Post-publication version of "Aaron Sloman develops a distinctive view of – Virtual Machinery and Evolution of Mind (Part 1)" pages 97-102
In *Alan Turing - His Work and Impact*, eds S. B. Cooper & J. van Leeuwen, Elsevier 2013.

Aaron Sloman Develops a Distinctive View of VIRTUAL MACHINERY AND EVOLUTION OF MIND (PART 1)

September 8, 2013

1 Virtual machines and causation

The idea of implementing one Turing machine in another can be seen as a precursor of the increasingly important idea of a virtual machine running in a physical machine. Some features of virtual machinery that are potentially relevant to explaining the evolution of mind and consciousness will be discussed, including their causal powers and the differences between implementation and reduction.

One of Turing's achievements was the specification of a *Universal Turing Machine* (UTM) within which any other Turing machine could be emulated by specifying its properties on the tape of a UTM (Turing, 1936). This led to proofs of important theorems, e.g. about equivalence, decidability and complexity. It can also be seen as a precursor of what we now call virtual machinery (not to be confused with virtual reality). I shall try to show how the combination of virtuality, causal interaction and (relative) indefinability can produce something new to science. Part 2 (in **Part III** of this volume) will present implications regarding evolution of mind and consciousness.

2 Virtuality

The UTM idea established that a computing machine can run by being implemented as a *virtual* machine in another machine. (I think the gist of this idea was understood by Ada Lovelace a century earlier.) The mathematical properties of a Turing machine's trajectory through its state space will not depend on whether it is run directly in physical machinery or as a virtual machine implemented in another computation. This has proved immensely important for theorems of meta-mathematics and computer science and for some of the practicalities of using one computer for multiple purposes, including time-sharing. One of the consequences is that a Turing machine implementing another Turing machine can also be a virtual machine implemented in a UTM: so that layered implementations are possible.

In the decades following publication of Turing's paper, engineering developments emerged in parallel with mathematical developments, with some consequences that have not received much attention, but are of great philosophical interest and potentially also biological import.

I'll suggest in Part 2 that biological evolution 'discovered' many of the uses of virtual machinery long before we did. Unfortunately, the word 'virtual' suggests something 'unreal' or 'non-existent', whereas virtual machines can make things happen: they can be causes, with many effects, including physical effects. To that extent they, and the objects and processes that occur in them, are *real* not *virtual*!

A possible source of misunderstanding is the fact that among a subset of computer scientists the label 'virtual machine' refers to software implementations of 'real', 'physical' machines which they accurately simulate (Popek & Goldberg, 1974). The notion of 'virtual machine' used in this paper includes machines whose operations cannot all be defined in terms of physical properties, although they are all *implemented* in physical machinery, and can interact with and control physical machinery. These virtual machines should not be regarded as surrogates for 'real' physical machines. They are real enough, in their causal powers, despite being virtual.

3 Causation and computation

Causation is a crucial aspect of the engineering developments in computing, as I shall now try to explain. It is possible to take any finite collection of Turing machines and emulate them running in parallel, in synchrony, on a UTM. This demonstrates that *synchronised* parallelism does not produce any qualitatively new form of computation. The proofs are theorems about relationships between abstract mathematical structures including sequences of states of Turing machines – and do not mention physical causation. A running physical machine can be an instance of such an abstract mathematical structure. However, being physical it can be acted on by physical causes, e.g. causes that alter its speed. Moreover, as remarked in (Sloman, 1996), standard computability theorems do not apply to physical Turing machines that are not synchronised. For example, if TM T1 repeatedly outputs '0', and T2 repeatedly outputs 1, and the outputs are merged to form a binary sequence, then if something (e.g. a device controlled by a geiger counter) causes the speeds of T1 and T2 to vary randomly and they run forever, the result could (and most probably would) be a non-computable infinite binary sequence, even though each of T1 and T2 conforms to theorems about Turing machines. (This claim will be refuted if it ever turns out that the whole physical universe can be modelled on a single Turing machine. I know of no evidence that such a model is possible.)

Likewise, if a machine has physical sensors and some of its operations depend on the sensor readings, then the sequence of states generated may not be specifiable by any TM, if the environment is not equivalent to a TM. So the mathematical 'limit' theorems do not apply to all physically implemented information-processing systems. In fact a machine with sensors and effectors connected to physical objects in the environment is fundamentally different from a Turing machine running its 'closed' world consisting only of its (infinite tape) and controlling transition table.

