaton  $M_3$ , which is bisimilar to  $M_1$ .

This automaton obviously computes generates the same language as  $M_1$  (if one assume B to be a final state, for example). Not only do they compute the same function, but in addition they can be regarded as “almost isomorphic” (i.e., “the same from the outside”, see Barwise and Moss, 1996, p. 37), in that there exists a non-empty relation  $R$  between  $M_1$  and  $M_2$  (called *bisimilarity*) such that:

1. if  $R(S_1, S_2)$  and  $(S_1, I) \rightarrow (T_1, O)$  in  $M_1$ , then  $(S_2, I) \rightarrow (T_2, O)$  in  $M_2$  for some  $T_2$  and  $R(T_1, T_2)$
2. if  $R(S_1, S_2)$  and  $(S_2, I) \rightarrow (T_2, O)$  in  $M_2$ , then  $(S_1, I) \rightarrow (T_1, O)$  in  $M_1$  for some  $T_1$  and  $R(T_1, T_2)$

To put this definition into plain English: two automata are considered bisimilar if states from one automaton can be associated with states from the other in such a way that for every state transition in one automaton there is a corresponding one in the other which leaves both automata in states that are associated and vice versa. If a physical system implements a particular automaton  $M$ , then it seems it should also implement *all* automata that are *bisimilar* to  $M$ : not only from a computational perspective because the input-output behavior is the same, but also from an implementational point of view, because the number of transitions necessary to obtain a certain output for a given input is the same (for all possible inputs).

It seems clear that Chalmers intuitions lean in this direction when he claims that all other computational formalisms can be expressed in terms of his combinatorial state automaton (CSA) and that these other computations are implemented by physical systems if their corresponding CSAs are implemented by those systems. In other words, if a physical system implements a certain CSA, then all automata that compute the same function as this CSA can “be viewed as being implemented as well” (note that Chalmers does not define how one would “view” them as being implemented via the “corresponding CSA”). In particular, a Turing machine with 10 times as many states as the CSA and an arbitrary long tape can still be viewed as being implemented by the physical system as long as its input-output function is the same as the one of the CSA. What remains unclear in this context, however, is *why this relationship is unidirectional*, i.e., why no CSA *with more states* (than the one implemented by the physical system) that computes the same function is viewed as being by his definition of implementation.

Suppose a physical system  $P$  (which has three physical states defined) implements  $M_3$  by virtue of a function  $f$  which *is* an isomorphism.  $P$  will also implement the bisimilar automaton  $M_1$  (using a function  $f'$  which *does not* give rise to an isomorphism). But it does not implement the bisimilar automaton  $M_4$  (according to Chalmers' definition):<sup>53</sup>

---

<sup>53</sup> I have to specify what I mean:  $P$  does not implement  $M_3$  in a “straightforward” manner taking internal states like “switch on”, “switch off” (i.e., without defining physical states, internal as well as input/output states, that depend on additional factors such as time, etc., for example “switch up at 10 p.m.” as opposed to the “timeless” “switch up”).

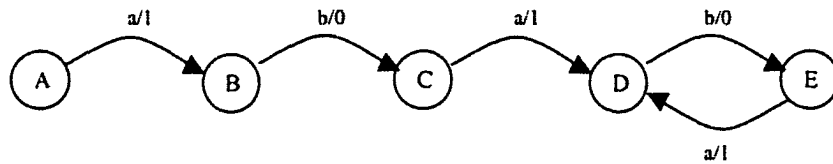


Figure 4.9 A graph of automaton  $M_4$ , which is bisimilar to  $M_1$ .

Why should a physical system implement bisimilar computations with fewer states and not those with more states? This is not clear at all! Remember that the requirement at the heart of Chalmers definition of implementation, which is thought to block Putnam-like constructions, is the that of *reliable state transition*: for every state transition between to computational states there has to be a *reliable* (counterfactual supporting) state transition between physical states associated with those computational states. Additionally, he required that the association be functional—different computational states cannot be associated with the same computational state. But that requirement seems quite arbitrary.

Consider an optimizing compiler that detects that two constants (or variables, for that matter) have the same value at all times in a given computation and maps both constants onto the same machine register (on a given machine). Are we now supposed to think that the program is not implemented any longer (because of the optimization step), while the same program compiled by a non-optimizing compiler would count as being implemented?

It seems that a physical system should *either* implement *one* computation (i.e., the one which is *isomorphic* to its causal structure for the given physical state types) or otherwise *all bisimilar* computations; if it does not get the causal structure right, then at

least it will get the input/output function right—always assuming, of course, that the physical states and their causal connections are fixed.

So, on the one hand, there are reasons to distinguish isomorphic computations from other bisimilar computations (they “mirror” the structure causal topology defined by the given physical states), and on the other hand there are reasons to view them as all being implemented by the same system (they “mirror” the computational sequences of transitions for every input). To put it into a slogan: while bisimilar computations get *the causal dynamics* of every possible computation of the physical system right (supporting counterfactuals!), the isomorphic computation gets its *causal structure* right in addition!

Fortunately, there is a simple definition of implementation that accounts for both aspects of computations (bismilar and isomorophic) by building upon the above definition of “realization of an FSA” (definition 4.4):

*Definition 4.7:* [Implementation of an FSA] A physical system  $P$  implements an FSA  $M$  in case  $M$  is bisimilar to the automaton that  $P$  realizes.

#### **4.8 A Pitfall for Correspondence Theories: The Slicing Theorems**

Returning to the issue of physical state types, I have already mentioned that what is considered “physical state” is already a kind of type. The switch, for example, in “totally up” position encompasses all (meta)physical occurrences of “the switch being totally up” (e.g., times at which the switch is in that position), hence, “switch totally up” is a physical state type. This has to be distinguished from types like “almost up” or “closely enough up”. One might want to count the former as well as the latter as part of a more abstract

type “switch up”. In that case, however, one has to define criteria for the formation of these more abstract types from (basic) physical types. Chalmers does not provide any criteria, neither for the formation of (basic) physical types nor for more abstract types. In his first definition, he is satisfied with the mere existence of a grouping of types into more complex types. In his second definition, he simply notes that his “[...] definition uses maximally specific physical states  $s$  rather than the grouped state-types” (Chalmers, 1994, p. 393). What exactly “maximally specific physical states” are is left open. Since time is left out in his definition, it must obviously enter here as part of the “physical state”, but it is not clear how. This too, makes it seem as if there need not be any resemblance between different physical states that can be grouped together to form a type as long as the function  $f$  maps them onto the same  $S_j$ . Obviously, this opens doors to all kinds of wild implementations!

Consider system  $P_I$ , augmented by the temporal attributes “on weekdays” and “on weekends”, so that pushing the switch upwards/downwards on weekends can be distinguished from pushing it upwards/downwards on weekdays. The same distinction is made with respect to internal states: the switch can be in up/down position on weekdays and on weekends. Note that the system, call it  $P_{I,4}$ , automatically changes states Fridays and Sundays at midnight, *without further ado*. The internal states of  $P_{I,4}$  are then “switch down on weekdays” (denoted by ‘ $\downarrow d$ ’), “switch up on weekdays” (denoted by ‘ $\uparrow d$ ’), “switch up on weekends” (denoted by ‘ $\uparrow e$ ’), and finally “switch down on weekends” (denoted by ‘ $\downarrow e$ ’). The time-dependent input states are “push Sw1 upwards on weekdays” (denoted by ‘ $S\uparrow d$ ’), “push Sw1 downwards on weekdays” (denoted by ‘ $S\downarrow d$ ’), “push Sw1 upwards on weekends” (denoted by ‘ $S\uparrow e$ ’), “push Sw1 downwards on

weekends” (denoted by ‘S↓e’). The output states of  $P_{1,4}$  are denoted by ‘1d’ for “light on on weekdays”, ‘0d’ for “light off on weekdays”, ‘1e’ for “light on on weekends”, and ‘0e’ for “light off on weekends”. Figure 4.10 depicts the causal structure of  $P_{1,4}$  (note that the  $\epsilon/\epsilon$ -transitions account for the automatic change in states resulting from the influence of real-time):

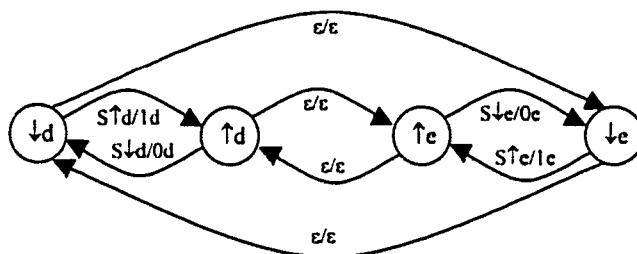


Figure 4.10 A graph depicting the causal structure of the physical system  $P_{1,4}$ .

Strangely enough,  $P_{1,4}$  implements automaton  $M_3$ :

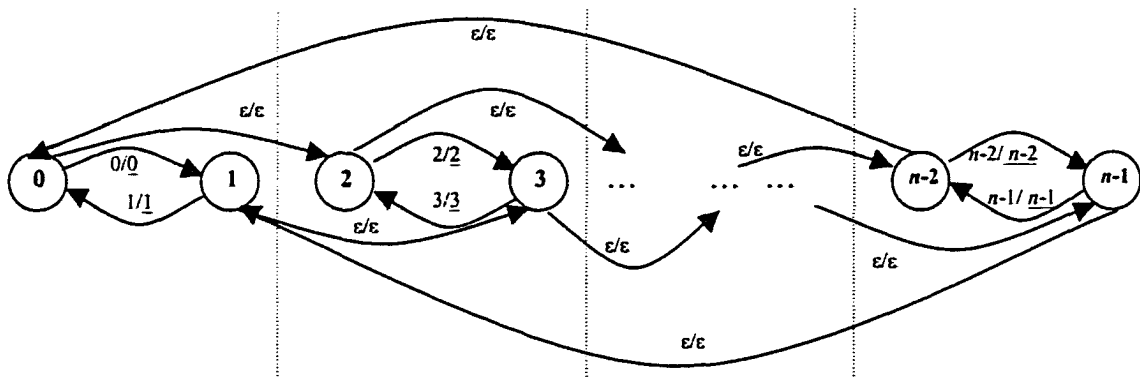
Inner states	$f(\downarrow d)=A$	$f(\uparrow d)=B$	$f(\uparrow e)=B$	$f(\downarrow e)=C$
Input states	$f(S\uparrow d)=a$	$f(S\uparrow e)=a$	$f(S\downarrow d)=b$	$f(S\downarrow e)=b$
Output states	$f(1d)=1$	$f(0d)=0$	$f(1e)=1$	$f(0e)=0$

This mapping can be generalized. In order for a switch system  $P_{l,n}$  to implement an arbitrary FSA (with  $m$  states,  $k$  different input and  $l$  different output symbols) one has to ensure that  $P_{l,n}$  has enough states and edges that can be mapped onto the graph of the FSA. The following theorem (which I christen the “Slicing Theorem” because it

generates additional states by “cutting off temporal slices” of existing ones) states the requirements and its proof exhibits the construction:

*Theorem 4.8:* [Slicing Theorem]  $P_{1,2k}$  implements any FSA with  $k$  transitions (for  $k > 1$ ).

*Proof:* Consider the physical switch system with  $n=2k$  internal states (for  $k > 1$ ) depicted as a graph with  $n$  nodes below. Each node is labeled with a (bold) natural number from 0 to  $n-1$ , and edges from node  $i$  to  $i+1$  are labeled with  $i/\underline{i}$  and edges from node  $i+1$  to  $i$  are labeled with  $\overline{i+1}/i+1$  (e.g., the edge from node 3 to node 4 is labeled with  $\overline{4}/4$ ). There are  $\epsilon/\epsilon$ -transitions from node  $i$  to  $i+2$  (for  $i < n-2$ ) as well as  $\epsilon/\epsilon$ -transitions from  $n-2$  to 0 and from  $n-1$  to 1. Each of the two original states of the switch system can be in  $k$  different states within an arbitrary given time interval  $I$  (e.g., if  $k=24$ , let  $I$  be one day and consequently consider switch states at each hour interval of the day).



**Figure 4.11** A graph depicting the physical system  $P_{1,2k}$  (dotted lines indicate states that lie within the same time interval).

Pick an arbitrary FSA  $M$  with  $k$  transitions, input alphabet  $\Sigma$  and output alphabet  $\Gamma$  (where  $c \in \Sigma$  and  $d \in \Gamma$  are symbols that do not occur in any transition). Notice that  $M$  can have at

most  $k+1$  nodes (but must have at least one node).<sup>54</sup> Without loss of generality, we can assume that the transitions are enumerated:  $\rightarrow_0, \rightarrow_1, \dots, \rightarrow_{k-1}$ . Define the following mapping  $f$  for  $P_{I,n}$ : for each transition  $(S,a) \rightarrow_i (T,b)$  (where  $S, T$  are states and  $a \in \Sigma - \{c\}$  and  $b \in \Gamma - \{d\}$ ) for  $i < k$ , let  $f(2i) = S, f(2i+1) = T, f(2i) = a, f(2i) = b, f(2i+1) = c, f(2i+1) = d$  (all transitions from  $2i+1$  to  $2i$  are neglected). Then it can be checked that under this mapping  $P_{I,n}$  implements  $M$ . The way this mapping works is simple: every transition of  $M$  is associated uniquely with a transition from node  $i$  to  $i+1$ .

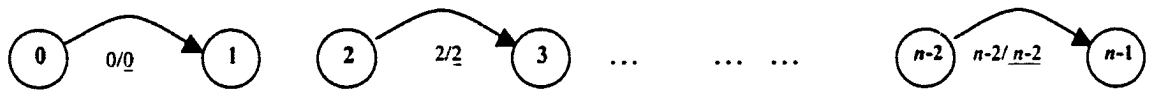


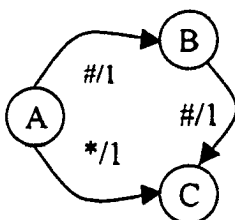
Figure 4.12 Transitions in  $P_{I,2k}$  that are mapped onto all  $k$  automata transitions.

Thus, for every transition  $(S,a) \rightarrow_i (T,b)$  in  $M$ , the following is true: if  $P_{I,n}$  is in internal state  $2i$ , it can only receive input  $2i$  and produce output  $\underline{2i}$  (given the time-dependence of all inputs and outputs,  $2i$  and  $\underline{2i}$  are the only states defined for that particular period of time, namely the time interval in which the switch system can be in states  $2i$  and  $2i+1$ —  
 (\*) other inputs/outputs that may be mapped onto  $a/b$  by  $f$  cannot be received/produced in that time period and need, therefore, not be taken into consideration). Given that  $P_{I,n}$  is in internal state  $2i$  and receives input  $2i$  (such that  $f(2i) = S$  and  $f(2i) = a$ ), this reliably causes it to enter internal state  $2i+1$  and produce output  $\underline{2i}$  such that  $f(2i+1) = T$  and  $f(2i) = b$ . q.e.d.

<sup>54</sup> There could be more nodes, if some of them are unreachable from the start state. In that case, one needs to divide the time interval further to obtain new states which can be mapped onto the unreachable ones, but this presents no difficulty.



There is an obvious weak spot in the above argument, marked by ‘(\*)’. One could argue that even though other input/output states are not applicable (because they have been just so defined), they should not be excluded *a priori*, but be taken into consideration in the proof. In other words, if there are other inputs, even though they are not available at the respective state of the system, that are mapped onto  $a$ , say, then *the definition of implementation* should take care of them, but this is not the case. Take, for example, the following automaton:



**Figure 4.13** An automaton which could be used to argue against the validity of the Slicing Theorem.

Define  $f$  according to the above construction for the three transitions:

$(S,a) \rightarrow_i (T,b)$	$f(..)=S$	$f(..)=T$	$f(..)=a$	$f(..)=b$
$(A,\#) \rightarrow_0 (B,1)$	$f(0)=A$	$f(1)=B$	$f(0)=\#$	$f(1)=1$
$(A,*) \rightarrow_1 (C,1)$	$f(2)=A$	$f(3)=C$	$f(2)=*$	$f(3)=1$
$(B,\#) \rightarrow_2 (C,1)$	$f(4)=B$	$f(5)=C$	$f(4)=\#$	$f(5)=1$

In checking whether the switch system so defined implements the automaton under  $f$ , one runs into the case where the switch system is in state 0 and receives the pendant to input

'#' , i.e., input 0 or input 4. Even though the latter is not possible if the system is in state 0, according to  $f$ , it maps onto input '#' in the automaton, and is, thus, a legitimate candidate for an input. It follows that the antecedent of the conditional (in the definition of implementation) is ("theoretically") true for internal state 0 and input 4, but that the consequent is false, because the system will not (reliably) transit into state 1 producing output 1. In fact, it will remain in state 0, because *it did not receive any input in the first place*. Hence, one could conclude that the theorem is not valid for cases such as the above.

Whether this kind of argument is valid or not, I leave to the reader to decide. Even if the objection is correct and the theorem has to be restricted to cases where no two transitions use the same symbol, a strange aftertaste remains: the simple switch system will still implement a restricted, yet infinite class of automata. Besides: every language accepted by an automaton with  $k$  transitions *without* such a restriction can be obtained as a homomorphic image of the language of an automaton with  $k$  transitions *with* the restriction.

Before I discuss the consequences of the Slicing Theorem for general theories of implementation, I would like to point out that this result can also be extended to CSAs, "which differ from FSAs only in that an internal state is specified not by a monadic label  $S$ , but by a *vector* [ $S^1, S^2, S^3, \dots$ ], where the  $i$ th component of the vector can take on a finite number of different values, or *substates*. [...] Input and output vectors are always finite, but the internal state vectors can be either finite or infinite. The finite case is simpler, and is all that is required for practical purposes" (Chalmers, 1994, p. 394). I will, hence, assume internal states to be finite. The implementation conditions are then:

“A physical system  $P$  implements an CSA  $M$  if there is a decomposition of internal states of  $P$  into components  $[s^1, s^2, \dots]$ , and a mapping  $f$  from the substates  $s^j$  into corresponding substates  $S^j$  of  $M$ , along with similar decompositions and mappings for inputs and outputs, such that for every state-transition rule  $([I^1, \dots, I^k], [S^1, S^2, \dots]) \rightarrow ([S'^1, S'^2, \dots], [O^1, \dots, O^l])$  of  $M$ : if  $P$  is in internal state  $[s^1, s^2, \dots]$  and receiving input  $[i^1, \dots, i^k]$ <sup>55</sup> which map to formal state and input  $[S^1, S^2, \dots]$  and  $[I^1, \dots, I^k]$  respectively, this reliably causes it to enter an internal state and produce an output that map to  $[S'^1, S'^2, \dots]$  and  $[O^1, \dots, O^l]$  respectively.” (Chalmers, 1994, p. 394)

Chalmers points out that “a natural requirement for such a decomposition is that each element correspond to a distinct physical region within the system [...] the same goes for the complex structure of inputs and outputs”. Again, he claims wrongly that “state-transition relations are isomorphic in the obvious way”. Furthermore, he is convinced that his CSA model prevents the notion of implementation from the threat of vacuity: “What counts is that a given system does not implement every computation [...] This is what is required for a substantial foundation for AI and cognitive science, and it is what the account I have given provides” (1994, p. 397). This can be contrasted with the following theorem:

*Theorem 4.9:* [Extended Slicing Theorem]  $P_{m,2k}$  implements any CSA with  $k$  transitions and  $m$  different substates of each state (for  $k > 1$  and  $m > 0$ ).

---

<sup>55</sup> I corrected the misprint in Chalmers' article by substituting 'k' for 'n'.