Mathematical entities, such as numbers, functions, proofs, and abstract models of computation, do not have spatio-temporal locations, whereas running instances of computations do, some of them distributed across networks. Likewise, there are no causal connections, only logical connections, between the TM states that form the subject matter of the mathematical theory of computation, whereas there *are* causal connections in the running instances,

depending on the physical machinery used and the physical environment. So, notions like 'reliability' are relevant to the physical instances, but not the mathematical abstractions. From a mathematical point of view there is no difference between three separate computers running the same program, and a single computer simulating the three computers running the program. However an engineer aiming for reliability would choose three physically separate computers with a voting mechanism as part of a flight control system, rather than a mathematically equivalent, equally fast, implementation in a single computer (Sloman, 1996), if all the computers use equally reliable components.

Physical details of time-sharing of the machines have other consequences. When the three separate machines running in synchrony switch states in unison, nothing happens between the states, whereas in the time-shared implementation on a single computer, the underlying machine has to go through operations to switch from one virtual machine to another. Such 'context switching' processes have intermediate sub-states that do not occur in the parallel implementation. A detailed mathematical model of one machine running three virtual machines will need to include the intermediate states that occur during switching, but a model of three separate concurrently active machines. A malicious intruder, or a non-malicious operating system, will have opportunities to interfere with the time-shared systems during a context-switching process, e.g. modifying the emulated processes, interrupting them, or copying or modifying their internal data.

Such opportunities for intervention (e.g. checking that a sub-process does not violate access restrictions, or transferring information between devices) are often used both within individual computers and in networked computers causally linked to external environments, e.g. sensing or controlling physical devices, chemical plants, air-liners, commercial customers, social or economic systems, and many more. In some cases, analog-to-digital digital-to-analog converters, and direct memory access mechanisms now allow constant interaction between processes. See also (Dyson, 1997).

The technology supporting the causal interactions includes (in no significant order): *memory management, paging, cacheing, interfaces of many kinds, interfacing protocols, protocol converters, device drivers, interrupt handlers, schedulers, privilege mechanisms, resource control mechanisms, file-management systems, interpreters, compilers, 'run-time systems' for various languages, garbage collectors, mechanisms supporting abstract data types, inheritance mechanisms, debugging tools, pipes, sockets, shared memory systems, firewalls, virus checkers, security systems, operating systems, application development systems, name-servers*, and more. All of these can be seen as contributing to intricate webs of causal connections in running systems, including *preventing* things from happening, *enabling* certain things to happen in certain conditions, *ensuring* that if certain things happen then other things happen, and in some cases *maintaining mappings* between physical and virtual processes, e.g. in device drivers. Philosophers who think that different causal webs at different levels of abstraction cannot coexist need to learn more engineering, unfortunately not a standard component of a philosophy degree.

4 Causation in RVMs

A running virtual machine can have many effects, including causing its own structure to change. Understanding how virtual machines can cause anything to happen requires a three-

way distinction, between: (a) *Mathematical Models* (MMs), e.g. numbers, sets, grammars, proofs, etc., (b) *Physical Machines* (PMs), including atoms, voltages, chemical processes, electronic switches, etc., and (c) *Running Virtual Machines* (RVMs), e.g. calculators, games, formatters, provers, spelling checkers, email handlers, operating systems, etc., running in general-purpose computers.

MMs are static abstract structures, like proofs and axiom systems. Like numbers, they cannot *do* anything. They include Turing machine executions whose properties are the subject of mathematical proofs. Unfortunately some uses of 'virtual machine' refer to MMs, e.g. 'the Java virtual machine'. These are abstract, inactive, mathematical entities, not RVMs, whereas PMs and RVMs are active and cause things to happen.