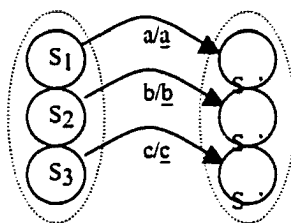
Proof: Since the general proof is rather lengthy, but not difficult in principle, I will sketch it for a CSA  $M$  with 8 states which are vectors of three components (substates) that can each assume one of the two values '0' and '1'.  $M$  will read inputs which are vectors of two components and deliver outputs that are vectors of one component, both substates take values from  $\{0,1\}$ .<sup>56</sup> Since there are 4 possible inputs and 8 possible inner states (output states do not have to be counted separately, because whenever input and inner state are the same, the output has to be same, too),  $M$  could have at most  $4 \cdot 8 \cdot 8 = 256$  transitions. I will show that  $P_{3,256}$ , a 3-switch system with three parallel switches/light bulbs connected to a battery and 256 internal states, implements  $M$ . First, consider the 3-switch system over some time interval  $Int$ , which is further divided into eight subintervals of equal length.<sup>57</sup> The first switch can be in states  $on_{Int1/2}$ ,  $off_{Int1/2}$ ,  $on_{Int2/2}$ ,  $off_{Int2/2}$ , the second in states  $on_{Int1/4}$ ,  $off_{Int1/4}$ ,  $on_{Int2/4}$ ,  $off_{Int2/4}$ ,  $on_{Int3/4}$ ,  $off_{Int3/4}$ ,  $on_{Int4/4}$ ,  $off_{Int4/4}$ , the third in states  $on_{Int1/8}$ ,  $off_{Int1/8}$ ,  $on_{Int2/8}$ , and so on ("Int1/2" designates the first half, "Int1/4" the first quarter, "Int2/4" the second quarter, etc. of  $Int$ ). Consider only transitions from "off" to "on" states. Then within  $Int$ , there are eight possible combinations of transitions for the three switches. Since there are also eight possible transitions between any two combinatorial states, one of the eight transitions can be mapped onto the one in  $M$ : if  $M$  transits between states  $[S_1, S_2, S_3]$  and  $[S_1', S_2', S_3']$  on input  $[I_1, I_2]$  outputting  $[O_1]$ , this corresponds to the 3-switch system transiting from state

---

<sup>56</sup> For example, the CSA could compute the "carry" in an addition: it would take the input as binary number and add it to the number represented by the current state, then transit into a state which represents the sum (modulo 8) and report if a carry over has occurred during the addition (by outputting 1, otherwise 0). To illustrate this, assume the automaton is in state  $[1,0,1]$  and receives input  $[1,1]$ . This makes it transit into state  $[0,0,0]$  and with output  $[1]$ . Had it been in state  $[1,0,0]$ , then the output would have been  $[0]$  and it would have transited to state  $[1,1,1]$ .

<sup>57</sup> For  $k$  substates, each of which can assume any of  $n$  different values, one would have to consider  $n^k$  different subintervals.

$[\text{off}_{Intx/2}, \text{off}_{Inty/4}, \text{off}_{Intz/8}]$  to state  $[\text{on}_{Intx/2}, \text{on}_{Inty/4}, \text{on}_{Intz/8}]$  on input  $[a_{Int}, bc_{Int}]$  outputting  $[\underline{abc}_{Int}]$  (where even values of the numerators  $X, Y, Z$  in the interval states correspond to the “0” value for automata substates, and those with odd values to the “1” value such that  $IntZ/8 \subseteq IntY/4 \subseteq IntX/2$ ). The transition  $[1,1],[1,0,1] \rightarrow [0,1,1],[0]$ , for example, would correspond to  $[a_{Int}bc_{Int}], [\text{off}_{Intx/2}, \text{off}_{Inty/4}, \text{off}_{Intz/8}] \rightarrow [\text{on}_{Intx/2}, \text{on}_{Inty/4}, \text{on}_{Intz/8}], [\underline{abc}_{Int}]$ . Note that inputs and outputs of the automaton have to correspond to combined inputs and outputs (indicated by concatenating the respective characters) in the 3-switch system.



**Figure 4.14** Transitions in the 3-switch system that are mapped onto all  $n$  automata transitions.

A physical state transition corresponding to a combinatorial state transition can then be defined as the transition taking place by pressing all 3 switches of the system during subinterval  $IntZ/8$  of  $Int$ . Physical states are defined correspondingly for this interval, inputs and outputs are combined states of switches and light bulbs (as described above). The rest of the construction proceeds as in the construction of the Slicing Theorem (e.g., the division of the a cyclic interval  $I$  into  $k$  parts in order to account for  $k$  transitions—in the case of  $M \geq 32$  such parts are needed, since there are 4 four different inputs for each of the 8 possible combinatorial states; e.g., if the cyclic interval is one day, then the automaton could spend three quarters of an hour in one of the  $Int$  states). The main

difference between the above construction and the previous one for the Slicing Theorem is that substates are mapped onto distinct spatial regions (i.e., the switches), and that the complex state transitions between substates is preserved. It follows, then, that  $P_{3,256}$  implements  $M$  in the sense of Chalmers' definition. As a consequence, generalizing the above construction, every CSA with at most  $m$  different substates of each combinatorial state is implemented by an  $m$ -switch system. q.e.d.

This kind of result that one physical system can implement “too many” computations is exactly what Chalmers tried to avoid when he proposed his definition against the background of Putnam's Realization Theorem (which states that every ordinary open system implements every finite state automaton without input and output). In a sense, the Slicing Theorems strengthens Putnam's program against charges such as “wrong notion of causality”, “input/output missing”, “wrong level of description”, “unnatural physical types”, etc. All of these, except perhaps for the last one, are dismissed by the Slicing Theorems. This adds evidence to the claim that Putnam's construction, as he notes at various places (e.g., Putnam, 1988, pp. 95), essentially points out the lack of appropriate state and type formation rules. All other points of critique are secondary. Thus, I deny Chalmers' belief that physical states are not the main problem at hand: “there does not seem to be an objective distinction between ‘natural’ and ‘unnatural’ states that can do the relevant work. [...] I will not pursue this line, as I think the problems lies elsewhere” (Chalmers, 1996, p. 312). Although I share his belief regarding the distinction between “natural and unnatural states”, I do not think that the problem lies elsewhere. In fact, I affirm that *states are the problem*, as shown by the Slicing Theorems.

Interestingly, the construction exploited in the Slicing Theorems differs in at least five crucial aspects from Putnam's construction:

1) While Putnam's construction shows how to implement a particular "run" of a computation, i.e., a particular sequence of state transitions, the above construction models the complete state-transitional structure of the automaton. That is why it only needs a finite (i.e., bounded) number of states to perform arbitrarily long computations (i.e., all possible computational sequences), whereas Putnam's construction requires an unbounded number of states (depending on the length of the respective computational sequence). It exploits the fact that at some level of description physical configurations can be viewed as *recurrent* (whereas Putnam used the "Principle of Non-Cyclic Behavior" to obtain *new* physical states).<sup>58</sup>

2) Tokens of state types such as "switch up on Mondays" can be easily individuated (we can check if it is—currently or at some other time—Monday and we can check whether the switch is in position "up" or "down"). These input states have the predictive capacity that Putnam's construction was criticized for; it is known ahead of time, what tokens of these types will look like and how they can be produced (as necessary requirement if one wants to control the input to a system!).

3) State transitions are reliable. Pressing switches is certainly as reliable an action as any reliable one can imagine (unless the switch is defective, etc., which can be accounted for by adding conditions of normal operations to the definition of implementation as in section 3 of chapter 2). The same holds of the light bulbs being lit or not lit under the

---

<sup>58</sup>Note that plausibility of the principle of non-cyclical behavior was one of the main points of attack of Putnam's argument, see, for example, Chrisley (1994). Without this principle, Putnam's construction cannot work.

respective circumstances: given a setup corresponding to the wire diagram, the respective light bulbs will be lit reliably if the switches are in a position such that current can flow (and neither the battery, nor the wires, nor the light bulbs are defective). Transitions from one time interval into the next are obviously reliable as well, as they happen without further ado (and rely on the physical laws regarding the permanence of objects which are not subject to internal decay processes and/or external influences).

4) State transitions support counterfactuals. To see this, note that input occurs only “within time slices”, i.e., whenever a switch is pressed down on Monday, say, the system will reliably change state into “switch down on Monday”. If it is pushed up again, the system will return to state “switch up on Monday”. It can never be case that the system receives input on Monday and ends up in a state on Tuesday. Thus, any counterfactual of the form “had the system been in state  $p$ , on input  $in$  it would have transited into state  $q$  producing output  $out$ ” is true for any time slice, and since input driven state changes cannot occur across time slices, it is vacuously true for those.

5) Because of the above and the laws of physics (i.e., circuit theory), the *relevant* state transitions (i.e., those that are input driven) are causal, not only according to a physical notion of causation, but to the stronger, counterfactual supporting notion that people like Chalmers and Chrisley, for example, require. It could be objected that temporal successions of one and the same physical state, i.e., state  $s$  at time  $t_n$  and state  $s$  at time  $t_{n+1}$ , cannot be said to be causal transitions. Granted! But this is true only of “irrelevant” state transitions (i.e.,  $\epsilon, \epsilon$ -transitions). In automata theory,  $\epsilon, \epsilon$ -transitions are supposed to model transitions in a system that happen *without further ado*: no input is necessary, no



output is produced, the automaton transits without input from one inner state to another—that is why these transitions are called  $\epsilon, \epsilon$ -transitions in the first place. The same is true of transitions between two time slices: no input is necessary, no output is produced, the system transits from one inner state into another without any external influence. In that sense these transitions are *not caused by anything*. That is why I limited the claim that state transitions are causal to *relevant transitions*, i.e., the ones that *are causal*. Only relevant transitions are mapped onto automata transitions, thus all transitions that are “mirrored” in the automaton are causal.

It seems that the only objection left to the above construction is the nature of the involved physical states, since the main charges (see Chalmers, 1996, or Chrisley, 1996) against Putnam’s Theorem that his notion of implementation is not based on reliable, counterfactual supporting, causal state transitions (i.e., that his notion of causality does not support counterfactuals) does not *mutatis mutandis* transfer to the Slicing Theorems. One conclusion to be drawn from the Slicing Theorems would be to disallow *temporal individuations* of physical states. That way one could savor Chalmers’ notion of implementation and explain what went awry in the above construction. This, however, seems to me too strong a restriction as there might be cases where temporality is crucial in individuating physical states: consider a shared memory between two processors such that the first processor accesses the memory during even clock cycles and the other during odd clock cycles. To understand what is going on in such a memory (e.g., when the value of a memory location has changed “between two successive states without

further ado” from the perspective of one processor) one would probably introduce notions like “even state” and “odd” state to refer to states at certain clock cycles.

There is a better reply to the objection that the unwanted results of the Slicing Theorems result from temporally individuated states. Instead of individuating these states temporally, one could slightly modify the switch system by adding “a clock”, which reliably goes through a fixed cycle of physical states in a given amount of time ( $12 \cdot 60 \cdot 60$  states per 12 hours, say). Then one can form the same “slices” that were used in the switch system to implement arbitrarily complex computations, except that the *temporal* individuation (the temporal slices) is now replaced by “spatial individuation” (of the clock hands, for example). By forming combined states such as “the switch at 5h34’33’ ” (where the time expression is used to fix a spatial position on the clock), the clock-switch system can implement very complex CSAs according to Chalmers’ definition of implementation using the construction of the Slicing (i.e., to be exact a one-switch system with one clock will implement any computation with up to  $12 \cdot 60 \cdot 60$  state transitions, and by adding additional clocks and/or increasing the number of distinct clock states, this number can be increased significantly). Yet, it is intuitively very clear that the system does not do any computational work.

#### **4.9 General Problems with the State-to-State Correspondence View of Implementation**

The analysis of Chalmers’ definitions of implementation for FSAs and CSAs has shown that they do not achieve what they promise, namely to view computation as an abstract way of specifying the causal structure of a physical system. They are insufficient for two

reasons: first, according to these definitions physical systems do not implement only isomorphic computations for a given set of physical states and their causal relations, but rather many different computations. Hence, computations cannot specify *the* causal structure of a system. In that case one would at least expect that the definitions view physical systems as implementing all computations that are *bisimilar* to their isomorphic computations. However, Chalmers' definitions do not permit this either, since only bisimilar computations with *fewer* states than the isomorphic ones are viewed as being implemented by the system. This deficiency was accounted for by distinguishing the notion "realization of a FSA" from "implementation of a FSA": the former is viewed as a special case of the latter, in which the automaton "mirrors" the causal structure of the implementing physical system by virtue of its being isomorphic to it, whereas the automata being implemented according to the latter are only bisimilar to the causal structure of the physical system.

However, this only repairs part of the deficiency. A much more severe flaw prevails: even simple physical systems can be viewed as implementing very complex computations (normally exhibited by very *complex* physical systems) under this definition, contrary to what one would intuitively expect! To prove this point, I have devised a construction, the one used to prove the two Slicing Theorems, which replaces spatial complexity of physical states (in "complex" physical systems) by temporal complexity of physical states (in "simple" physical systems such as the switch systems). That way the lack of spatial structure is compensated for by temporal structure and spatially structured causal patterns are turned into temporally extended causal patterns.

Interestingly enough, one could even use switch systems with the above-defined “temporally divided” states for real computations: one only needs to add a clock to turn non-effective temporal properties such as “the state of  $x$  on Monday” into effective spatial properties such as “the state of the all hands” on a certain clock...<sup>59</sup>

The Slicing Theorems pose a serious threat not only to Chalmers’ definition of implementation, but to any state-to-state correspondence view of implementation (as well as any semantic view of implementation, for that matter: just substitute “part of the system” for “state of the system”), because any such view crucially depends on a notion of “physical state” (“physical part”, respectively)—I will refer to both views together as “SV/CV”. SV/CVs need this notion to set up a correspondence between the physical and the abstract, yet the question of what counts as a *legitimate physical state* is not answered by any of them (as far as I know). SV/CVs can only provide an answer to the implementation problem for systems for which a set of physical states is given. If physical states are *not* given, SV/CVs run into insurmountable difficulties: following the construction of the Slicing Theorems, physical states *supporting counterfactuals* can be defined, for which the system implements almost any computation. In short, the reliance of SV/CVs on a notion of physical state is their essential weak spot!

One could object to this reasoning by pointing out that it is not part of a SV/CV to deliver a definition of “physical state”. Granted! But how would one decide what

---

<sup>59</sup>Jack Copeland, in personal communication, admitted that such systems would compute according to his definition of computation too (see chapter 3). Interestingly, Copeland believes that these systems are *rightfully* viewed as computing, while it seems to me that they really are not doing any *computational work*. Surely, they can *be used* for computing, if, for example, a human (or another device, for that matter) presses the switches in the right order at the right times, but that does not make them *compute by themselves*. The situation is similar to humans using an abacus, which *qua* physical devices helps storing computational states—i.e., it functions as some sort of memory—yet, I would claim, the computation takes only place in the system “human *cum* abacus”, not in the abacus alone!).

computation a physical system implements whose description does not provide a notion of physical state? Is one then to assume that this system does not implement anything?

Physical states of a physical system are normally defined by the theory in which that system is described. As it happens with classical fields, there might be too many states that could potentially correspond to some abstract, in this case *computational*, state. In order to exclude certain unwanted candidates, one has to define an individuation criterion according to which physical states are singled out. This criterion, however, is not defined within the physical theory that is used to describe the object, but rather at a higher level of description. In the worst case, this will be exactly the computational level, namely in the case that none of the potential “lower level” theories can define a property in their respective languages such that the set of states conforming to that property corresponds in a “natural” way to the computational state. The potential circularity is apparent: what it is to be a certain computational state, is to be a set of physical states which are grouped together because they are taken to correspond to that very computational state.

Every SV/CV must, therefore, avoid being 1) vacuously broad (because physical state type formations are too liberal), and 2) circular (because individuation criteria for physical states are not provided at level lower than the computational).

In the case of physical fields, one is left with a very pessimistic prospect: there are more than countably many different possible physical states according to the state space of fields (for every interval of real-time). Which of those correspond to a physically possible object, and which correspond to a given object in a “natural way”? Since there are *even more possible mappings* from physical states onto abstract states, it seems totally implausible, if not impossible, to specify finite criteria that single out the right mappings.

The only way we could find such a mapping is either by pure chance or by using higher level properties that constrain possible objects significantly and hence the plethora of mappings. If we are lucky, then the number of mappings will be so constrained by these properties that we can actually write down the definition of a (correspondence/interpretation-)function. But again, this “will work” only by using properties defined at levels of description higher than physical fields, yet lower than the computational level of description (which must not be used in defining a mapping from physical states to computational states, if the task is to find out what kind of computation a given physical system implements). One advantage of higher level theories is that they supply “higher level”-objects that can be individuated according to criteria supplied by these theories, and properties of these objects, in turn, could be used to define states.<sup>60</sup> State-to-state correspondences (or parts as label bearers, respectively) would then have to be defined separately for individual theories.

Involving higher level theories, however, does not solve the problem of forming physical state types if the theory does not provide such a concept. Take a pyramidal cell, for example, and its physical description in the language of biochemistry, which does not provide a notion of physical state type that could be set in direct correspondence with states of connectionist units. How would one go about defining physical state types such that the behavior of the cell corresponds to the computation of its “connectionist counterpart” and, at the same time, these type formation rules exclude type formations that would give rise to “unwanted” computations? This does not seem clear.

---

<sup>60</sup> At the level of fields, the question what exactly counts as an individual is a highly debated issue. It is not all that clear that Quantum Field Theory, for example, even contains individuation criteria for particles.

So the main difficulties of the SV/CV sneak in the backdoor again, if the question “How are abstract computations tied to the concrete?” is asked. Even the rephrasing “What computation is implemented by a concrete system?” is not sufficient, since it, too, assumes a notion of computation.

Two possible, non-exclusive conclusions are implied: 1) computation is not the right kind of explanatory device for causal organizations of physical systems (and as a consequence for theories of mind), and/or 2) SV/CVs are not the right kind of approach to a general theory of implementation (as they will fail as soon as a physical theory does not provide a well-defined notion of physical state).

I am inclined to believe the latter, whereas I am somewhat undecided on the former. SV/CVs are certainly applicable if physical states are given, which in real-world hardware design, for example, is obviously the case. Using the modification to Chalmers’ definition (definitions 4.4 and 4.5), one can easily specify the class of CSAs that is implemented by various digital circuits, for example. One could even extend this definition and define a physical system to *implement a computation* if it is bisimilar to the FSA realized by the system. This notion of implementation would then encompass any computational formalism that provides notions of input, inner, and output state and permits one to relate them to physical systems *via* the isomorphic FSA.

Still, the success of SV/CVs with systems that have been designed so as to allow for an easy state-to-state correspondence between physical states (which are given) and more abstract states should not distract from their failure in the general case. Because computations have to be linked to concrete systems (which are described at a certain

level) in order to *be computations*, the implementation-relation must hold between computations and *levels of descriptions* of systems.

We are left with various questions unanswered: which (lower) level is the *right* one? Which level supplies the right kinds of states to be linked to the computational ones? Is there a systematic way to 1) find the right level and 2) find the right states/state types at this level? A theory of implementation should be able to answer all these questions in a systematic way for all possible levels of description. Any state-to-state correspondence or semantic view, however, is naturally limited to a level of description and a notion of state/part at that level (if it exists at all, otherwise the particular choice has to be justified with all its consequences...), and can, therefore, not provide any criterion for particular choices of levels. Furthermore, a theory of implementation should provide *necessary and sufficient* criteria to determine whether a class of computations is implemented by a class of physical systems (described at a given level), otherwise the term “implementation” is not appropriate. As long as these problems are not solved, constructions like the ones in the Slicing Theorems will present a potential threat to any SV/CV, precisely because there exist levels of description at which—paraphrasing a famous dictum by Feyerabend—*anything computes*.



## Chapter 5:

### When Physical Systems Realize Functions...

#### 5.1 Taking the Physical Seriously

The main reason for all the difficulties with a satisfactory account of implementation is, in my view, that computation is normally defined abstractly at a “level of symbol manipulation”, rather than in terms of an abstraction over the physical properties determining the functionality of a physical device. Although for logical purposes this approach is necessary, it is certainly not the one taken by computer practitioners, who need to define programming languages for the hardware they construct (in order to make it accessible and, hence, usable for other people). By abstracting over hardware specifics such as particular brands of parts, speeds of gates, etc. they are able to come up with an abstract description of what it means to *compute on their kind of machine*. This way implementation and computation are defined together and the question of how computations are tied to the physical in general does not arise. It is this kind of practical wisdom that theories of implementation need to capture.

Even if one willingly granted Turing machines, for example, such a link (assuming that there are physical systems that correspond to them, ignoring all the difficulties mentioned earlier in this work), they would still be mere models of what can be done *mechanically* (by a human, a robot, etc.). Gandy (1980), for example, shows that four principles underwrite the concept of “mechanical doability” and that a violation of each

of these principles gives rise to “Super-Turing” computation. It follows that any system whose behavior can be completely described at the level of configurations, changes of configurations, etc. and which conforms to these four principles, will be at best Turing-computable. It does not follow, of course, that this level of description is the appropriate one for what humans can do (if they use scratch paper, but are not bound to using rules, say).<sup>61</sup>

The relevance for cognitive science (assuming CCM) is immediate: suppose brains are best described at a lower-than-mechanical level of description  $L$  (e.g., a biological level), and the mechanical level of description is not sufficient for  $L$ . If some of the phenomena not describable in terms of “mechanics” are crucial to a theory of mind, then either minds are not computational (if computational is meant to be “mechanical”) or a different notion of computation is required (e.g., if one wants to describe biological systems such as cells, autopoietic systems, neural networks, etc. as “computational”).

Considerations of this sort have already inspired many cognitive scientists to shift their explanatory framework from computational to dynamical, because they believe that the “computational level of description” is not essential to understanding cognition (see, e.g., van Gelder (1998)). Although one has to be careful with statements like this, because their truth depends on what “computational” means, I would agree that Turing machines are not well-suited to describe the behavior of various physical systems at lower levels such as chemical levels or even biological levels. And since the class of functions computable by Turing machines is the same as the class of recursive functions,

---

<sup>61</sup> Gödel (1958), for example, thought that human intuition, especially mathematical intuition, could exceed Turing computability, and his hunch is shared by influential logicians and scientists such as Feferman and Penrose.

it follows that these functions, too, *might* not be adequate to describe the input-output behavior of systems at lower levels of description. It even seems possible that the behavior of some physical systems could only be adequately described using recursively enumerable functions (certain quantum processes, say).

There is a quite a bit of literature on this issue of whether there are “non-computational” processes in nature. Various physicists as well as logicians have speculated what such processes would look like (e.g., see Copeland, 1998a, for various references). If such processes exist, it will be the job of the engineers to find ways and methods to utilize them (no doubt, a computer performing some non-Turing-computable function would, besides being revolutionary, be extremely desirable, and not only from a scientific point of view). Yet, this is not a conceptual problem, but an empirical issue. Whether such processes exist or not does not depend on a logical system, but on the physical properties of the “real world”. Thus, although it is possible to investigate logical properties of machines that can compute more functions than Turing machines—take, for example, the so-called “oracle machines” envisioned by Turing himself (Turing, 1939)—it is almost a truism that it will remain a physical problem whether these machines are physically possible. If it were possible to utilize their behavioral complexity for “computational” purposes, (computer) scientists would willingly extend their notion of “computation” to the class of functions “implemented” by those systems.<sup>62</sup>

This also opens up a new perspective on the relation between mind and computation: minds might just be oracle machines! Copeland, in pointing out Searle and Penrose’s misreadings of the Church-Turing Thesis, writes:

“O-machines are digital computing machines. They generate digital output from digital input by means of a step-by-step procedure consisting of repeated applications of a small, fixed number of primitive operations, the procedure unfolding under the control of a finite program of instructions which is stored internally in the form of data on the machine's tape. Thus even if 1) [the claim that the human brain is equivalent to some Turing machine] is false, the theory that the brain is a computing machine might nevertheless be true.” (Copeland, 1998a, p. 1, comments in brackets are mine)

All of these considerations together have led me to believe that a different theoretical framework is required in order to capture not only accepted, but also potential notions of computation. For example, this notion should (in principle) not exclude systems that could outperform a Turing machine. Furthermore, it should allow one to define a corresponding notion of implementation (for each notion of computation) which explains how to link the abstract to the concrete without opening doors to Putnam-like constructions. One possible strategy to develop such a framework—the one I will take up in this chapter—is to start in the concrete, in the physical, and not in the abstract: take a physical theory  $P$  and consider its (simple and complex) objects together with their properties.  $P$  will supply laws that describe the behavior  $F$  of a given arrangement of these objects—called physical system  $S$ —under certain environmental conditions over time. Depending on  $P$ , objects, behaviors, and environmental conditions will be very different. However, every object in  $P$  will be subject to a certain change in some physical dimension (its “output”) if exposed to certain environmental conditions (its “input”). The next sections will show how input, output, and behavior of a physical system as described

---

<sup>62</sup> See also the last chapter of this dissertation.

by a physical theory can be used to distill a (matter-independent) mathematical mapping  $f$ —the function realized by  $S$ —between inputs and outputs of  $S$  by abstracting over every physical dimension. This abstraction will not be arbitrary, but determined by  $P$ , resulting in a *computational description*  $C$  of  $F$ . At the same time, the trace to the “real stuff” is not forgotten,  $C$  can always be seen as an abstraction of  $F$ . Thus, a natural link between  $S$  and  $C$  is supplied by the process of developing  $C$  out of  $F$ . This process can be viewed as providing the *implementation* criterion for  $C$  on  $S$ .

The notion underlying the process that eventually will lead to a computational description of certain physical systems together with an objective implementation criterion is that of “realization of a function”. This notion comes in a variety of different forms at different levels of abstraction, one of which will be identified as the computational level. Depending on the level of abstraction at which physical systems are described, they will realize different functions. Since the choice of the level of description always depends on pragmatic considerations, there is no “true and genuine” function realized by the system. Only once a level is chosen, then there is a unique function that describes the behavior of the system at that level. It will be the task of the following sections to define these levels guided only by practical constraints and pragmatic choices. Nothing ontological will be claimed about the process of abstraction, except that this progression explicates wisdom slumbering in the practice of computing. Yet, it will allow us to see computation not as an abstract phenomenon (“mere syntax” as Searle would say, and thus observer-relative), but merely as a very abstract description of real-world processes. In other words, once one can agree on the behavior  $F$  of a physical system  $S$  as well as on the level of abstraction at which descriptions can be viewed as

computational, the computation  $C$  implemented by  $S$  will be logical consequence of the framework developed here and not observer- or interest-relative. So, the burden of interest relativity will be shifted from implementation to the particular choice of the physical theory that is used to describe  $S$ —the way it should be!

## 5.2 Setting the Stage: Electromagnetic Fields and Circuit Theory

Let us, then, start with a physical theory at a rather low level of description, *the level of electromagnetic fields*, and see how we can develop a notion of what it means for a physical system (described at this level) to realize a function. The theory of electromagnetism, as axiomatized by the four Maxwell equations, describes the behavior of (moving) charges: how (moving) charges give rise to two kinds of fields, the electrostatic and the magnetostatic field, and how these fields interact over time, resulting in electromagnetic fields.<sup>63</sup> In the simplest case, fields are studied in a vacuum, but Maxwell's equations can be modified account for fields in material media as well as across the change across material boundaries. In particular, the theory explains the interaction of potentials and currents in spatial regions “filled” with various materials over time (e.g., a cylindrical region of space filled with copper). Dividing materials into two rough categories, conductors and insulators, one can launch an investigation into the nature of different spatial combinations of materials giving rise to different electric properties. Without providing a detailed mathematical derivation, one can imagine how a categorization of combinations of different materials into types with the same electric

---

<sup>63</sup> Here, I assume classical fields for simplicity sake. I am not be concerned with the integration of electromagnetic fields into quantum field theory (e.g., how particles arise out of fields, etc.).

properties could be attempted (by abstracting over particular material properties such as conductance or spatial arrangement such as volume). A cylindrical region filled with a given material in a vacuum, for example, will exhibit certain law-like properties with respect to the difference in potential between its two ends, if current flows through it. Furthermore, it will be possible to extract laws from the study of different such arrangements (e.g., Ohm's law, which describes the relation between potential, current, and conductance of such "material regions").

This abstraction process leads, in the end, to the development of *circuit theory*, a theory which arises from the theory of electromagnetic fields by abstracting over material properties of spatial regions as well as the regions themselves.<sup>64</sup> It considers "closed-loop" arrangements (i.e., *circuits*) of two sorts of "higher level" objects: active and passive components. Active components are energy sources (e.g., batteries), passive ones are energy consumers (e.g., resistors, capacitors, or inductors).<sup>65</sup> The "electric" properties of components and circuits are expressed in terms of "potential", "current", "conductance", "inductance", "resistance", "capacity", "voltage", etc.<sup>66</sup>

The reason for choosing circuit theory as a venture point in the current enterprise is twofold: on the one hand, circuit theory is motivated by computational practice (although it also has some relevance for the neurosciences—see the end of this section), since

---

<sup>64</sup>Note that I do not claim that this happens necessarily so or that it happened historically that way.

<sup>65</sup>A charged capacitor, of course, functions as an energy source, too.

<sup>66</sup>Notice that what is an *electric component* (e.g., resistors, capacitors, transistors, vacuum tubes, copper wires, etc.) is assumed to be already determined and given. In particular, I will be talking about those electronic objects that I can buy in a store and weld into a board with copper connections on one or both sides to build things like radios, amplifiers, pocket calculators, and the like. Of course, a star has also a resistance (and, to some degree, resembles a resistor) and so does a molecule, but neither of them has the right size to be of use for a device that I can build, and this is all that matters if I want to listen to my favorite radio show or quickly calculate the accumulated interest in my bank account.

computers by and large are built out of electric components. On the other, it provides “basic objects”, which can be individuated according to their properties and combined to form circuits, as opposed to electromagnetism, where space points together with their charge are the “basic objects” of the theory. Although one can *reduce* “circuit talk” to “field talk”, a field-theoretic approach would distract from the current goal because of its mathematically involved nature. Additional information about certain kinds of materials, their atomic make-up, as well as facts from crystallography, atomic physics, chemistry, etc. would be needed to describe circuits completely at a level of fields. Circuit theory allows one to abstract over these “physical peculiarities” and assume objects such as resistors and capacitors without having to know their physical realization. Yet, one is guaranteed that these types of objects (within practical limits, of course) are readily available, i.e., can be built, since circuit theory was developed under the pressure of practical, engineering tasks. It, thus, combines the “physical rigor” of classical fields with the “engineering view” of idealized circuits, which are both necessary to describe actual and possible objects that are metaphysically tenable and physically plausible.<sup>67</sup> For the rest of this chapter, I will assume circuit theory for all examples, yet allow “*P*” to range over any physical theory in all of the formal definitions.<sup>68</sup>

---

<sup>67</sup> Besides the fact that *all kinds of objects* can be described by the theory of “electronic circuits”, one could even doubt that it is clear what “object” means in this case. In other words, one could question the very notion of “object” and argue that circuit theory does not provide sufficient criteria for the individuation of its basic objects. Then the above level cannot be taken for granted, and one has to dig deeper into the metaphysical stuff to find substance and defining properties of objects (e.g., see Smith, 1996). From a pragmatic point of view, however, I believe that this step is not necessary: there are ways to find out if something is a (standard) “resistor”, say, and even if a “thing” is not clearly a resistor, if it has the appropriate resistance (and that can be measured) and the appropriate shape, form, size, etc. it could be used as one.

<sup>68</sup> The term “physical theory” is meant to comprise every theory that describes natural phenomena at a certain level of description in terms of “physical” laws such as biology, chemistry, etc.



Any (physical) theory used to describe real-world phenomena is built upon the mathematical framework that has been developed to describe *change*: the theory of differential equations (this is a matter of fact, not a necessity). Formally, this means that the theory consists of all mathematical (and logical) axioms needed for the theory of differential equations together with all necessary physical *eigenaxioms* (i.e., the axioms of the physical theory). It will contain additional predicates for different physical magnitudes (such as mass, energy, and time, for example) as well as other factors depending on the theory and the purpose it is used for. A formal version of circuit theory, in particular, could contain predicates like “is\_a\_resistor(x)”, “is\_a\_voltage(x)”, etc. and relational primitives like “has\_resistance(x,y)”, “is\_connected\_to(x,y)”, etc. Using these predicates, one can formulate general laws such as

$$\forall x(\text{is\_a\_resistor}(x) \rightarrow \exists ! y(\text{is\_a\_resistance}(y) \wedge \text{has\_resistance}(x,y))).$$

Given this axiom, a function symbol can be introduced for the resistance of a component, and this function in turn can be used to define Ohm’s law.

### 5.3 What is Means for a Physical System to Realize a Function

Physical theories, as already noted, describe the behavior of physical objects (and possible arrangements of them) under certain environmental conditions over time. What a physical object is depends on the theory under consideration. Each theory will supply a notion of primitive object (even if this can only be determined by looking at the domain the quantifiers of the theory range over). I will use the term “physical system” to emphasize this theory-dependence. Circuit theory, for example, can describe the *behavior* of the system “resistor” when a voltage is applied. This can be also viewed as

describing the “input-output function” of the component: take a  $2 \Omega$  resistor (the “component”) to which a voltage of 200 mV (the “input”) is applied. The current flow then—according to Ohm’s law  $V=R \cdot I$ —is 1 mA (the “output”).<sup>69</sup> So, the resistor has the property of “embodying” the function  $F(x)=x/2$  from voltages to currents.

What does it mean, therefore, to “embody” a certain function, to “have” a certain function, to “function” in a certain way? It means to *obey the laws of physics* that are described by that function. So, each concrete, individual resistor of  $2 \Omega$  will under an applied voltage of 200 mV yield a current of 100 mA—this is what the laws of physics predict (and if they are correct, this is what will happen modulo some practical problems such as “purity” of the material, exactness of the applied voltage, etc. which I will ignore for the moment). There are obviously critical notions involved in the above statement such as “predictions”, “law obeying”, maybe even “counterfactual”, but I will not be able to go into details here.<sup>70</sup> However, exactly *how* the resistor “achieves” this mapping is a different question and can be answered by looking at the chemical structure, the arrangement of molecules, facts about electrons, etc. At this point we are only interested in the “*what*”-question and the answer to it does not require theories at “lower levels” of description.<sup>71</sup>

There is a sense in which the above resistor realizes the same function as a copper wire that is split into two parallel wires at some point: according to Ohm/Kirkhoff’s laws

---

<sup>69</sup> Notice that Ohm’s law is a special case of a differential equation where  $\Delta t=0$ , i.e., where time is left out.

<sup>70</sup> Actually, one reason to pursue this particular line of construction was exactly to avoid arguments about notions such as “counterfactual”, “(natural) law”, “law-likeness”, “obeying a law”, etc. Consent on what it means to “obey a physical law” or “to be a law of nature” is simply presumed.

<sup>71</sup> It seems to be a characteristic property of levels of description that if at a given level  $n$  a “what”-question can be answered, the corresponding “how”-question has to be answered at a lower level (if it can be answered at all).

the current will split in half (assuming they are joined together or fed into equal loads). So if the current at one end was 100 mA, then it will be 50 mA at each of the two other ends. The function “realized” by this wire is then  $F'(x)=x/2$  from currents to currents, as opposed to the function  $F(x)=x/2$  from voltages to currents. So, although the physical dimensions that are used for input and output (i.e., the domain and the range of the function that describes an aspect of the system under consideration) are different, the abstract mapping between those objects is the same. If, therefore, one drops the physical “qualities” (dimensions such as voltage or current) in the description of the “function” of the components and just considers the “quantities” (magnitudes of the physical units), then one can describe the “input-output”-function of both components as  $f(x)=x/2$  from numbers to numbers (*Reals* to *Reals*, say, given that voltages and currents are usually defined as *Real* values in circuit theory). In short: dimensions are dropped, units are retained.

This step of abstraction is critical to the whole enterprise and I will, therefore, develop the argument in greater detail. Eliminating physical particulars of inputs and outputs will allow us to describe what resistor and wire have in common with respect to their mathematical (functional) description, namely that “the resistor and the wire both *realize* the function  $f(x)$ ” (in different physical ways, of course). The resistor realizes  $f$  *in the sense* that for every voltage  $x$ , if  $x$  is applied to the resistor, a current of  $x/2$  will result by the laws of physics; the wire, in that for every current  $x$ , if  $x$  is applied to one end of the wire, a current of  $x/2$  will result by the laws of physics on each one of the other ends. What is, therefore, different is the domain of  $f$  (since in one case it is the set of all currents, in the other the set of all voltages), what is the same is the syntactic structure of

(the definition of) the function. There are two ways to express the difference between voltages and currents, 1) syntactically by using a two-sorted logic:  $f(v)=v/2$  and  $f(a)=a/2$  (where “ $v$ ” is a variable ranging over volts and “ $a$ ” a variable ranging over amperes), and 2) semantically by using a model-theoretic interpretation  $\forall V(x)f(x)=x/2$  and  $\forall A(x)f(x)=x/2$  (where  $V$  denotes the set of voltages and  $A$  the set of other amperes). In the former case, the algebraic field axioms (within the formal theory) coincide for both sorts.<sup>72</sup> Hence, for  $r=1$  in Ohm’s law  $\forall v\forall r\exists!a v/r=a$  (where “ $/$ ” defines the “division of voltages by resistances”), one obtains  $\forall v\exists!a v=a$ , which implies the extensional identity of  $V$  and  $A$ .<sup>73</sup> It follows that voltages and currents are interchangeable (within the formal theory). In the latter case,  $V$  and  $A$  can be shown to be extensionally identical using the same idea:  $y=1$  in  $\forall V(x)\forall R(y)\exists!A(z) x/y=z$  implies  $\forall V(x)\exists!A(z) x=z$  and, hence, the extensional identity of both predicates. Therefore, in either case, one arrives at the mapping  $f(x)=x/2$  from *Volts* to *Volts*, say.

Ideally, however, one would like to arrive at something like the mapping  $f(x)=x/2$  from *Reals* to *Reals*, since there are formal (physical) theories in which no “law” exists that connects different physical quantities (e.g., Newton’s second law  $F=ma$  can express the same relationship  $f(x)=x/2$  from “forces” to “accelerations”, but if it is added to circuit theory together with the fact that components have a mass, there will be no formal axiom that relates forces and voltages, or forces and currents for that matter).<sup>74</sup> And even in

---

<sup>72</sup> With “field axioms” I mean the set of axioms that describe properties of voltages and amperes with respect to “(voltage/current) addition” and “(voltage/current) multiplication”. These axioms are necessary in order to define the “mathematical” properties of volts and amperes...

<sup>73</sup> Note that I left out quotation marks around all formulas in favor of readability.

<sup>74</sup> A radical position would, of course, maintain that if there is no such law then there is no reason to assume that both qualities can be identified with respect to their quantities. In that case, “realizing the same function” would be a notion

those cases one would like to view the “quantitative” sides of forces and volts as being the same. One strategy is to argue that physics uses a mathematical language to describe concrete objects and that both volts and forces are modeled by *Reals* in that language.<sup>75</sup> This opens the discussion about whether *Reals* are the “right models” for physical quantities, etc. Another argument comes from model theory stressing the fact that two sets are *identical up to isomorphism* (with respect to the field operations “+” and “\*”), if they satisfy all field axioms. It follows that a model where the set of Volts is the set of *Reals* is as good a model as one where the set of Volts is the set of *Rationals*, since the extensions of the predicates “*V*” and “*A*” are isomorphic.<sup>76</sup> This view depends on one’s stance on issues like “standard interpretation” and “standard models” (and is related to questions like “How are the real numbers constructed?”, “What are real numbers, sets of natural numbers or limits of Dedekind cuts?”, or “What is the standard model of real numbers?”). A third possibility is to stay within the syntactic realm (of the formal theory). Then one can treat variables of different sorts as being of yet another sort, namely the sort “*Real*”, if all field axioms are defined for all of them. In the one-sorted case, one can substitute “*Real*(*x*)” for every other predicate restriction of a quantifier (e.g., “ $\forall Real(x)...$ ” for “ $\forall V(x)...$ ”), if all field axioms are defined for the predicate.<sup>77</sup> That

---

confined within the boundaries of one physical theory. It would not be possible to compare different systems described by different theories with respect to the function they realize.