Physical machines on our desks can now support varying collections of virtual machinery with various kinds of concurrently interacting components whose causal powers operate in parallel with the causal powers of underlying virtual or physical machines, and help to control those physical machines. Some of them are *application* RVMs that perform specific functions, e.g. playing chess, correcting spelling, handling email. Others are *platform* RVMs, like operating systems, or run-time systems of programming languages, which are capable of supporting many different higher level RVMs. Different RVMs have *different levels of granularity* and *different kinds of functionality*. They all differ from the granularity and functionality of the physical machinery. Relatively simple transitions in a RVM can use a very much larger collection of changes at the machine code level and an even larger collection of physical changes in the underlying PM – far more than any human can think about. Apart from the simplest programs, even machine-code specifications are unmanageable by human programmers. Automatic mechanisms (including compilers and interpreters) are used to ensure that machine-level processes support the intended RVMs.

Interpreted and compiled programming languages have important differences in this context. An interpreter ensures *dynamically* that the causal connections specified in the program, are maintained. If the program is changed while running, the interpreter's behaviour will change. In contrast, a compiler *statically* creates machine code instructions to ensure that the specifications in the program, are subsequently adhered to, and the original program, plays no role thereafter. Changing it has no effect, unless it is recompiled (e.g. if an *incremental* compiler is used). In principle the machine code instructions can be altered directly by a running program (e.g. using the 'poke' command in Basic) but this is usually feasible only for relatively simple changes and would probably not be suitable for altering a complex plan after new obstacles are detected, and modifying the physical wiring would be out of the question. So some kinds of self-monitoring and self-modification are simplest if done using process descriptions corresponding to a high level virtual machine specified in an interpreted formalism and least feasible if done at the level of physical, structure. Compiled machine code instructions are an intermediate case.

There are two different benefits of using a suitable RVM: namely (1) the already mentioned coarser granularity of events and states compared with a PM or low level RVM, and (2) the use of an ontology related to the application domain (e.g. playing chess, making airline reservations). Both of these are indispensable for processes of design, testing, debugging, extending, and also for run-time self-monitoring and control, which would be impossible to specify at the level of physical atoms, molecules or even transistors (partly because of explosive

combinatorics, especially on time-sharing, multi-processing systems where the mappings between virtual and physical machinery keep changing). The coarser grain, and application-centred ontology makes self-monitoring (like human debugging of the system) more practical when high-level interpreted programs, are run than when machine code compiled programs are run. This relates to the third aspect of some virtual machinery: ontological irreducibility.

5 Implementable but irreducible

The two main ideas presented so far are fairly familiar, namely (a) a VM can run on another (physical or virtual) machine, and (b) RVMs running in parallel can interact causally with one another and with things in the environment. A third consequence of 20th century technology is not so obvious, namely: *some VMs include states, processes and causal interactions whose descriptions require concepts that cannot be defined in terms of the language of the physical sciences: they are non-physically describable machines (NPDMs).* Virtual machinery can extend our ontology of types of causal interaction beyond physical interactions.

This is not a form of mysticism. It is related to the fact that a scientific theory can use concepts (e.g. 'gene', 'valence') that are *not definable* in terms of the actions and observations that scientists can perform. This contradicts both the 'concept empiricism' of philosophers like Berkeley and Hume, originally demolished in (Kant, 1781), and also its modern reincarnation, the 'symbol grounding' thesis popularised by (Harnad, 1990), which also claims that all concepts have to be derived from experience of instances. The alternative 'theory tethering' thesis, explained in (Sloman, 2007), is based on the conclusion in 20th Century philosophy of science that undefined symbols used in deep scientific theories get their meanings primarily, though not exclusively, from the structure of the theory, though a formalisation of such a theory need not fully determine what exactly it applies to in the world. The remaining indeterminacy of meaning is partly reduced by specifying forms of observation and experiment (e.g. 'meaning postulates' in (Carnap, 1947)) that are used in testing and applying the theory, 'tethering' the semantics of the theory. The meanings are never uniquely determined, since it is always possible for new observations and measurements (e.g. of charge on an electron) to be adopted as our knowledge and technology advance.

Ontologies used in specifying VMs, e.g. concepts like 'pawn', 'threat', 'capture', etc. used in specifying a chess VM, are also mainly defined by their role in the VM, whose specification expresses an explanatory theory about chess. Without making use of such concepts, which are not part of the ontology of physics, designers cannot develop implementations and users cannot understand what the program is for, or make use of it. So, when the VM runs, there is a physical implementation that is also running, but the two are not identical: there is an asymmetric relation between them. The PM is an *implementation* of the VM, but the VM is not an implementation of the VM, and there are many other statements that are true of one and false of the other. The RVM, but not the PM, may include threats, and defensive moves. And neither 'threat' nor 'defence' can be defined in the language of physics. Not all the concepts used to describe objects, events and processes in a RVM are *definable* in terms of concepts of physics even though the RVM is *implemented* in a physical machine.