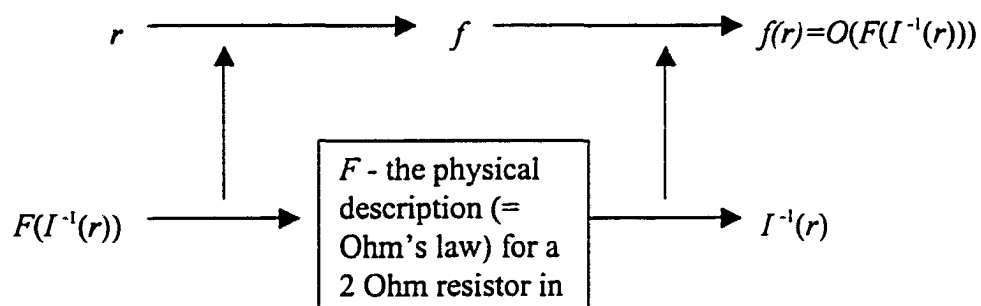
<sup>75</sup> It is an interesting fact that we are completely used to talking about “quantitative physical properties” in terms of numbers, so used that it seems impossible to leave numbers out: all physical properties have a qualitative and a quantitative aspect (e.g., 100 kg or 20 m/sec). One likely reason is that the language of physics is built upon the language of mathematics, i.e., the language of (real/complex) analysis. There are, however, other ways of defining quantitative aspects in physics (e.g., nominalizing physics in the sense of Field). Hence, a mapping from volts to *Reals* only *seems* to be an “identity” mapping.

<sup>76</sup> This is, of course, only true for the field axioms. If additional axioms are added (e.g., that every quadratic equation has a solution), then the *Rationals* might become excluded.

<sup>77</sup> Formally, this means that the theory is either extended by another sort plus all the axioms for that sort or that a new predicate is introduced together with all relevant axioms.

way one can avoid being forced to take a stand on either physical modeling paradigms or whether “identical up to isomorphism” is sufficient to “nail down” the set of *Reals* (i.e., if isomorphism is sufficient to determine a set of objects or if “more” is needed). Hence, I will use the term “syntactic isomorphism” in the following to emphasize that I have this third syntactic alternative in mind, even though I will treat isomorphisms by and large semantically for the sake of expositional clarity.<sup>78</sup>

The “common structure” of the two functional descriptions  $F$  and  $F'$  for resistor and wire, respectively, can now be viewed as the mapping  $f(x)=x/2$  from *Reals* to *Reals*. The “abstraction” over the physical dimension is achieved by supplying two syntactic isomorphisms: the *input encoding*  $I(x)$  from voltages to *Reals* and the *output encoding*  $O(x)$  from currents to *Reals*. Figure 5.1 depicts the relations among  $F$ ,  $f$ ,  $I$  and  $O$  (in this case for the resistor, but it really works for any physical system).



**Figure 5.1** The relation between the resistor and the function it realizes: given a certain *Real*  $r$ , the value  $f(r)$  is then obtained by taking the *encoding of the input*  $I(r)$ , applying it to the resistor, and then decoding the output  $F(I^{-1}(r))$ , using the *output encoding*, resulting in  $O^{-1}(F(I^{-1}(r)))$ , which is equal to  $f(r)$ .

<sup>78</sup> Note that this way even nominalists who are not committed to the existence of *Reals* can accept the following

This suggests a general/generic definition of what it means for a physical system  $S$  described by a (physical) theory  $P$  to *realize a function*  $f$ :<sup>79</sup>

*Definition 5.1:* A function  $f$  with domain  $D$  and range  $R$  is *realized* by a physical system  $S$  (describable in a theory  $P$ ) if and only if the following conditions hold:

1. There exists a (syntactic) isomorphic mapping  $I$  from the “input domain” of  $S$  to  $D$ <sup>80</sup>
2. There exists a (syntactic) isomorphic mapping  $O$  from the “output domain” of  $S$  to  $R$
3. There exists a function  $F$  that describes the physical property (=behavior) of  $S$  for the given input-output properties (i.e.,  $F$  is a mapping from the “input domain” of  $S$  to its “output domain” described in the language and by the laws of  $P$ ) such that for all inputs  $x$  the following holds:  $O(F(I^{-1}(x)))=f(x)$ .

Note that just requiring bijective mappings  $I$  and  $O$  is not enough to determine  $f$ : let  $I(x)=x$  for all  $x$  except for  $I(1)=0$  and  $I(0)=1$  (and the same for  $O$ ), and  $F(x)=x/2$ . Then  $f(1)=O(F(I^{-1}(1)))=O(F(0))=O(0)=1$ , although  $f(1)=1/2$  would have been the correct value.

---

definitions.

<sup>79</sup> Cummins (1989) defines a notion called “a device satisfying a function”, which *prima facie* looks very similar to my “a physical system realizes a function”. There are two main differences, however: first, Cummins defines the relation of satisfaction only for functions that have the same input and output domain as the physical system, and second, he requires that input-output criteria for a given device can be specified which will determine whether a given state of the system is an output value and, if so, which state the corresponding input value was. His definition does not use a physical description of the device (which would eventually lend itself to a functional specification), but involves counterfactuals (to specify “state-transitions”), and in the end falls prey to Putnam’s construction (since it does not specify the level of description of the physical device).

<sup>80</sup> One could “relax” the mapping by requiring that  $D$  only be a subset of the input domain of  $S$ , thereby allowing the system to also realize functions that are “less complex” (in very much the same manner that the identity function over

One could even argue that “isomorphism” is still too weak, since it does not distinguish between  $I$  and all  $I'$  such that  $I'(x)=a*I(x)+b$  for all  $a, b$  in the domain of  $f$  (see also Cummins (1989), pp. 102). The same is true for the output encoding:  $O'(x)=c*O(x)+d$  for all  $c, d$  in the range of  $f$ . Therefore, any one of the functions  $f'(x)=cf(ax+b)+d$ , i.e.,  $c(ax+b)/2+d$  (which reduces to  $a'x/2+b'$  for some  $a'$  and  $b'$ ) seems to be a possible candidate for the function realized by  $F$ . However, not all of them are reasonable, since  $f'(x)=x$  for  $a'=2$  and  $b'=0$ , for example, and this obviously defeats the purpose of the resistor. In a way, there is only one (isomorphic) mapping that can do the job, the one taking “0-volt” to “0”, “1-volt” to “1”,  $n$ -volt (where “ $n$ ” is defined as the sum of  $n$  “1-volt” elements) to  $n$ , etc. This also suggests an answer to a similar problem posed by Cummins, who introduced the notion “direct interpretation” for a particular isomorphism between symbolic and physical input/output, which he admittedly could not define (Cummins, 1989, p. 104).

The above definition, being cast as a *schema* (parameterized by the theory  $P$ ) to allow for greatest possible generality, is naturally quite vague. It can neither specify what “physical system” means (let alone its behavior) nor what the inputs and outputs are for that system, but that is not its purpose anyway. These “empty places” will have to be filled in with particular values from the respective theory  $P$  to make the definition complete. In the above examples, one would substitute “circuit theory” for “ $P$ ”, “resistor” for “physical system” together with the appropriate inputs and outputs (i.e., voltages and currents at the two ports of the resistor) as described. Other problems,

---

the *Reals* contains the one defined over the *Rationals* which in turn contains the one defined over the *Integers*). See section 5.6.



however, such as the (short) time lag between the point of application of the voltage/current and the current on the output side (electrons moving at almost the speed of light), or the range of functionality of the physical system, require attention and will be tackled as we start to mould this definition guided by practical constraints.

#### 5.4 Analog Electric Circuits

So far, we have talked about the fact that electronic components can be described by an “input-output” function (which will naturally differ from component to component). Each individual function is rather restricted, hence quite simple. So in order to allow for more complex functions, one could consider more complex arrangements of these components (systems of components). Take, for example, high-pass filters (consisting of a resistor and a capacitor) which realize functions from frequency to frequency that will be the identity function for high frequencies and the constant function  $f(x)=0$  for low frequencies. One could also consider systems that change their input-output behavior depending on other external factors (e.g., resistors change their resistance dependent on their temperature). In particular, these external factors might be exploited to make a system more versatile. Take, for example, an amplifier which contains a potentiometer to allow for adjustment of the amplification factor. It will then realize the function  $f_p(x)=x*p$  (where  $p$  is the amplification factor,  $0 < p \leq 100$ , say). Notice that this function is parameterized by and dependent on  $p$ .<sup>81</sup> These kinds of functions are especially practical, because they allow a single system to realize multiple functions; in other words,

---

<sup>81</sup> It is worth pointing out that parameters differ from inputs in an essential way: they are normally adjusted until a desired value is reached (e.g., the volume of the amplifier before the guitarist starts to play), and then they remain set

they make it a “multi-functional” system. Every radio, for example, is such a multi-functional system, being capable of receiving multiple channels and extracting the low frequency information that was coded in the high frequency at different volumes (its function is parameterized by “volume control”, “channel selection”, etc.).<sup>82</sup>

Although all components work in the real world and are, therefore, subject to time and space constraints, I have not taken either into account. Take, for example, a delay circuit (which outputs incoming signals, i.e., voltages, after a certain delay  $d$ ). This system seems to realize the identity function  $f(x)=x$ , but we would agree that it differs in an essential way from a single copper wire (which also realizes the identity function): the former only computes identity if time is left out; otherwise it realizes the function  $f(x,t)=g_x(t-d)$  (where  $f(x,y)$  is 0 for all  $y<d$  and  $g_x(t)$  is the function that describes the value of  $x$  at time  $t$ ). This function is much more complex than the identity function, which now can be seen as special case when  $d=0$ .<sup>83</sup> Since time dependencies between input and output do not have to be constant either—just take a delay where the delay factor is a multiple of the magnitude of the input signal—even sophisticated input-output behaviors (with respect to time) are possible, once *time matters*. Thus, the definition of “realization of a function” has to be augmented by two time factors: an abstract time which is attached to the function  $f$  and the real world time as described by  $P$ . In the following “*RealTime*”

---

to this value, whereas the input will continue to change. This is, of course, only a rough cut, but it hints at the role of parameters in reconstructing the concept of “programs” from physical peculiarities of certain systems.

<sup>82</sup> Adding adjustable parameters to physical systems actually marks a crucial step in my endeavor of representing the computational story. Once the transition from realizing one function to realizing multiple functions has been made, it is only natural to ask: “What *class of functions* does a system realize?”. In the extreme case this class might turn out to be the whole class of functions itself (for a given definition of “realization of a function”). Those systems (the existence of which has to be argued for, of course) could then be called “universal” (with respect to the given definition).

<sup>83</sup> If time is taken seriously, then no physical system will ever realize the identity function, since  $d$  will never be 0.

will always denote the set of times derived from the theory  $P$  (e.g., the projection of the forth coordinate of space-time), whereas “*Time*” is supposed to denote the set of abstract times (intervals or points).

Another important factor is the physical condition of the system when input is applied, since, in most cases, the input-output behavior of the system will depend on it. The output of a fully charged capacitor for a given input differs significantly from its output for the same input if it is uncharged, for example. Hence, it is essential that the physical description of the behavior of a system also contain a (complete) description of its physical condition at the time when the input is applied. I will assume from now on that this description of the physical condition is part of the description of  $S$  (e.g., the initial conditions for a dynamic system).

Finally, it has to be assumed that the physical system under consideration behaves “normally” throughout the interval during which inputs are applied and outputs are measured. If environmental conditions are such that the laws that describe the behavior of  $S$  (i.e., in normal circumstances) do not apply any longer, the system does not work “normally”. Obviously, this notion involves agreement on what “normal behavior” means: “[.] fixing the conditions of normal operation is crucial for making determinate claims about what function a system is computing” (Stabler, 1987, p.10). How one can reach such an agreement is yet another issue. I tend to follow Hardcastle’s suggestion that no principled answer is available (Hardcastle, 1995, p. 306). Engineering practice, however, usually does not have these theoretical concerns and agreement whether or not a system is behaving properly is reached guided by pragmatic considerations such as “reproducibility of a behavior”, “usability of a behavior for building devices”, “reliability

of predicted behavior”, etc. I will, therefore, ignore issues of explanation in the philosophy of science that normally arise in the context of “normal behavior of physical systems”, but instead assume that there is an agreement at least among engineers on what it means to “behave normally” for all systems under consideration (it is very likely that the physical description of systems is not very reliable if no such criteria are available in the first place).

*Definition 5.2:* A function  $f$  with domain  $D \times Time$  and range  $R \times Time$  is realized by a physical system  $S$  (describable in a theory  $P$ ) if and only if the following conditions hold:

1. There exists a (syntactic) isomorphic mapping  $I$  from the “input domain” of  $S$  to  $D$
2. There exists a (syntactic) isomorphic mapping  $O$  from the “output domain” of  $S$  to  $R$
3. There exists a (syntactic) isomorphic mapping  $T$  from  $RealTime$  to  $Time$
4. There exists a function  $F$  that describes the physical property (=behavior) of  $S$  for the given input-output properties over  $RealTime$  (i.e.,  $F$  is a mapping from the “input domain”  $\times RealTime$  of  $S$  to its “output domain”  $\times RealTime$  described in the language and by the laws of  $P$ ) such that for all  $t \in Time$  and all  $x \in D$  the following holds: if  $F(I^{-1}(x), T^{-1}(t)) = \langle y, r \rangle$ , then  $\langle O(y), T(r) \rangle = f(x, t)$  (for  $y \in$  “output domain of  $S$ ” and  $r \in RealTime$ ).

This straightforward augmentation has wanted as well as unwanted effects. The input-output behavior of a system is now described as a functional relation between two graphs: the graph of the input signal and the graph of the output signal over time. This way, the temporal behavior of a system can be completely described. Suppose, for example, that it

takes time  $d$  for the electrons to pass a wire, then the function realized by the wire is basically the one described above for the delay. However, abstract functions now have a “time” attached to them and it is not quite clear what it means to have a “timed” version of the addition function, say. One possible answer is to argue that physical systems simply do not realize “timeless” functions (i.e., functions from timeless input to timeless outputs). In other words, every input to a physical system has to happen in time and hence the function which is realized by the system has to have a time parameter attached to it (e.g., a *Real* parameter).<sup>84</sup> In some cases this time parameter is exactly what distinguishes one system from another; given a delay circuit with 10 msec delay and another with 15 msec, dropping this parameter would mean no longer being able to tell the two systems apart.<sup>85</sup> Often, however, the time relation between input and output does not matter. Two wires, made of different material with different lengths, might (theoretically) still have the same resistance (except that the output is sooner available in one wire than in the other). In this case, we would like to ignore the time lag between input and output in order to be able to speak of “the same function that both realize”. So, we first have to define what it means for a system to realize a “timeless” function:

---

<sup>84</sup> This point has been stressed by adherents of dynamicism as one of the major shortcomings of the standard notion of computation: computations are defined in terms of *computational steps*, not in terms of *time* (e.g., see van Gelder, 1998).

<sup>85</sup> This idea could eventually give rise to a very different approach to computation, an approach that is *essentially* built upon the temporality of physical processes. Abstraction would, of course, be possible in many directions (e.g., towards digitality, see the next section), but duration and temporal order, being central, defining concepts, could never drop out during an abstraction process. This view on computation might come closer and do more justice to the behavior of what are called “embedded systems” (but I will not dwell on this here). Interestingly, operating or real-time system designers have been making a living out of coping with real-time constraints for quite some time.

*Definition 5.3:* A “timeless” function  $f(x)$  with domain  $D$  and range  $R$  is realized by a system  $S$  (describable in a theory  $P$ ) if and only if the following conditions hold:

1. There exists a (syntactic) isomorphic mapping  $I$  from the “input domain” of  $S$  to  $D$
2. There exists a (syntactic) isomorphic mapping  $O$  from the “output domain” of  $S$  to  $R$
3. There exists a function  $F$  that describes the physical property (=behavior) of  $S$  for the given input-output properties over *RealTime* (i.e.,  $F$  is a mapping from the “input domain” $\times$ *RealTime* of  $S$  to its “output domain” $\times$ *RealTime* described in the language and by the laws of  $P$ ) and there exists a “delay-function”  $d(x)$  from inputs to times (derived from  $F$ ) such that for all  $r \in \text{RealTime}$  and all  $x \in D$  the following holds: if  $F(I^{-1}(x), r) = \langle y, r + d(x) \rangle$ , then  $O(y) = f(x)$  (for  $y \in$  “output domain of  $S$ ”).<sup>86</sup>

Notice that the time delay of the output is defined as a function of the input because the relation between time lag and input will not be constant in all systems, but may depend on specific inputs (e.g., if input and output to a capacitor are currents, then the capacitor will produce delayed output depending on its capacity). This definition can then be used to show that both wires discussed above, in fact, realize the same “timeless” function  $f(x)=x$ : there exist two delay-functions  $d_1(x)$  and  $d_2(x)$  (from inputs to times), namely  $d_1(x)=0.01$  and  $d_2(x)=0.015$  for all inputs  $x$ , such that if  $x$  is the input to both systems at time  $t$ , then one system will output  $x$  at time  $t+d_1(x)$  and the other will output  $x$  at time  $t+d_2(x)$ . The last step accomplishes a crucial abstraction: the actual duration, the link

---

<sup>86</sup> Note that if  $d$  were totally unconstrained, very strange time lags would be allowed in principle. Fortunately this is not the case, since the physics of the system (of components in this case) will put restrictions on the time-dependencies between input and output (such that there will be no “jumps”, no “points of discontinuity”, etc., the function will rather be “smooth”, “continuous”, etc.): time dependencies will be law-like. More precisely,  $d(x)$ , being extracted from  $F$ , cannot be a “strange” function without rendering the whole system abstruse, if not absurd.

between input and output, the “behavioral” processes that the system exhibits when an input is applied, resulting in a delayed output, is neglected in favor of a “timeless” mapping! This way various different systems will realize the same timeless function independent of the time lag between input and output, i.e., their “speed”. It now becomes possible to compare different systems with respect to their “functionality”, when time does not matter.<sup>87</sup>

Three remarks seem necessary at this point:

1) Physics certainly places restrictions on the domain of the input, the domain of the output, and the system itself. For example, one cannot expect to apply arbitrarily high currents to a wire of a given size, not only because it would be hard to generate them, but also because the wire would melt. This kind of dependence automatically delimits the range of the abstract function that a physical system is able to realize. So, strictly speaking, a wire that realizes the identity function according to the above definitions does not realize the *whole* function, but only a part of it (the part in which the wire “operates normally” according to the laws of physics; in the other part it will, of course, still obey the laws of physics, but different factors will come into play, and the original equations will no longer apply). There are also physical reasons why certain components have to have (at least or at most) a certain size (e.g., a capacitor that can store the charge of 500 F will be too big to be soldered into a standard circuit board). These constraints, in turn, might have an influence on the time-factor of the system, as space and time are inseparably interwoven.

---

<sup>87</sup> This is again a commonplace for computer practitioners: a PCs with a 133 MHz CPU and one with 200 MHz CPU (other things being the same) differ only in speed, but not in functionality.

2) There is also a limit to the accuracy with which an input signal can be generated and an output signal can be recognized (see also Haugeland, 1982). This problem results from the limits of measurability of physical magnitudes; no measurement will be 100% exact, but will always contain an error—i.e., will be within some (small) range of the actual value. Hence, from a practical point of view, the mere knowledge that a certain circuit actually realizes a very complicated function might not be of great help if there is no way of making inputs precise enough and/or reading off its outputs with sufficient precision.

3) One final remark concerning the nature of electrical circuits. When one hears “electric component”, one automatically associates this term with man-made parts that are used to solder circuits. Thus, one implicitly assumes a certain physical structure: silicon-arsenide molecules, gold wires, etc. And, in fact, at the beginning of this section, I suggested exactly that. However, one could broaden one’s perspective and also subsume “natural” (i.e., non man-made) *electrical components and circuits*. In particular, one could view neurons as circuits with electrical properties (a description of the functionality, of course, will include laws of chemistry, cell-biology, etc.). Given their electrical properties and the nature of their inputs and outputs, neurons then realize certain functions (the ones that can be related in the above sense to their physical make-up). And the fact that neurons are made of “biologically describable” stuff does not mean that there might not be a man-made “artificial neuron” (possibly made out of inorganic substances) with the same properties (or if not entirely the same, then at least with respect to the input-output behavior) that in turn will realize the same functions. So one way to understand natural cognitive systems is to analyze what functions neural circuits realize,



at exactly the electric level of description. These functions can then be compared (with or without taking time into account) with the functions realized by artificial systems, or analyzed mathematically. In any case, it will be possible (once the physics of neurons is fully understood, if that is possible...) to determine the class of functions that are realized by neurons. And that, in turn, will eventually allow us to look at the complex input-output behaviors of networks of neurons (which then can be described as possible combinations—such as compositions, iterations, etc.—of the class of functions realized by a single neuron).

### 5.5 Digital Electronic Circuits

The last section showed that the electric level of description allows us to describe a vast variety of complex circuits together with the functions they realize. However, I have already mentioned that theory is only part of the story, and that practice is quite another. Although one can theoretically define devices that realize very interesting functions, it might not be possible to build and/or detect them (due to a lack of sufficiently precise tools, measurement instruments, etc.). Another factor to consider is the impact of the environment on real-world devices. The influence of noise levels and other disturbances might prevent the circuit from functioning according to the “idealized” laws of physics (“idealized” in the sense that all possible environmental influences are normally *not* taken into considerations in standard formulations of physical laws when used in practical settings...).<sup>88</sup> Hence, every description will have a “small” error term  $\epsilon$  attached to it for

---

<sup>88</sup> Interestingly, the proof of Putnam’s Realization Theorem relies on the so-called “Principle of Non-Cyclical Behavior”, which is true of systems that are not shielded from environmental influences.

the margin of error within which the system's behavior cannot be *exactly* predicted. However, once such an error boundary is given (together with reliable physical conditions that the system will stay with sufficiently high probability within these boundaries taking the environment of the system, etc. into account), then one can use these systems for an inquiry into the nature of the functions they realize. The main difference now (compared to the previous definition 5.3) is that the output of such a system will not be the same under the same input conditions, but always within a certain interval (actually, the same will be true for the input as well, since it is practically impossible to generate arbitrarily precise inputs—or so at least it is commonly believed).

This change in precision (i.e., the relaxation of the constraints) has to be taken into account in the formulation of a new version of the definition “realization of a function”. Note that four different error terms will have to be defined in advance ( $\epsilon_{\text{din}}$ ,  $\epsilon_{\text{dout}}$  for input-output magnitudes and  $\epsilon_{\text{tin}}$ ,  $\epsilon_{\text{tout}}$  for input-output times):

*Definition 5.4:* A “timeless” function  $f(x)$  with domain  $D$  and range  $R$  is (practically) realized by a system  $S$  (describable in a theory  $P$ ) with  $\epsilon_{\text{din}}$ ,  $\epsilon_{\text{dout}}$  for input-output magnitudes and  $\epsilon_{\text{tin}}$ ,  $\epsilon_{\text{tout}}$  for input-output times if and only if the following conditions hold:

1. There exists a (syntactic) isomorphic mapping  $I$  from the “input domain” of  $S$  to  $D$
2. There exists a (syntactic) isomorphic mapping  $O$  from the “output domain” of  $S$  to  $R$
3. There exists a function  $F$  that describes the physical property (=behavior) of  $S$  for the given input-output properties over *RealTime* (i.e.,  $F$  is a mapping from the “input

domain” $\times RealTime$  of  $S$  to its “output domain” $\times RealTime$  described in the language and by the laws of  $P$ ) and a “delay-function”  $d(x)$  from inputs to times (derived from  $F$ ) such that for all  $r \in RealTime$  and all  $x \in D$  the following holds (for notational convenience abbreviate the error interval for  $y$  determined by  $\varepsilon$  as  $INT(y, \varepsilon) =_{\text{def}} \{y | y - \varepsilon \leq y \leq y + \varepsilon\}$ ): if  $F(I^{-1}(INT(x, \varepsilon_{\text{din}})), INT(r, \varepsilon_{\text{tin}})) = \langle Y, Z \rangle$ , then  $O(Y) \subseteq INT(f(x), \varepsilon_{\text{dout}})$  and  $Z \subseteq INT(r + d(x), \varepsilon_{\text{tout}})$  (for  $Y \subseteq$  “output domain of  $S$ ”).<sup>89</sup>

This definition works in a practical setting as follows: suppose the system  $S$  is given together with a physical description of its functionality  $F$  and measurement errors  $\varepsilon_{\text{din}}$ ,  $\varepsilon_{\text{dout}}$  for the input-output magnitudes, and  $\varepsilon_{\text{tin}}$ ,  $\varepsilon_{\text{tout}}$  for the input-output times. For every input  $x$  to the system (which could be anywhere between  $I^{-1}(x - \varepsilon_{\text{din}})$  and  $I^{-1}(x + \varepsilon_{\text{din}})$ ), the output needs to be within  $O^{-1}(f(x) - \varepsilon_{\text{dout}})$  and  $O^{-1}(f(x) + \varepsilon_{\text{dout}})$  (under the given mappings  $I$  and  $O$ , of course). If  $f$  (and  $F$  for that matter) is unknown, then it can be approximately determined by repeatedly applying various values of  $x$  to the system. Engineers, for example, when they measure voltages at different places in a circuit in order to find the reasons for the system’s improper functioning, use implicitly a definition of the above kind. In general, every measurement will reach the limits of precision at some point, and then the “original” value can only be “estimated” from the results of many different measurements (within a certain interval).

---

<sup>89</sup> Two notational remarks: I used ‘ $f(X)$ ’ (for a set  $X$  and a function  $f$ ) to mean ‘ $\{f(x) | x \in X\}$ ’, ‘ $\langle X, Y \rangle$ ’ to denote the Cartesian product of  $X$  and  $Y$ . Also, if time errors depend on inputs  $x$ , this can be accounted for by using error *function terms* (such as  $\varepsilon_{\text{tin}}(x)$  and  $\varepsilon_{\text{tout}}(x)$ ) instead.

Although this definition seems more appropriate given practical limitations, it is still not satisfactory. The most urgent problem is certainly one that cannot be accounted for by a definition in principle: what if the errors are (too) large? Then either the (input/output data-entry/measurement) instruments have to be refined or the system is probably of no practical use (independent of the function it *theoretically* realizes...). If we restrict ourselves, therefore, to systems with practically acceptable error margins, then the following three increasingly important shortcomings call for improvements:

Firstly, notice that the range of data-entry/measurability is limited for every physical system (as already mentioned at the end of the previous section). From a practical point of view, there is not even a single physical system that realizes the addition function for all integers. The best that can be hoped for are systems that realize parts of that function. So, further restrictions have to be imposed on inputs and outputs: the domain of the abstract function has to be restricted to the interval  $[x_{\min}, x_{\max}]$  which is determined by 1) the data-entry/measurability constraints of devices producing the inputs and measuring the resulting outputs and 2) the range within which the physical system functions *normally* according to the laws of physics (a resistor, for example, does not behave *normally* when it starts to melt because the applied current is too high).

Secondly, the relation between input and output errors has hitherto been neglected. The only requirement imposed was that they be small enough to be of practical use. However, their relation becomes critical as soon as networks of circuits (which are constructed by connecting outputs of some circuits to inputs of others) are considered. Assume that three serially connected resistors form a circuit. Suppose that all resistors have an output error which is 10 times greater than their input error. Then the output

error of the whole circuit will be 1000 times the input error (and that might already be unacceptable). It seems, therefore, reasonable to require that the output error be less than or equal to the input error to allow for the construction of complex circuits, while at the same time keeping the overall error small.<sup>90</sup>

Lastly, but most importantly, there are generally problematic cases in which the difference of two inputs to a system,  $x_1$  and  $x_2$ , whose “error intervals”  $[x_1 - \epsilon_{in}, x_1 + \epsilon_{in}]$  and  $[x_2 - \epsilon_{in}, x_2 + \epsilon_{in}]$  overlap, is crucial. It is not clear at all how such a system could be of practical use. The main problem is that the function  $f$  which is supposed to be realized by  $S$  will be real-valued (as a consequence of  $P$  for most theories  $P$ ), whereas the function describing the system’s behavior (in terms of measurability) seems rather discrete-valued (a function from intervals of *Reals* to intervals of *Reals* with interval size  $2*\epsilon$ ). Or to use a metaphor: the abstract function is too “precise” for its “smudged” worldly counterpart. As long as there are overlapping regions between intervals in  $F$  which correspond to two distinct values of  $f$ , there will be cases in which, given a value  $x'$  in an overlapping region, one can neither determine the original  $x$  nor the resulting  $f(x)$ .<sup>91</sup> Hence, it would make more sense to let  $f$  take values in the *Integers* (or *Rationals*) rather than the *Reals*. This would allow one to map discrete values to unique intervals if, in addition, the data-entry/measurement errors are small enough so the intersection of any two images of

---

<sup>90</sup> Note that there are really *two* output errors involved: the first is determined by the exactness of the measurement of the output—this is the one we have considered so far. The second is determined by the physical system itself, by the degree to which the system deviates from its formal description (e.g. imperfection and/or impurities of the material). Although the (overall) output error is really a combination of both individual errors, for theoretical purposes *one error term* that comprises both suffices: simply take the product of both error terms (especially since it might not be clear which error actually contributes more/most to overall error).

<sup>91</sup> The reason is that no bijection exists (hence no isomorphism either) between the *Reals* and disjoint intervals of the *Reals* of size  $2*\epsilon$  for any  $\epsilon > 0$ .

discrete values, i.e., intervals, is empty. In such a system, all values of  $f$  could be measured/produced, hence the system would not only *theoretically*, but also *practically* (and verifiably) realize the function  $f$  (of course, only within the boundaries of  $[x_{\min}, x_{\max}]$ ). Notice that this last requirement implies that only certain *finite* (“timeless”) functions can ever be totally realized (since there are smallest and largest values determined by the errors and/or physical possibilities of data-entry/measurement such as energy constraints, uncertainty, observability, etc. which in turn are determined by size constraints of the physical system...).

Taking all three modifications into account, we can define what it means to realize a part of a discrete-valued function  $f$  for given error terms  $\epsilon_{\text{data}}$  and  $\epsilon_{\text{time}}$  (which describe the measurement error of data and time, respectively—the output error is now assumed to be at most the input error, hence only one error term is needed):<sup>92</sup>

*Definition 5.5:* A “timeless” discrete function  $f(x)$  with domain  $D$  and range  $R$  is (practically) realized within  $[x_{\min}, x_{\max}] \subseteq D$  realized by a system  $S$  (describable in a theory  $P$ ) with errors  $\epsilon_{\text{data}}$  for input-output magnitudes and  $\epsilon_{\text{time}}$  for time if and only if the following conditions hold:

1. There exists a (syntactic) isomorphic mapping  $I$  from disjoint intervals of the “input domain” of  $S$  (where the interval length is  $> 2 * \epsilon_{\text{data}}$ ) to  $D$

---

<sup>92</sup> The “construction” of the syntactic isomorphism is more complicated in this case, since intervals of a certain length need to be defined for input, output, and time domain. Then axioms for the discreteness of  $f$  need to be added, etc. to allow for an appropriate formal treatment.

2. There exists a (syntactic) isomorphic mapping  $O$  from disjoint intervals of the “output domain” of  $S$  (where the interval length is  $>2*\epsilon_{data}$ ) to  $R$
3. There exists a function  $F$  that describes the physical property (=behavior) of  $S$  for the given input-output properties over time (i.e.,  $F$  is a mapping from the “input domain” $\times RealTime$  of  $S$  to its “output domain” $\times RealTime$  described in the language and by the laws of  $P$ ) and a “delay-function”  $d(x)$  from inputs to times (derived from the  $F$ ) such that for all  $r \in RealTime$  and all  $x \in [x_{min}, x_{max}]$  the following holds:  
 $F(\Gamma^1(x), r) = \langle O^1(f(x)), r' \rangle$  (where  $r' \in [r+d(x)-\epsilon_{time}, r+d(x)+\epsilon_{time}]$ ).<sup>93</sup>

This definition has achieved a great abstraction step: the magnitudes of physical dimensions have been “discretized” to guarantee practical applicability, i.e., continuity has been given up. However, the discrete values are still closely tied to the continuous values that describe the functionality of  $S$  in  $P$ . What is different, in more metaphorical terms, is that a grid (with box size  $>2*\epsilon_{data}$ ) has been superimposed on the functional graph, and only the points where the graph intersects with the grid are now considered.

Comparing definition 5.5 to definition 5.1, it becomes obvious how by incorporating physical and practical constraints, one arrives at a very restricted definition of what it means for a physical system to realize a function. Surely, we could have been satisfied with definition 5.2 or even definition 5.3, and for theoretical purposes they are just fine. But since we are interested in systems that we can actually *use* for various tasks (and might have to *build* in order to use them), we have to restrict ourselves to accessible ones,

systems that can be utilized because they allow us to generate inputs and measure outputs.

One of the tasks that systems can be used for is “computation”—i.e., they can be used as “computers”. For something to qualify as a “computer” it has to be at least a useable, physical system, which allows for data input and for measurable output, which works within reasonable time constraints and is sufficiently reliable.<sup>94</sup> Of course, other conditions will have to be added depending on one’s view of “computers” such as “executing an algorithm”, etc. Some systems that realize functions according to definition 5.5 have all those properties by virtue of discretizing magnitudes and restricting the domain of possible values, others—where time plays a crucial role—require additional constraints on time (see definition 5.6). Not for all such systems will it be possible to define small error values, but some systems, those that are especially designed to facilitate the input/output-mappings for certain intervals, will be extremely reliable (because intervals are “far” apart and error values, for both time and magnitude, are small). Furthermore, these systems will be constructed to realize functions with a very small, finite domain (of mostly only *two* values!).

These properties together have often been summarized as the “digitality” of a system (see Haugeland, 1982, p. 215, or Haugeland, 1996, p. 9), i.e., the fact that there are “reliable” procedures for applying input and measuring output within the operating limits

---

<sup>93</sup> One could require the weaker  $F(I(x),r) \subseteq O(f(x),r')$  to make the error resulting from an inaccurate description (“idealization”) of the physical system itself explicit.

<sup>94</sup> Reliability is really a tricky issue. On the one hand, it is guaranteed by the laws of physics *qua* laws (i.e., that is exactly what it is to be a law: *to be reliable*), on the other hand, certain physical laws describe processes only up to probabilities. In those cases, “reliability” is automatically reduced to “(high) probability”. In any case, how reliable a system will be at the end depends on the theory  $P$  that describes its nature and functionality as well as on practical constraints (such as material, size, environmental influences, etc.).



of the system and that there are finitely many distinct, discrete values given those limits. “Digital” means something like “of, relating to, or using *calculation* directly with *digits* rather than through measurable physical quantities” (Webster’s dictionary, 1995, italics are mine).<sup>95</sup> Although systems that fall under definition 5.5 do not necessarily *calculate*, let alone with *digits*, they provide all the prerequisites that a system must have to *support* digits, since they *realize* discrete (timeless) functions by virtue of the physical laws that describe their behavior for a given set of inputs. Hence I will call them “digitality supporting systems”.

Many electric components are especially designed to *support two digits*, so-called Boolean circuits (such as AND-gates, OR-gates, NOT-gates, etc.). Most of them realize very simple functions such as the XOR function:  $f(x,y)=0$  if  $x=y$ , otherwise  $f(x,y)=1$  (where ‘0’ and ‘1’ are the two digits). Note that it is sometimes necessary to explicitly bring time into the picture again, especially to describe functions that use feedback (or to avoid unwanted behaviors in complex networks of Boolean circuits where the delay time of each unit becomes a critical factor in the overall performance). This requires us to make a final abstraction and change  $f$  in definition 5.5 from timeless to “discrete time” (analogous to definition 5.2), where the input and output domains of  $f$  now also contain “discrete time points” from the set *Time* (normally modeled by integers):

*Definition 5.6:* A discrete function  $f(x,t)$  with domain  $D \times \text{Time}$  and range  $R \times \text{Time}$  is (practically) realized within  $[x_{\min}, x_{\max}] \subseteq D$  by a system  $S$  (describable in a theory  $P$ ) with

---

<sup>95</sup> Haugeland (1982, p.214) argues that digits do not necessarily have to represent anything.

errors  $\epsilon_{\text{data}}$  for input-output magnitudes and  $\epsilon_{\text{time}}$  for time if and only if the following conditions hold:

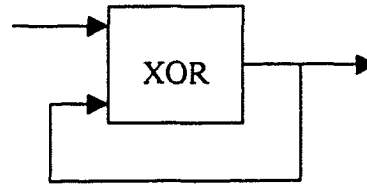
1. There exists a (syntactic) isomorphic mapping  $I$  from disjoint intervals of the “input domain” of  $S$  (where the interval length is  $>2*\epsilon_{\text{data}}$ ) to  $D$
2. There exists a (syntactic) isomorphic mapping  $O$  from disjoint intervals of the “output domain” of  $S$  (where the interval length is  $>2*\epsilon_{\text{data}}$ ) to  $R$
3. There exists a (syntactic) isomorphic mapping  $T$  from disjoint intervals of *RealTime* (where the interval length is  $>2*\epsilon_{\text{time}}$ ) to *Time*
4. There exists a function  $F$  that describes the physical property (=behavior) of  $S$  for the given input-output properties over time (i.e.,  $F$  is a mapping from the “input domain” $\times$ *RealTime* of  $S$  to its “output domain” $\times$ *RealTime* described in the language and by the laws of  $P$ ) such for all  $t \in \text{Time}$  and all  $x \in [x_{\min}, x_{\max}]$  the following holds: if  $F(I^{-1}(x), T^{-1}(t)) = \langle X, Z \rangle$ , then  $\langle X, Z \rangle = \langle O^{-1}(x'), T^{-1}(t') \rangle$  (where  $f(x, t) = \langle x', t' \rangle$ ).<sup>96</sup>

This definition looks very much like definition 5.2, except that all values are discrete instead of continuous. So, the above XOR function can now be captured as  $f(x, y, t+1) = 0$  if  $x=y$  at  $t$ , otherwise  $f(x, y, t+1) = 1$  (and  $f(x, y, 0) = 0$ , say). And it can be used to define more complicated functions using feedback over time, such as the “oscillator” function  $g$  which alternates between 0s and 1s, once the input changes from 0 to 1:  $g(x, t+1) = f(g(x, t), x, t+1)$

---

<sup>96</sup> Again,  $\langle X, Z \rangle \subseteq \langle O^{-1}(x'), T^{-1}(t') \rangle$  would be the weaker requirement (see footnote 87).

and  $g(x,0)=0$  (it will be *realized* by a simple XOR gate where the output is fed back into its second input line).<sup>97</sup>



**Figure 5.2** The oscillator circuit: once the input is changed from 0 to 1, the output will oscillate between 0 and 1 as long as the input is 1.

To summarize the achievements so far: starting at the level of electric components and the physical descriptions of their properties, I defined the notion “a system realizes a function” where the system consisted of components describable in terms of the physics of electricity. This notion was then—step by step—refined to account for practical problems regarding precision of production and measurement of signals, reliability, range of functioning, environmental influences, etc. The final definitions 5.5 and 5.6, respectively, related a discrete function (with a time parameter) to *digitality supporting* systems, i.e., systems that are of practical significance because inputs to them can be generated, and outputs (occurring after a limited time, possibly depending on the nature of the inputs) can be recognized and measured.<sup>98</sup>

---

<sup>97</sup> Notice that the functional definition of  $g$  resembles the *scheme of recursion* where recursion is defined over time (this is no coincidence, but I will not be able to explicate the connection here).