The physical machine could include some of the environment with which the RVM interacts. The detailed description of the PM is not a specification of the VM, since the VM could be the same even if it were implemented on a very different physical machine

with different physical processes occurring during the execution even of a particular sequence of chess moves. The VM description is also not equivalent to any fixed *disjunction of descriptions* since the VM specification determines which PMs are adequate implementations. Programmers can make mistakes, and bugs in the virtual machinery are detected and removed, usually by altering a textual specification of the abstract virtual machinery not the physical machinery. When a bug in the program is fixed it does not have to be fixed differently for each physical implementation – a compiler or interpreter for the language handles the mapping between virtual machine and physical processes and those details are not part of the specification of the common virtual machine.

Neither can the VM machine states and processes be defined in terms of physical input-output specifications, since very different technologies can be used to implement interfaces for the same virtual machine, e.g. using mouse, keyboard, microphone or remote email for input. Moreover, some VMs perform much richer tasks than can be fully expressed in input-output relations, e.g. the visual system of a human (or future robot!) watching turbulent rapids in a river. (Compare the critique of Skinner in (Chomksy, 1959)).

The indefinability of VM ontologies in terms of PM ontologies does not imply that RVMs include some kind of 'spiritual stuff' that can exist independently of the physical implementation machinery, as assumed by those who believe in immortal minds, or souls. Despite the indefinability there are close causal connections between VM and PM states, but that includes things like detection of a threat causing a choice of defensive move, which is a VM process that can cause changes in the physical display and the physical memory contents. We thus have what is sometimes referred to as 'downwards causation', in addition to 'upwards causation' and 'sideways causation' (within the RVM).

6 Implications

The complex collection of hardware, firmware, and software technologies, developed since Turing's time has made possible information-processing systems of enormous complexity and sophistication performing many tasks that were previously performed only by humans and some that not even humans can perform. This has required new ways of thinking about *non-physically describable* virtual machinery (NPDVM) with causal powers. The new conceptual tools are relevant not only to engineering tasks but also to understanding what self-monitoring, self-controlling systems can do. Philosophy now has the task of working out in detail metaphysical implications of multiple coexisting causal webs with causation going sideways, upwards and downwards. Implications for evolution of mind are discussed in Part 2 of this paper, included in **Part III** of this volume. Finally, Part 3 of this paper, presenting the concept of meta-morphogenesis (the processes by which the processes of change and development change) will be included in **Part IV** of this volume.

References

Carnap, R. (1947). *Meaning and necessity: a study in semantics and modal logic*. Chicago: Chicago University Press.

Chomksy, N. (1959). Review of skinner's *Verbal Behaviour*. *Language*, *35*, 26–58.

Dyson, G. B. (1997). *Darwin Among The Machines: The Evolution Of Global Intelligence.* Reading, MA: Addison-Wesley.

Harnad, S. (1990). The Symbol Grounding Problem. *Physica D*, *42*, 335–346.

Kant, I. (1781). *Critique of pure reason.* London: Macmillan. (Translated (1929) by Norman Kemp Smith)

Popek, G. J., & Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, *17*. (7)

Sloman, A. (1996). Beyond turing equivalence. In P. Millican & A. Clark (Eds.), *Machines and thought: The legacy of alan turing (vol i)* (pp. 179–219). Oxford: The Clarendon Press. Available from `http://www.cs.bham.ac.uk/research/projects/cogaff/96-99.html#1` ((Presented at Turing90 Colloquium, Sussex University, April 1990)

Sloman, A. (2007). *Why symbol-grounding is both impossible and unnecessary, and why theory-tethering is more powerful anyway.* (Research Note No. COSY-PR-0705). Birmingham, UK. (http://www.cs.bham.ac.uk/research/projects/cogaff/talks/#models)

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, *42*(2), 230–265. Available from `http://www.thocp.net/biographies/papers/turing_oncomputablenumbers_1936.pdf`