<sup>98</sup> Note that all definitions assumed only one input domain  $D$  and one output domain  $R$ , but, of course, they can be straight-forwardly extended to complex domains  $D^n$  and  $R^m$ .

## 5.6 From Circuits to Digital Systems

Definitions 5.5 and 5.6 have made two major abstractions (as compared to definitions 5.3 and 5.2), but neither times nor input/output magnitudes are completely “decoupled” from the concrete. The relation between input/output magnitudes as well as the one between points in time (i.e., the space-time metric) is still reflected in the input-output mappings together with their respective error terms. Values still stand for themselves and time points still have their unique place in the continuous flow of time, whereas in *genuine digital systems only* order matters, (spatial or value) distance and duration, i.e., metric distances, are secondary (if defined at all). It is not essential how “far apart” any two consecutive points are in time or space (of course, within limits) to be a particular digital system. In fact, different digital systems are shown identical *qua digital system* exactly by virtue of abstracting over physical peculiarities. Two steps must, therefore, be undertaken in order to prepare physical systems for *genuine digitality*: existential quantification over the particular error terms and a relaxation of the input-output mappings from isomorphisms to isomorphic embeddings. The first takes care of spatial and temporal distance of value and time, respectively. Different physical systems will then realize, i.e., *be the same digital system*, even though they differ with respect to their error terms and the magnitudes of their input-output values. Take, for example, two binary AND-gates which use different voltages for the binary values 0 and 1 and have different gate times. Both will now realize the *same AND-gate* (because of existential quantification over their error terms). To see that this is not sufficient, however, consider a ternary AND-gate,  $AND_3$ , using the values 0, 1, and 2 (the strong Kleene AND-

function, say). Assume that two different physical systems,  $S_3$  and  $S_{10}$ , are given.  $S_3$  operates within  $[0,4]$  Volts and “maps”  $\{0,1,2\}$  to voltages according to  $I(x)=O(x)=[x+1-0.1, x+1+0.1]$  with input-output and time errors of 0.1 (where the time mapping is given by  $T(x)=[x+1-0.1, x+1+0.1]$ ).  $S_{10}$  operates within  $[0,130]$  Volts and “maps” values from  $\{0, \dots, 9\}$  to voltages according to  $I(x)=O(x)=[(x+1)*(x+1)-0.5*(x+1), (x+1)*(x+1)+0.5*(x+1)]$  with input-output and time errors of  $0.5*(x+1)$  (the time mapping is again given by  $T(x)=[x+1-0.1, x+1+0.1]$ ). Whereas the former is built to work for ternary systems only, the latter is thought to work for decimal systems. Notice that input-output mappings in  $S_{10}$  are not linear, but quadratic (because they depend on the input-output error). Suppose now, that  $S_{10}$  is used as  $AND_3$  instead of  $S_3$  (because the following voltages: “100” ( $\pm 10$ ) for “1”, “10” ( $\pm 10$ ) for “0”, and “50” ( $\pm 10$ ) for “2” are required in a given practical setting, say). Then  $S_{10}$ , realizing the decimal AND-gate  $AND_{10}$ , also *realizes*  $AND_3$  under the *isomorphic embeddings*  $I^*$  and  $O^*$  obtained by composing  $I$  and  $O$  with  $E$  defined by  $E(0)=1$ ,  $E(1)=9$ , and  $E(2)=4$ .<sup>99</sup> So, both systems realize the same (abstract) function, namely  $AND_3$  under relaxed input-output constraints (“digits are decoupled from the grid imposed on  $F$  by the input-output mappings and the error terms”). And the speed of the circuits is negligible as long as it stays within some limits determined pragmatically by the purpose of the circuit’s use. Yet, it is important to keep in mind that these mappings are still not arbitrary (as they still preserve the relation between different inputs); rather they permit us to “pick and choose” specific values as

---

<sup>99</sup> Note that the order of the ternary elements induced by this mapping will be  $0 < 2 < 1$ , although nothing depends on the order in this. However, there will be other cases where the order matters (e.g., in an ADDER circuit).

digits (if only the distinctness of digits and neither their order nor relative magnitudes matters).

For some circuits, the current output does only depend on the current input. For many others, the previous output will matter, too, especially for those circuits that allow (internal) feedback. The current output of the “oscillator” circuit defined in the previous section, for example, depends on the previous output and the current input. Hence, we arrive at functions that are realized by *digital systems* over time:

*Definition 5.7: (Digital Systems)* A discrete function  $f(x,t)$  with domain  $D \times Time$  and range  $R \times Time$  ( $D$  and  $R$  finite) is realized by a system  $S$  (describable in a theory  $P$ ) if and only if the following conditions hold:

1. There exists a (syntactic) isomorphic embedding  $I$  from disjoint intervals of the “input domain” of  $S$  (where the interval length is  $>2 \cdot \epsilon_{data}$  for some  $\epsilon_{data}$  dependent on  $S$ ) to  $D$
2. There exists a (syntactic) isomorphic embedding  $O$  from disjoint intervals of the “output domain” of  $S$  (where the interval length is  $>2 \cdot \epsilon_{data}$  for some  $\epsilon_{data}$  dependent on  $S$ ) to  $R$
3. There exists a (syntactic) isomorphic mapping  $T$  from disjoint intervals of *RealTime* (where the interval length is  $>2 \cdot \epsilon_{time}$  for some  $\epsilon_{time}$  dependent on  $S$ ) to *Time*
4. There exists a function  $F$  that describes the physical property (=behavior) of  $S$  for the given input-output properties over time (i.e.,  $F$  is a mapping from the “input domain”  $\times RealTime$  of  $S$  to its “output domain”  $\times RealTime$  described in the language

and by the laws of  $P$ ) such for all  $t \in \text{Time}$  and all  $x \in D$  the following holds: if  $F(\Gamma^1(x), T^1(t)) = \langle X, Z \rangle$ , then  $\langle X, Z \rangle = \langle O^1(x'), T^1(t') \rangle$  (where  $f(x, t) = \langle x', t' \rangle$ ).<sup>100</sup>

Digital systems (according to definition 5.7) are physical systems that realize certain, discrete functions with finitely many input and output values (depending on their physical make-up). Only a finite set of discrete magnitudes corresponding to (some of the) input-output values together with the temporal order of applying input at a certain time and receiving output at some later time is retained (from the physical description of their functionality). The specifics (of *how* these finite values relate to each other or what the duration between any two points in time is) are ignored; hence the information about them is lost. It is given up in exchange for a purely theoretical treatment of these systems: *the physical (engineering) level of description is left in favor of a mathematical level of description*, where one can study functions realized by digital systems and their properties without recourse to the concrete using tools from mathematics and formal logic.

The system  $S_{10}$  from above (which realizes  $\text{AND}_3$  as well as  $\text{AND}_{10}$ ) may elucidate this shift from the concrete into the abstract realm. Only three out of the ten values  $S_{10}$  was designed for, found their use in  $\text{AND}_3$ , and their choice was merely guided by practical reasons; nothing theoretical forced it. In a sense, it was the set of all possible inputs (and outputs, i.e., the practical constraints) *together* with the function  $E$  that determined whether the gate functioned as a ternary or as a decimal AND-gate. The input

---

<sup>100</sup> For those circuits whose output at  $t+1$  only depends on their input at  $t$ , one can generally ignore the additional time place in the function and just consider finite one-place functions (e.g., the way AND-gates are normally defined), as

1, for example, could be either mapped onto  $\Pi)=[3,5]$  or onto  $I^*1)=I(E1))=I^*(9)=[95,105]$  Volts, depending on the input encoding that is used. In other words, the “embedding”  $E$  suggests that  $AND_3$  can be “realized” by  $AND_{10}$ , where this kind of “realizing” can be spelled out as “there exist embeddings  $I_{3,10}$  and  $O_{3,10}$  (in the above case  $E$ ) between the domains and ranges of  $AND_3$  and  $AND_{10}$ , respectively, such that for all  $x$  and  $y$  in the domain of  $AND_3$  the following holds:  $O_{3,10}^{-1}(AND_{10}(I_{3,10}(x),I_{3,10}(y)))=AND_3(x, y)$ , so “comprising” would be a better term. The fact that  $I^*$  is composed of  $I$  and  $E$  (i.e.,  $I^*=E \circ I$  for some  $E$ ) shows *that and how* one can determine, *on purely mathematical grounds*, that  $S_{10}$  realizes  $AND_3$ . It even provides a strategy to determine the class of functions that a physical system  $S$  realizes under definition 5.7: take the function  $f$  that  $S$  realizes according to definition 5.6. Then all possible subsets of that function (up to renaming of the elements) will be realized by  $S$  according to definition 5.7. Notice that instead of considering the relationship between a function and a physical system, the relationship between two functions becomes the focus of attention, as captured in the following definition:

*Definition 5.8:* A finite function  $f(x)$  with domain  $D_f$  and range  $R_f$  is *comprised by* discrete, finite function  $g(x)$  with domain  $D_g$  and range  $R_g$  if and only if the following conditions hold:

1. There exists a bijective mapping  $I$  from  $D_f$  into a subset of  $D_g$
2. There exists a bijective mapping  $O$  from  $R_f$  into a subset of  $R_g$

---

long as it is understood that the output of the circuit occurs one “time step” after the input has been applied.



3. For all  $x \in D_f$ :  $g(f(x)) = O(f(x))$ .

Using this definition, one can show that  $S_{10}$  realizes every  $\text{AND}_n$  for  $n < 10$  by proving that every  $\text{AND}_n$  for  $n < 10$  is comprised by  $\text{AND}_{10}$ , and in consequence, that  $S_{10}$  realizes the function  $\text{NOT}_n(\text{OR}_n(\text{NOT}_n(x), \text{NOT}_n(y)))$  for all  $n \leq 10$  (once its identity with  $\text{AND}_n$  is proved for all  $n$ ). The same technique can, furthermore, be used to prove that  $S_{10}$  will *not* realize any  $\text{AND}_n$  for  $n > 10$  (assuming the given physical description of  $S_{10}$ ). And this kind of negative result could never be obtained within the physical theory itself!

### 5.7 Digital Systems and the Theory of Computation

Digitality allowed us to abstract from physical realizations and to talk about “abstract” digital systems such as the “AND-gate” (knowing at the same time, though, how these digital systems can be physically realized). Whereas practical reasons have served their purpose to push us from below upwards to digitality, they can no longer influence the development, once the realm of the physical is left. This is where the mathematical formalism takes over completely, and logical possibility replace physical possibility. Therefore, theoretical arguments are needed to single out certain digital systems that theoretically “more interesting” than others.

One, if not the main theoretical interest regarding any physical system that can be used for computation is to find out what exactly it can compute (note the modality!). This was obvious with single logic gates, but quickly becomes quite involved if arbitrary combinations of such gates are allowed. In short, what we are interested in is the class of functions realized by digital systems.

According to definition 5.7, functions realized by digital systems have a composite domain, consisting of a finite and an infinite part (the finite being  $D$ —the set of “digits— and the infinite being *Time*). While for some systems it will be possible to ignore *Time*, as it does not influence the input-output relation of the digits (i.e., the first component), most systems that have a “memory” (i.e., “internal” states) will exhibit behavior that does depend on their history, i.e., on the kinds of digits presented at particular (previous) times. For those systems it is crucial to pay close attention not only to single digits, but rather to *the sequences of digits*, as the output of the system is determined by and dependent upon them.

The revised standard account of implementation in chapter 2 has already introduced the notions “input sequence” and “output sequence” to account for the fact that every computational physical system has only finitely many relevant computational states. This notion, of course, has to be tailored to the peculiarities of each particular system under scrutiny, as it will depend on the system where sequences start and end (something that cannot be decided theoretically but only by looking at the physical description of the system). By introducing the notion of sequence of (input/output) states, however, the revised standard account has tacitly made a crucial transition: it did not consider the function realized by the system any longer, but rather another function, which was obtained from the function realized by the system by looking at sequences of inputs and outputs. More formally, this transition can be presented as follows: let  $f$  be the function realized by a physical system  $S$ , and let  $D$  be the finite domain of  $f$ ,  $R$  be the finite range

of  $f$  and  $Time$  be the infinite set of discrete (abstract) times. Then the function  $f^\omega$  over input-output sequences generated by  $f$  can be defined as:

$f^\omega(\{i_1, i_2, \dots, i_n\}) = (\{o_2, o_3, \dots, o_{n+1}\})$  iff  $f(i_1) = o_2$  and  $f(i_2) = o_3$  and ... and  $f(i_n) = o_{n+1}$ , where indices denote the first  $n+1$  elements in  $Times$  and  $i_k \in D$  and  $o_m \in D$  for  $0 < i \leq n$ ,  $1 < m \leq n+1$  (" $i_n$ ", thus, denotes the input at time  $n$ ).

Note that  $f^\omega$  has infinite domain and range, since it maps arbitrary long sequences onto other sequences, and that  $f^\omega$  does not have a time parameter involved anymore (which was used to define the "sequences").

Since the function  $f^\omega$  (obtained from the function realized by a digital system) can be viewed as a mapping from strings of a finite alphabet to strings of another finite alphabet, it is only natural to ask where it figures in the Chomsky hierarchy. And it should not come as a surprise that the answer is: it depends! To be precise, it depends on the physical theory (that is used to describe the physical system). It is the physical theory that will delimit the "computational possibilities" of physical systems by its description of possible input-output mappings. If the physical theory is circuit theory (and the "primitive objects" are thus "resistors", "capacitors", "transistors", etc.), for example, and we are assuming a fixed system (i.e., a system that is not modified over time, such as adaptive and/or self-reproductive systems) that cannot exchange information with any other system except taking in inputs and producing outputs, then  $f$  is going to be a function that can be computed by a FSA (i.e.,  $f$  will be *regular* as the class of regular

languages is coextensive with the class of functions computed by a finite automaton)—this is exactly what one would expect from a finite system with a fixed bound on its resources, a bound that does not depend on input magnitude. By the same token, every function computed by a FSA can be implemented by a physical system that can be described by circuit theory.

To see that every function realized by a digital system (according to circuit theory) can be computed by a FSA, suppose that  $n$  is the number of bits of the system's internal memory (bits are used here to measure the number of different distinguishable "internal states" the system can reliably be in—"distinguishable" and "reliable" are to be understood in the sense of the previous discussion). This number is a constant (for a non-changing system), because the number of digits of the system is finite, the number of parts that could be in different states is finite and there are only finitely many parts. Thus, there are only finitely many different state transitions that the system could exhibit, therefore there exists a FSA that "mirrors" the state-transitional structure of the system. Actually, this FSA is merely an abstraction over the physical peculiarities of states and transitions which retains only the minimum necessary to describe them.

For the opposite direction, to see that every function computed by a "reasonably-sized" FSA can be realized by a digital system described by circuit theory, just represent the FSA as a two-dimensional matrix ("lookup table"), where states are row indices, input characters column indices, and matrix elements contain the "next state" for the respective

indices.<sup>101</sup> This matrix, in turn, can be implemented using digital circuits (such as memories, counters, decoders, etc.).

On the other hand, if an unbounded resource is assumed (e.g., a disk drive with an unbounded capacity regardless of its physical realization or a network connections that ties the finite system to an “unbounded resource”—an arbitrarily extendable memory, for example), then one can see that physical systems described by circuit theory realize recursive functions. One way of arguing this is to separate a Turing machine  $M$  into its FSA part  $F$  (that is, its finite control) and its tape part, which will then be substituted by a some sort of (unbounded) “intelligent” memory (the details of how this memory works and stores information are not important). The tape head will be substituted by a network connection between the FSA and the memory: the FSA sends two characters over the network (one from the alphabet  $\Sigma$  and one from  $\{L,R,S\}$ , none of which are in  $\Sigma$ ), and receives one character in  $\Sigma$  back from the network. Depending on this character, the FSA changes state and sends another two characters, unless it has reached a final state. The operations of the intelligent memory are simple: the first character it receives will be stored at the current memory location (which it maintains as current). The second character will be interpreted as a command to change the current memory location: in case of ‘L’ it will switch to the left neighboring location, in case of ‘R’ it will switch to the right neighboring location, in case of ‘S’ it will stay at the current location. This assumes that the memory is organized in some linear fashion (but nothing crucial hinges upon it as long as memory locations can be addressed relatively).

---

<sup>101</sup> It is crucial to add the restriction “reasonably-sized”, because otherwise the statement would plainly be false: an

It is easy to see that *F cum network memory* performs the same operations as *M*: for every state transition  $\langle p, a, q, b, m \rangle$  of *M* (meaning that *M* being in state *p* upon reading character *a* transits into state *q* writing character *b* and performing the tape head movement *m*), *F* being in state *p* receiving *a* from the network, transits into state *p* writing first *b* and then *m* to the network. The *network cum memory* (i.e., the unbounded resource) makes sure that the characters are stored and retrieved in the right order.

The FSA part *F*, being an FSA, as well as its ability to send characters to and receive them from the network can be built (or at least defined within circuit theory). However, this assumption of an unbounded resource is necessarily an idealization (similar to Turing's assumption of the idealized, abstract human computer). So the class of recursive functions, and thus those computed by Turing machines are an upper bound for the class of functions of digital systems *as described by circuit theory*.

But what about other physical theories? Are there physical theories in which we can describe systems that do not give rise to recursive functions? And what would those systems look like? Siegelman and Sontag (1992), for example, show that neural networks with rational weights can compute exactly the class of recursive functions. If one relaxes the requirement and allows one (!) real number as a weight, then any function over the natural numbers can be computed by such a neural network. The argument is quite simple: assume an enumeration of all "argument(s)-value"-tuples of an *n*-ary function *f* over the natural numbers and let  $p = \langle k_1, k_2, \dots, k_n, m \rangle$  be such a tuple. Let *r* be the string that can be obtained from concatenating the representations over  $\{0, 1, 2, 3\}$  of all

---

automaton with more states than particles, for example, in the universe could certainly not be built.

such tuples according to their enumeration, where the representation over  $\{0,1,2,3\}$  of a tuple is simply the string

$$\langle \text{binary expansion of } k_1 \rangle 2 \langle \text{binary expansion of } k_2 \rangle 2 \dots \langle \text{binary expansion of } m \rangle 3$$

The so obtained infinite string  $r$  can be interpreted as the base 4 expansion of a *Real*. Thus, the so-obtained *Real*  $[r]$  encodes the function  $f$ . Every  $n+1$ -tuple of  $f$  can obviously be “looked up” in a finite amount of time (just search the string from the beginning for its arguments represented in binary and separated by 2). Even partial functions can be represented if one agrees upon their representation (e.g., after the last argument ‘3’ is added right away instead of the representation of a value for the previous arguments). It follows, in particular, that if the function so encoded is non-computable, the so-obtained *Real* will not be computable. If it is possible to store physical manifestations of such a *Real* (whatever they might be) and use them for computations (such as the ones in the network), then a physical system that incorporated such a *Real* could be used to solve decision problems that cannot be solved by any Turing machine.

While it seems very unlikely from the point of view of today’s physics that realizations of “non-computable Reals” can be found in nature (let alone that they can be somehow manipulated so as to become an intrinsic part of a “Super Turing computer”), it is important to notice that the definition of digital system *does not preclude* such systems. For example, it allows for “oracle systems”, i.e., systems that have a subsystem that can somehow answer some questions that Turing machines cannot answer (how exactly they perform their computations does not matter) (see Copeland, 1998b).

This generality of the definition of “digital system” is not a disadvantage by any means. Alas, it shows that the notion of “digital system” does not automatically constrain the class of functions that can be realized by physical systems to recursive functions (or equivalently Turing machine computable functions); rather the class of functions that can be realized by a class of physical systems is delimited by the physical theory in which these physical systems are described. This insight is an essential part of the suggested approach to physical realization of functions: if there is anything to the term “physical realization” that should be kept from its ordinary understanding, then it is the fact that physics constrains ultimately what can be realized! Computation (in the sense of “computable function”) is certainly no exception.



## Chapter 6:

### Conclusions and Prospects

At this point I will interrupt the exposition to reconsider the notions of computation and implementation in the light of the notion “digital system” developed in the last chapter. The main goal of that chapter was to show how one could bridge the gap between the concrete and the abstract, between physical systems and the computations they implement, without falling prey to the various objections analyzed earlier. In order to do that the notional pair “computation-implementation” had to make way for the notion “realization of a function”, i.e., what it means for a physical system  $S$  described in a (physical) theory  $P$  to realize a function  $f$ . Beginning with a definition of “realization of a function” that was closely tied to the theory describing the system, more and more constraints were incorporated leading to more and more restricted classes of physical systems (and thus restricted functions that are realized by those systems). Stepwise abstraction over physical dimensions and magnitudes eventually led to *digitality*, to the total abstraction over physical realizations. It enabled us to talk about functions such as AND-gates, registers, and even von Neuman computers, independent of their physical realization, while knowing at the same time, how these functions could eventually be realized physically in *digital systems*.<sup>102</sup>

---

<sup>102</sup> Note that the notion of physical realization underlying the notion of digital system is neither a syntactic, nor a semantic, but a purely physical/mathematical notion obtained from an abstraction process over physical dimensions. This notion still resides within the realm of the physical theory within which we performed these abstractions!

Digital systems can be viewed as physical systems that admit of a special description of their behavior: a description that only needs an order relation (to model temporal succession) and a set of distinct, discrete entities (the digits). It is a virtue of digital systems that, because they contain these discrete entities, their temporal evolution consists of successions of (different) combinations of digits, in other words, of transformations between patterns. Not only is it well-known that formal grammars can capture transformations of patterns (i.e., grammars consisting of a “start state” and finitely many rules describing the next state), but also that these transformations lend themselves to algorithmic descriptions. Hence, some digital systems can be described by algorithms, namely those which realize *recursive* functions.<sup>103</sup> This, in turn, implies that the functions they realize are algorithmically expressible. Others, whose mapping between input and output sequences cannot be described by finite grammars, can nevertheless be described by listing all their countably many input-output-sequence pairs. In either case, digital system can be described *syntactically*.

According to Searle, however, this syntactic specifiability of digital systems is a fundamental weakness of the computationalist program. Even placing further restrictions on the notion of (Turing-)computation will be of no help in solving the “implementation problem”,

“because the deep problem is that syntax is essentially an observer-relative notion. The multiple realizability of computationally equivalent processes in different physical media

---

<sup>103</sup> I do not know of any realizable digital system that realizes a non-recursive function. Yet, the definition of digital system is indifferent on this issue, as already mentioned earlier; constraints are solely imposed by the physical description of the system.

is not just a sign that the processes are abstract, but that they are not intrinsic to the system at all. They depend on an interpretation from the outside.” (Searle 92, p. 209)

Reflecting on the development of the notion of digital system as lined out in the last chapter, Searle’s conclusion comes as a surprise: it *does* seem that computationally equivalent processes are intrinsic to the physical system. It might well be the case that this property cannot be detected from a “rich physical description” (which contains all the physical dimensions and qualities of these dimensions of the system), but can only be seen at a more abstract level, which has dispensed with these properties. In particular, whether a system is a digital system *solely depends* on its physical description and *not* on any judgement by *outside observers*. The syntactic characterization of a digital system is *not observer-relative*: if one agrees that a given physical theory *P* does describe the behavior of a physical system *S* adequately, then one is also committed to whatever the result of the abstraction process predicates about the physical system with respect to the notion of digital system (derived from that very physical theory), i.e., to whether *S* is a digital system with respect to *P*. Understanding digital systems (and consequently digital computers) as systems that realize a certain class of functions, Searle’s claim that

“there is no way you could discover that something is intrinsically a digital computer because the characterization of it as a digital computer is always relative to an observer who assigns syntactical interpretation to the purely physical features of the system.”

(Searle, p. 210)

does not seem justified any more. Of course, this is only true under the assumption that the system is adequately described by the given physical theory. This, interestingly, puts the problem back in the ballpark of physics: if anything is observer-relative at all, it

seems to be what counts as an adequate “physical description”! Once agreement is reached on how to physically cast a system’s behavior, its computational guise will also be fixed.

It is also worth pointing out at this place that the abstraction process introduced in the previous chapter is not intrinsic to physical theories *per se*, but rather to the “mathematical part” of the physical theory. As a consequence, it will be the same for different physical theories that use the same mathematical underpinnings (i.e., dynamical systems theory). There might be minor modifications resulting from descriptions of physical systems that use rational numbers instead of real numbers, or some of the abstraction steps could be skipped altogether if the physical description of the system already used integers to measure physical magnitudes. Only if the mathematical apparatus differs significantly, the sequence of abstraction steps (that eventually lead to digital systems) will have to be adapted.<sup>104</sup>

The approach to “physical realization” developed in the last chapter differs significantly from both SV (the semantic views) and CV (the state-to-state correspondence views) in at least one major respect: it does not require a notion of physical state, but determines *directly* the function realized by a physical system by extracting this function from the physical description of the system. Furthermore, it neither has to assume a particular computational formalism (to which the physical system exhibits a state-to-state correspondence) nor a particular formally specified computational architecture (which can be interpreted over models generated from “labeled physical

systems”), but can be related to computations described by any computational formalism (such as FSAs, TMs, PASCAL programs, cellular automata, etc.) *via* the function that these computations give rise to (see figure 6.1). This suggests the following general definition of when a physical system implements a computation:

*Definition 6.1:* A physical system  $S$  implements a computation  $C$  iff there exists a function  $f$  such that  $S$  realizes  $f$  and  $C$  computes  $f$ .<sup>105</sup>

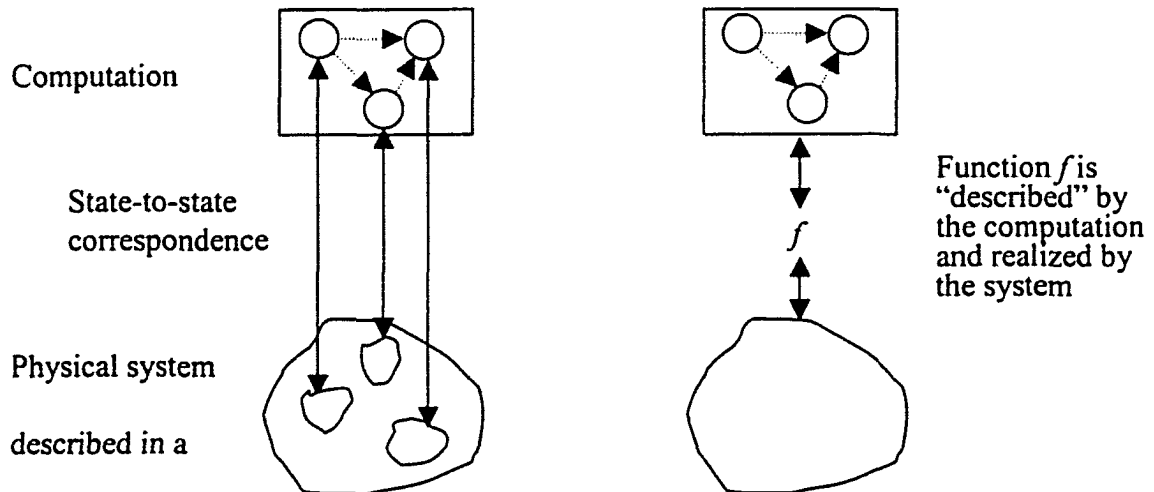
Note that whereas SV takes implementation to be a relation between formal systems and standard models of these systems, it is implied by the above that implementation is better understood as a relation between *two formal systems* (or, more restrictively, two parts of a formal theory): the formal theory that is used to describe the behavior of a physical system (the physical theory) and the formal theory that is used to specify computations (e.g., automata theory)—the latter is linked to the former *via* the mathematical concept of function.<sup>106</sup>

---

<sup>104</sup>It might have to be altered completely if the underlying mathematics has changed completely. However, in such a case, the notions of computation (and with it notions of implementation) would probably not be spared from a complete revision either, hence I would think this restriction should not matter much.

<sup>105</sup>Note that it is part of the computational formalism, in which  $C$  is expressed, to define what function the computation  $C$  “computes”.

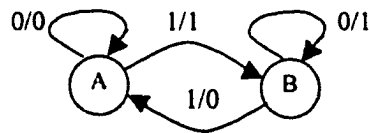
<sup>106</sup>There is a similarity between my approach and what Copeland probably had in mind: Copeland required that the labeling be systematic—in my approach it is the abstraction process (over physical dimensions guided by practical constraints) that is systematic. Note, however, that while Copeland’s requirement allows for different systematic labelings, my approach fixes the sequence of abstraction steps to be taken (which eventually leave the mathematical skeleton of the physical input-output function restricted to practical constraints).



**Figure 6.1** The difference between the state-to-state correspondence view (left) and the “functional realization model” (right): in the latter, functions serve as mediators between physical systems and computations.

The suggested approach works for arbitrary levels of description of physical systems as well as different computational formalisms (as long as they provide a criterion of how to specify the function they compute). It can be even used to define/find physical states that do somehow correspond to computational ones for systems that are now to implement a certain computation, but where physical states are not given. For example, take the oscillator circuit (described in circuit theory) from the previous chapter (figure 5.2), which is functionally specified as  $g(x_t, t+1) = f(g(x_t, t), x_t, t+1)$  and  $g(x, 0) = 0$  (where  $x_t$  is the input to the system at time  $t$ ). Compare this to the 2-state automaton in Figure 6.2, which also “computes” the same function  $g$  (where  $x_t$  is the  $t$ -th input character). According to definition 6.1., the oscillator circuit *implements* the above automaton (by virtue of “realizing” the function  $g$ , which is “computed” by the automaton). Yet, there are no obvious candidates for “inner states” in the oscillator circuit that could be set in

correspondence to the automata states A and B (as would be required by the state-to-state correspondence view).



**Figure 6.2** An automaton, which computes the function realized by the oscillator circuit from the previous chapter.

The fact, however, that the oscillator circuit implements the automaton might help to single out (quite abstract) states in the oscillator circuit that could be set in correspondence to the automata states. First, consider a natural interpretation of the “inner” states of this automaton, which views them as corresponding to whether the automaton has received an even or odd number of 1s in its inputs  $x_0, x_1, \dots, x_{t-1}$  (where  $x_t$  is the current input): the automaton will be in state A if the number of 1s received was even, otherwise it will be in state B. In general, such an interpretation of inner states can be quite misleading as one is tempted to look for physical counterparts (e.g., a “bit” that keeps track of the current state), which might not exist. In this case, the recurrent output line that feeds into the second input line in the XOR gate plays, indeed, the role of a one-bit memory. Thus, if (the only) inner states are defined to be the voltages on this line, then low voltages can be set in correspondence to state A and high voltages to state B (the same has to be done for inputs and outputs, respectively). Using a state-to-state correspondence notion of implementation such as Chalmers’, one can now see that the oscillator circuit implements the automaton (under this notion of implementation).

Despite the above success with the simple oscillator circuit, I am not sure that this strategy will work for more complex (and, in particular, natural) systems. We might fail to choose the right level of abstraction, at which a correspondence could be established, or we might fail to find the right kinds of physical states (that can be set in correspondence with computational ones), or we might not be able to select the computationally relevant physical states. Whatever the reason may be, it will still be possible to say that a certain physical system implements a computation if definition 6.1 is satisfied. And looking at the computational description of that physical system might still be instructive, if not helpful in understanding *how a system of that kind* could be organized, even if the computation does not reflect the causal organization of *that particular system*. In fact, I venture to say that this kind of reasoning is exactly what ultimately motivated AI researchers in the beginning (and maybe still does) to develop machines that have the same functional capabilities as we humans, yet a quite different “physiology”.

Returning to the assessment of the here-developed approach to implementation, how does it cope with the kinds of objections raised against other accounts of implementation (e.g., the intuitive account of implementation), in particular, Putnam’s and Searle’s arguments? As far as “Putnam-like constructions” (including those of the Slicing Theorems)<sup>107</sup> are concerned, they are simply not applicable here, since neither state-types nor mappings have to be defined: the only mappings involved, i.e., the syntactic isomorphisms (for input and output), cannot be defined arbitrarily, but are derived in a



clearly specified way from the physical description of the system; if they are not appropriate, then neither is the physical description! Therefore, these kinds of constructions do not pose a threat.

The same is true for Searle's "wall computer" as well. It seems extremely unlikely to me that the physical description of the "input-output behavior" (assuming that input and output regions have been chosen) of a wall will permit an abstraction (at the level of digital systems) such that the resulting function is the function computed by the Wordstar program. For a definite answer, however, one would have to fix a certain class of walls, inputs, and outputs, etc. and consider their physical description (at a certain level, e.g., the level of molecules).

To summarize the most pertinent properties of digital systems: every digital system has a finite set of input and a finite set of output digits. Its input-output relation is a function, which maps digits onto digits over time. *How* the system performs this function is of no concern (at this level of abstraction), what counts is *what* and *that* the system performs it. Furthermore, nothing has to be known about the internal structure of the system to be able to use it (so no notion of "internal state" is required). If the system is to be used in a practical setting, only the physical nature of its inputs and outputs is of importance, because that will determine how these systems can be connected to other systems (i.e., measurement instruments, other computational systems, etc.) Yet, for engineers, who have to construct, change and maintain complex (digital) systems, it is of great importance to be able to look inside the system and *divide it into/built it from*

---

<sup>107</sup> I assert that under definition 6.1 the switch system will compute a very simple function:  $f(\text{switch-on})=\text{light-on}$ ,  $f(\text{switch-off})=\text{light-off}$ . However, I have not gone through the details of the physical description of the switch-system

smaller parts. The smallest of these parts, however, will again be viewed as *black boxes* and it will be satisfactory for all the engineering purposes to treat them that way as long as it is known what function they realize (what the nature of the inputs and outputs is, etc.). This is were the property of digital systems that any assembly of digital systems (no matter how the inputs and outputs are connected, if they can be connect at all according to the nature of the respective inputs and outputs) is again a digital system comes in handy. Complex digital systems can thus be assembled from smaller (digital) ones. In this way, engineering can deal with the otherwise intractable complexity of large digital systems (e.g., such as computers). Yet, once they are built and they are ready for use (i.e., for any practical purposes), they can be regarded as a mere black boxes, where not inner states, parts or internal connections, but only the mappings between finite sets of inputs and finite sets of outputs matter, i.e., *the functions they realize*.

Digital systems can be viewed as *implementing* the functions they realize, and these functions, in turn, as the *atomic computations* implemented by digital systems. The qualifier “atomic” is certainly necessary here, since for most people there is more to computation than just realizing a certain function, and rightfully so. Cummins (1989), for example, when he speaks of physical systems “satisfying a function” also regards these functions as atomic operations which can, in a more complex system, be used to implement an algorithm. Some people (e.g., Searle) might find even such a restricted notion too permissive (e.g., because planet systems or digestion do realize certain functions), others (e.g., Copeland) might disagree with the fact that a notion of physical causality underlies this notion of computation (e.g., virtual machines or even the

---

to derive this function formally.

mathematical concept of a Turing machine are excluded that way), and yet others (e.g., Shagrir, Smith) might find the whole approach fruitless, because it is, contrary to their convictions about how to tackle the notion of computation, intrinsically non-semantic. Note, however, that definition 6.1 is silent about the intensional relationship between two computational formalisms that compute the same function  $f$ , call them  $C_1$  and  $C_2$ , and as a consequence about the question, whether semantics is necessary to understand this relationship. While computing “the same function”, according to definition 6.1, implies that both computations are implemented by the same class of physical systems, this identification is restricted *extensionally* to the class of physical realizers. Nothing is said (and should not be said) about the intensional relationship of  $C_1$  and  $C_2$ , since otherwise hidden claims about the semantic interpretation of physical theories would enter the picture and to keep them out was one of motivations for definition 6.1 in the first place.

Regardless of whether one wants to speak of computation already or mere “atomic operation” (or “step-satisfaction” to use Cummins’ term), it seems clear that digitality, i.e., the discreteness in space and time, is a prerequisite of physical systems to admit of algorithmic descriptions. Every (reasonably-sized) finite state automaton (FSA)—as was pointed out earlier—can be described by a function realized by a digital system (described in circuit theory). Even if some might argue that FSAs are neither algorithms nor models of algorithms, their behavior can be described algorithmically. Thus, as a consequence, the behavior of digital systems implementing FSAs can be described algorithmically, too.

If a physical system does not support digits, it is not clear how a computational description could be imposed on it without rendering the account arbitrary and, hence,

vacuous (again, the truth of this claim hinges on the definition of “computation” and is largely a terminological problem). Granted that it is possible to define “gerrymandered digits” by *fiat* for every physical system (e.g., simply stipulate the property of being a digit in that system as a certain physical condition that obtains for a certain amount of time whenever, possibly only once). Using digits so defined one can view almost every physical system as “digital system”, but this notion of digitality has nothing in common with the previous notion extracted from the physical laws describing the behavior of that system nor, as far as I can see, will it do any useful practical work. The question whether or not a system qualifies as digital according to the previously developed notion has an objective, observer-independent answer that solely depends on the nature of the physical theory in which the system and its behavior are described. It is defined once (and for all) for all potential physical systems and does not depend on *ad hoc* assumptions made of and definitions introduced for particular systems.

Beyond clarifying the relation between the notions of computation and implementation, digital systems have many more interesting properties that I have not been able to address. Two virtues of digital systems that are of crucial concern to “symbol manipulation” and thus to cognitive science, for example, are *compositionality* and *representation*. Digits, by virtue of their distinct form and shape, can be combined to form complex “strings of digits” (i.e., with a finite set of digits one can potentially form an infinite set of expressions). In binary digital computers, for example, 8 wires (each of which, being a “bit”, can assume one of the binary values 0 or 1) are combined to form complex bit patterns. So it is possible to generate 65536 different patterns with 16 two-valued wires instead of having only one wire with 65536 values (which would make

design issues more delicate).<sup>108</sup> “Strings of digits”, in turn, can be used as representations (e.g., to represent numbers). The binary number system (which should really be called “binary numeral system”) is but one example of such a representational scheme. And, of course, representations can represent other representations, which themselves represent, and so forth (take, for example, letters which are represented by ASCII codes which, in turn, are represent by 128 different bit patterns). Once representations are introduced, there is almost no limit as to what can be represented and what counts as a representation. In particular, representations are a necessary prerequisite for the notion of “program” and in turn for programmable computers.

Digits cannot only be used to represent “data” (on which operations are to be performed), but also to represent (i.e., denote) *operations*. This is possible for two reasons: first and foremost, because an appropriate machine architecture can be provided (e.g., the well-known von Neumann machine)<sup>109</sup>, and secondly, because any system—given an appropriate architecture—can reliably detect what function is denoted by a certain digit (or a combination of digitis) because of the defining characteristics of digits (see also Haugeland, 1982).

Systems that use digits to represent internal operations can be designed in a way that permits them to select certain operations from a “pool” of basic operations and apply them to data. The sequence in which they are applied gives rise to a more complex operation (which is defined from the basic operation by virtue of exactly this sequence).

---

<sup>108</sup> See Agre (1997, ch. 4) for a description of the role of design issues in the development of digital computers.

<sup>109</sup> What I mean here is CPU, ALU, memory, clock, etc., in short, all the essential parts of a modern computer. I cannot go into details here, but it is an interesting endeavor in its own right to investigate the necessary structural features of digital machines to allow for universality...

If the class of functions that can be defined from the basic pool of operations using machine dependent ways of combining them (such as composition, recursion, etc.) is the class of all functions that can be realized by digital systems (with respect to a given physical theory), then the particular system can be called “universal” (with respect to a given physical theory). If this sequence is, furthermore, a system parameter (i.e., can be influenced), then the system can be said to be “programmable”. Both ideas combined give rise to the idea of “universally programmable system” (which can be compared to a universal Turing machine, for example).

The above remarks are, of course, mere hints at the different directions into which the previous story could develop: one possible extension could analyze the different possibilities to use digits within a system to denote parts of that very system and attempt a reconstruction of a “precursor notion of representation” from these simple, causally connected relationships between parts and patterns, also showing how these relationships can be utilized to get more “decoupled representations”. Another could attempt to develop the notion of “universally programmable digital system” and relate it to the notion of “universal Turing machine”, studying the abstractions and idealizations involved in such a task. Or the whole discussion could be broadened and extended to include the relationship of “constitution” in addition to the input-output system’s behavior. I believe it would be worthwhile to examine all of these possible extensions (for different reasons). What would be common to all these projects is the idea that computation and physics are closely tied together. The “gap”, that the notion of implementation is supposed to bridge, should in my view not be located between the physical description of a system and its computational description, but left where it has

been in the first place: between our physical descriptions (i.e., in the end perception) of the world and the world itself.<sup>110</sup>

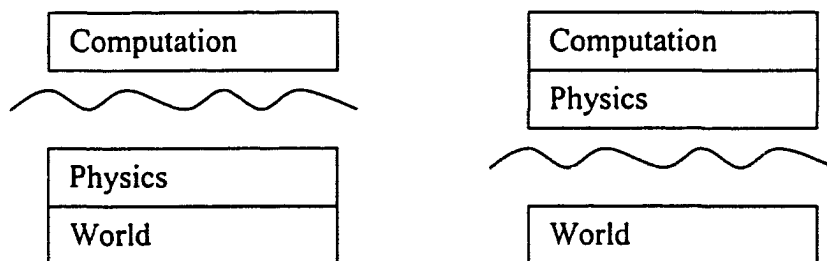


Figure 6.3 Relocating the alleged gap between computations and implementations as a gap between the world and our physical description of it.

This shift places the “burden of proof”—which theory is the best, most adequate, etc.—first on the shoulders of physics, but eventually on practice. It is my conviction that whatever works best, can be used technologically, can explain most, etc. will ultimately find the broadest acceptance. This is why the whole enterprise was guided almost exclusively by practical considerations: because I wanted to focus on functions realized by systems that we can *recognize* and *use* (eventually as *computers*). Computing is an activity that be recognized as such, initiated, performed, analyzed, etc. It is an activity that is employed for various purposes. Searle’s wall, for example, might realize a very complicated function at the level of fields, but we are not able (or at least not now) to utilize it for computation. That is not to say that only systems computing “by virtue of their digitality” are fit for computation. Various analog systems (and their physical

---

<sup>110</sup> Note that this figure is intended to be silent about whether one wants to place gap between “physics” and the “world” in the left graph in addition to the one between “computation” and “physics”. If one is so inclined, I would then speak of “closing the gap between computation and physics” rather than “relocating” it.

properties) have been applied for fast, reliable computations.<sup>111</sup> In the end, what kind of system is *useful* for computation is decided by *who is using it*: digital computers are built by us humans, because they are *useful* to us. Brains have evolved in animals and are obviously *of use* for them. Different kinds of currently not considered systems could be appropriate and applicable for computation. If “computation” is to mean “computing a function” and if this, in turn, is interpreted as “finding the value of the function for a given argument” (not necessarily “effectively”), then standard notions of computation are included as well as ones that do not rely on “manipulations of representations” (such as analog computations performed by VLSI chips, neural networks, etc.). As long as these systems accept digits as inputs and produce digits as outputs, they can be treated pragmatically as “computing black boxes” (not to be confused with NEXT’s “black computing box”)—how they achieve the computation does not matter!

In a way, every system (described at level  $L$ ) that realizes a function could be seen as a computer, namely a computer computing that very function. But most of those “computers” are not useful for us; because we have no influence on their inputs and outputs, we might not be able to measure them or even recognize them as such. Hence, these systems do not qualify as “computers” (in a practical sense), although they could still be of interest for cognitive science. It might well be the case that cognitive systems are best described at level  $L$  and that at this level we cannot produce the right kinds of inputs and outputs, design and assemble the right kinds of components, etc. (because of

---

<sup>111</sup> It does not matter *how* a system realizes a function, i.e., if it computes using representations or if the result is obtained by the laws of physics (see, e.g. Mills’ analog retina), as long as the system is *digital from the outside*: a digital system can be treated as a “black box” with digital inputs and outputs, the inner organization does not matter for computation.



technical problems). This result would be fatal for the artificial intelligence branch of cognitive science, since it would preclude the construction of *artificial* cognitive systems and very likely an understanding of cognitive systems in general. What makes us believe that this is not the case is that certain cognitive systems (e.g., human brains) can at some level be described as being digital (e.g., the symbol level).<sup>112</sup> This abstraction over the physical properties of human beings was essential to Turing's definition of the "ideal human computer" where human capacities to calculate were phrased in purely symbolic (i.e., digital) terms (token manipulation, rule application, etc.; see also Haugeland, 1996). If the description of the brain at this or at a lower level (and the functions realized at that level) is essential to human cognition remains an empirical issue. It necessarily affects CCM, since the class of functions realized by digital systems is exactly what is commonly taken to be the extension of "computable". Assuming that the brain can only be described adequately at a level lower than the digital one, there are two possibilities: either the notion of computation has to be changed to the class of functions realized by systems described at that level, then CCM is true, otherwise CCM is false. If, however, the "digital level of description" is adequate for the brain, then CCM is true. In any case, this is an empirical question that can only be decided by looking at the functions that (parts of) natural brains realize.

---

<sup>112</sup> To some extent brains can even at a neuronal level be described as digital system, see, for example, McCulloch, and later von Neumann...

## References

- Agre, P. E. (1997) *Computation and Human Experience*. Cambridge University Press.
- Barwise, J., and Moss, L. 1996. *Vicious Circles*. CSLI Lecture Notes, Cambridge University Press.
- Block, N. (1995) "The Mind as the Software of the Brain", in *An Invitation to Cognitive Science. 2nd Edition: Vol. 3*. (Osherson, Daniel N. And Smith, Edward E. eds.) Cambridge: MIT Press.
- Block, N. (1996) "What is Functionalism?" *The Encyclopedia of Philosophy Supplement*, Macmillan.
- Bridgeman, B. (1980), "Brains + Programs = Minds", reply to Searle, *Brain and Behavioral Sciences* 3.
- Chalmers, D. J. (1994) "On Implementing a Computation", *Minds and Machines* 4, 391—402.
- Chalmers, D. J. (1996) "Does a Rock Implement Every Finite-State Automaton?", *Synthese* 108, 310—333.
- Chalmers, D. J. (1997) "A computational Foundation for the Study of Cognition". (published on the internet)
- Chrisley, R. L. (1994) "Why Everything Doesn't Realize Every Computations", *Minds and Machines* 4, 391—402.
- Cleland, C. E. (1993) "Is the Church-Turing Thesis True?" *Minds and Machines* 3, 283-312.
- Copeland, B. J. (1996) "What is Computation?", *Synthese* 108, 403—420.
- Copeland, B. J. (1998a) "Turing's O-machines, Penrose, Searle, and the Brain", *Analysis* 58, 128-138.
- Copeland, B. J. (1998b) "Super Turing-Machines". *Complexity* 4 (October).
- Cummins, R. (1989) *Meaning and Mental Representation*, Cambridge, MA, MIT Press.

- Davis (1965),
- Demopoulos, W. (1987), "On Some Fundamental Distinctions of Computationalism", *Synthese* 70, 79—96.
- Denett, D. C. (1986) "The Logical Geograph of Computational Approaches: A View from the East Pole", *The Representation of Knowledge and Belief* (M. Brand and R.M.Harnish, eds.), University of Arizona Press.
- Dietrich, E. (1990) "Computationalism", *Social Epistemology*, vol 4, no 2, 135-154.
- Endicott, R. P. (1996) "Searle, Syntax, and Observer Relativity", *Canadian Journal of Philosophy*, v26, 101-122.
- Fodor, Jerry A. (1981) *RePresentations. Philosophical Essays on the Foundations of Cognitive Science*. Cambridge, MA, MIT Press.
- Gandy, R. (1980) "Church's Thesis and Principles for Mechanism". Proceedings of the Kleene Symposium (J. Barwise, H. J. Keisler and K. Kunen, eds.). New York: North-Holland Publishing Company.
- Gandy, R. (1988) "", in *The Universal Turing Machine: A Half-Century Survey*. Kammerer & Unverzagt: Berlin.
- Garfield, j. (1995) "Philosophy: Foundations of Cognitive Science", in *Cognitive Science: An Introduction*. 2nd edition. (Stillings, Neil A. et al.) Cambridge, MA: MIT Press.
- Gödel, K. (1958) "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes", *Dialectica* 12, 455-475.
- Haugeland, J. (1982) "Analog and Analog", *Mind, brain, and function*. Norman: University of Oklahoma Press.
- Haugeland, J. (1996) "What is Mind Design?" *Mind Design II*. Cambridge, Massachusetts: MIT Press.
- Hardcastle, V. (1995) "Computationalism", *Synthese* 105, 303—317.
- Herken, R. (1988) *The Universal Turing Machine: A Half-Century Survey*. Kammerer & Unverzagt: Berlin.

- Hodges (1988), “”, in *The Universal Turing Machine: A Half-Century Survey*. Kammerer & Unverzagt: Berlin.
- Hopcroft and, J. E. and Ullman, J. D. (1979) *Introduction to Automata Theory, Languages, and Computation*. Massachusetts: Addison-Wesley Publishing Company.
- Johnson-Laird, P. N. (1988) *The Computer and the Mind*. Cambridge, Massachusetts: Harvard University Press.
- Kearns, (1997) “Thinking Machines: Some Fundamental Confusions”, *Minds and Machines* 7, 269—287.
- Kim, J. (1996), *Philosophy of mind*, Westview.
- Kripke, S.A. (1981). *Wittgenstein on Rules and Private Language*. Oxford: Blackwell.
- Kutz (1998) “Mathematical models of dynamical physical systems” in the *Mechanical engineer’s handbook*, John Wiley & Sons, Inc.: New Jersey, ch. 27.
- MacLennan, B. J. (1994) “Words Lie in Our Way”, *Minds and Machines* 4, 421—437.
- McCulloch, S. W. and Pitts, W. H. (1943) “A Logical Calculus of the Ideas Immanent in Nervous Activity”. *Bulletin of Mathematical Biophysics*, Vol. 5, Chicago: University Press, 115—133.
- Melnyk, A (1996) “Searle’s Abstract Argument Against Strong AI”, *Synthese* 108, 391—419.
- Messiah, A. (1961) *Quantum Mechanics*, Vol. 1 & 2. Amsterdam, Netherlands: North-Holland.
- Osherson, Daniel N. And Smith, Edward E. eds. (1995) *An Invitation to Cognitive Science. 2nd Edition: Vol. 3*. Cambridge: MIT Press.
- Pratt, V. (1987) *Thinking Machines – The Evolution of Artificial Intelligence*. Oxford: Basil Blackwell.
- Port, R. and van Gelder, T. (1995) *Mind as Motion: Explorations in the Dynamics of Cognition*. MIT Press, Cambridge.

- Putnam, H. (1967) "Psychological Predicates". (Reprinted as "The Nature of Mental States". *The Nature of Mind*. D. Rosenthal (ed.) New York: Oxford University Press (1991).
- Putnam, H. (1988) *Representation and Reality*. Cambridge: MIT Press.
- Scheutz, M. (1997) "Facets of implementation" (unpublished manuscript)
- Scheutz, M. (1998) "Do Walls Compute After All? – Challenging Copeland's Solution to Searle's Theorem Against Strong AI". *Proceedings of the 9th Midwest AI and Cognitive Science Conference 1998*, AAAI Press.
- Scheutz, M. (1999) "When Physical Systems Realize Functions...". *Minds and Machines* 9,2, 161–196.
- Searle, J (1980) "Minds, Brains and Programs", *The Behavioral and Brain Sciences* 3, 417-424.
- Searle, J. (1984) *Minds, Brains and Science*. Cambridge, Massachusetts: Harvard University Press.
- Searle, J. (1990) "Is the Brain a Digital Computer?", *Proceedings and Addresses of the American Philosophical Association* 64, p. 21-36.
- Searle, J. (1992) *The Rediscovery of Mind*. Cambridge, Massachusetts: MIT Press.
- Searle, J. (1997) *The Mystery of Consciousness*, New York: New York Review.
- Shapiro, S. (1995) "Computationalism". *Minds and Machines* 5, 517—531.
- Slezak, P. and Albury, W. R. (1989) *Computer, Brains and Minds: Essays in Cognitive Science*. Dordrecht, Holland: Kluwer Academic Publishers.
- Smith, B. C. (1995) "The Middle Distance". (unpublished manuscript)
- Smith, B. C. (1996) *The Origin of Objects*. Cambridge, Massachusetts: MIT Press.
- Smith, B. C. (1998) *The Age of Significance*. (forthcoming)
- Stabler (1987), "Kripke on Functionalism and Automata", *Synthese* 70, 1—22.
- Sterelny, K. (1989) "Computational Functional Psychology: Problems and Prospects", in *Computer, Brains and Minds: Essays in Cognitive Science* (Slezak and Albury, eds.), Holland: Kluwer Academic Publishers, 71—93.

- Stillings, Neil A. et al. (1995) *Cognitive Science: An Introduction*. 2nd edition. Cambridge, MA: MIT Press.
- Turing, A. M. (1936) "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society, Series 2* 42, p. 230 –265.
- Turing, A.M. (1939) "Systems of Logic Based on Ordinals" *Proceedings of the London Mathematical Society* 45, 161-228.
- Turing, A. M. (1950) "Computing Machinery and Intelligence", *Mind* 59, 433-60.
- Van Gelder, T. (1995) "What Might Cognition Be, If Not Computation?". *Journal of Philosophy* 91, 345-381.
- Van Gelder, T. J. (1998) "The Dynamical Hypothesis in Cognitive Science", *The Behavioral and Brain Sciences* 1999 (forthcoming).
- Wang, Hao (1974) *From Mathematics to Philosophy*. New York: Humanities Press.
- Yorke, James A. et al. (1996) *Chaos: An Introduction to Dynamical Systems* (Textbooks in Mathematical Sciences): Springer Verlag.

# Matthias Scheutz

## Address:

512 E Angela Blvd.  
South Bend, IN 46617  
Work: (219) 631-8752  
Home: (219) 287-4719  
email:mscheutz@cse.nd.edu

## Work Address:

Department of  
Computer Science  
and Engineering  
University of Notre Dame  
Notre Dame, IN 46545

## Education

<b>USA</b>	<b>Indiana University</b> (Bloomington, IN) Joint Ph.D. in <b>Cognitive Science and Computer Science</b> <i>Dissertation:</i> The Missing Link: Implementation and Realization of Computations in Cognitive Science and Computer Science	Fall 94 - Summer 99
	<b>Indiana University</b> (Bloomington, IN) M.S., <b>Computer Science</b>	Fall 94 - Spring 96
<b>Austria</b>	<b>University of Vienna</b> (Vienna) Ph.D., <b>Philosophy</b> <i>Dissertation:</i> Ist das der Titel einer Dissertation? - Selbstreferenz neu analysiert" (engl. "Is this the Title of a Dissertation? - Self-reference revisited)	Fall 89 - Winter 94
	<b>Vienna University of Technology</b> (Vienna) Dipl.-Ing. (=M.S.), <b>Computer Science</b> <i>Thesis:</i> "Programmierung eines NC-Postprozessors mit integrierter Technologiedatenanpassung – Übersetzung von ANVIL5000 APT nach FANUC" (engl. "Programming of a NC-Postprocessor with Integrated Technology Data Adjustment - Translation from ANVIL5000 APT to FANUC)	Fall 86 - Spring 93
	<b>University of Vienna</b> (Vienna) Mag. rer. nat. (=M.S.), <b>Formal Logic</b> <i>Thesis:</i> Die Grundlagen der klassischen Aussagenlogik mit unären Temporaloperatoren" (engl. The Foundations of Classical Propositional Logics with Unary Temporal Operators)"	Fall 86 - Spring 93
	<b>University of Vienna</b> (Vienna) Mag. phil. (=M.A.), <b>Philosophy</b> <i>Thesis:</i> Über die Bedingung der Möglichkeit von Metaphysik: die Grenzen der Letztbegründung" (engl. On the Condition of the Possibility of Metaphysics: The Limits of Ultimate Justification)	Fall 85 - Summer 89

## Work Experience

<b>Current</b>	<b>Visiting Assistant Professor</b> in the Department of Computer Science and Engineering, <b>University of Notre Dame</b>	<i>Current</i>
	<b>Lecturer</b> in the Department of Philosophy of Science and Social Studies of Science, <b>University of Vienna</b>	<i>Summer 00</i>
<b>Previous</b>	<b>Lecturer</b> in the Department of Formal Logic, <b>University of Vienna</b>	<i>Summer 99</i>
	<b>Visiting Assistant Professor</b> in the Department of Computer Science and Engineering at the <b>University of Notre Dame</b>	<i>Spring 99</i>
	<b>Visiting Assistant Professor</b> in the Department of Philosophy of Science and Social Studies of Science, <b>University of Vienna</b>	<i>Summer 98 - Fall 98</i>
	<b>Lecturer</b> in the Department of Formal Logic, <b>University of Vienna</b>	<i>Fall 97 - Spring 98</i>
	<b>Teaching Assistant</b> in the Department of Cognitive Science, <b>IU</b>	
	<b>Lecturer</b> in the Department of Formal Logic, <b>University of Vienna</b>	<i>Summer 97 Fall 96 - Spring 97</i>
	<b>Teaching Assistant</b> in the Department of Cognitive Science, <b>IU</b>	<i>Fall 95 - Spring 96</i>
	<b>Teaching Assistant</b> in the Department of Computer Science, <b>IU</b>	

## Publications

<b>Book</b>	Scheutz, Matthias (1995). <i>Ist das der Titel eines Buchs?- Selbstreferenz neu analysiert</i> . Wien: WUV (Wiener Universitätsverlag).
	Scheutz, Matthias (Ed.) (2000?). <i>Computationalism: New Directions</i> . MIT Press. Cambridge: MA ( <i>in statu nascendi</i> )
<b>Journal</b>	Scheutz, Matthias (Ed.) (2000) "Computationalism-The Next Generation?" <i>Special edition of the Conceptus series</i> . (forthcoming)
<b>Articles</b>	Scheutz, Matthias (1999) "When Physical Systems Realize Functions...". <i>Minds and Machines</i> . (forthcoming)
	Scheutz, Matthias (1999) "Implementation: Computationalism's Weak Spot". <i>Conceptus</i> . (forthcoming)
<b>Book Article</b>	Scheutz, Matthias (1999) "The Ontological Status of Representations". In A. Riegler, M. Peschl & A. von Stein (eds.) <i>Understanding Representation in the Cognitive Sciences</i> . Plenum Academic / Kluwer Publishers: Holland.



**Proceed-  
ings**

Scheutz, Matthias (1999) "A New Computationalism?". In *Proceedings of NTCS'99*. University of Vienna. MIT Cognet.

Scheutz, Matthias (1998) "Do Walls Compute After All? Challenging Copeland's Solution to Searle's Theorem Against Strong AI". In *Proceedings of the Ninth Midwest AI and Cognitive Science Conference*: AAAI Press.

Scheutz, Matthias (1997) "The Ontological Status of Representations". In *Proceedings of NTCS'97*. University of Vienna.

Scheutz, Matthias, and Naselaris, Thomas (1997). "Combining Genetic Algorithms and Neural Networks: Evolving the Vision System for an Autonomous Robot." In *Proceedings of the Eighth Midwest AI and Cognitive Science Conference*: AAAI Press.

Scheutz, Matthias, and Tillotson, Jenett (1996). "A Dynamic View of Reference". In *Proceedings of the Seventh Midwest AI and Cog Sci Conference* (published online).

**Book  
Reviews**

Scheutz, Matthias (1999) Book review of *The Philosophical Computer* by Patrick Grim, Gary Mar, and Paul St. Denis. *Philosophical Psychology*.

Scheutz, Matthias (1999) Book review of *Eine Elementare Einführung in die Theorie der Turing-Maschinen* by Oswald Wiener, Manuel Bonik und Robert Hödike. *Eureka*.