What is the Role of the Observer in a Computation?

Peter Leupold[1]

Abstract. Computing by Observing is a model of computation that was inspired by the way experiments in the natural sciences are conducted. An observer writes a kind of protocol of some process, and this is the result, in our case of the computation. We explore in how far this notion of observer can reveal relevant facts for answering the question whether all computation is observer-relative in the sense of Searle.

We first argue that the observer must be allowed to do a little computing itself. If this is allowed, then rather simple processes can indeed be used as the central part of any computation, namely context-free rewriting processes. This gives some support to Searle's claim that just about anything can be seen as a digital computer. Further, there is one process that, with different observers, can be interpreted to compute any recursively enumerable function. So what is computed, is extremely relative to the observer in this case.

1 OBSERVATIONS ON THE OBSERVATION OF A SIMPLE COMPUTATION

We want to explore the role of an observer in a computation. But at a first look it seems that an observer does not form an integral part of the process of computing at all. Only the agent or mechanism that executes some rules of calculation seems to be necessary to compute something. To motivate our study, let us first look at a simple example of something that probably most of us will immediately recognize and accept as an example for a computation.

Suppose that we observe a person writing down the following sequence of symbols from left to right and top to bottom:

$$\begin{array}{ccc} & X & Y \\ + & & Z \\ \hline & Z & X \end{array}$$

The most probable consequence is that we will believe that we have observed this person doing an addition like $29 + 3 = 32$ or $59 + 6 = 65$ with the letters representing the respective digits. But these solutions only work in the decimal system that we are used to or in number systems of a higher base; with a smaller base, there is no number 9.

So for example an observer, who is more accustomed to numbers in base 8, would probably first arrive at one of the interpretations $27 + 3 = 32$ or $57 + 6 = 65$, because these are closer to his experiences and expectancies. The first point to note here is that different observers can interpret one and the same process as different calculations.

[1] University of Leipzig, Institute of Informatics, Germany, eMail: leupold@informatik.uni-leipzig.de

Secondly we note that in order to interpret the writing as a computation, the observer must be able to interpret not only the final result but the entire process. If someone simply spits out a number, there is no way of telling whether it has been computed in some way, or whether it is just produced randomly. So if the observer wants to judge, whether something is a computation at all, he must answer the more specific question what is computed. Without the meaning, computation and random symbol manipulations cannot be distinguished. We only consider the example from above a computation, if we are able to find digits that substitute the letters in a sensible way, i.e. if we manage to assign an interpretation to the steps of the process we witness.

This makes a point similar to the one of the Chinese Room argument, which in Searle's own words showed that '... semantics is not intrinsic to syntax' [12]. In our example different semantics can be assigned to the syntax. We can conclude that there is not one specific semantics that is intrinsic to the sequence of symbols, i.e. to the syntax.

However, when Searle speaks about observer-relativeness, he locates this between the levels of syntax and the physical implementation of this syntax. Thus our example does not yet bring us closer to his conclusion 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 a syntactical interpretation to the purely physical features of the system.' Because in our example syntax and the physics beneath it basically coincide, it does not serve to distinguish between these two levels. Nonetheless we want to exploit it a little bit more.

Basically the calculation (or: the manipulation of symbols) that we have described can be done by many different agents in similar ways: by a person as the computer that Alan Turing intended to describe, by a Turing Machine or other formal models of computation, and also by electronical computers as we use them. In all these cases an observer could interpret the process in different ways. So is computation observer-relative in a different sense? Can every computational process be interpreted as many different computations depending on the way it is observed? We could adopt two opposing points of view on this question.

The first point of view is along the lines we have argued so far: Obviously the two observers attribute different meanings to the process they observe. So there can be no doubt about the fact that the opinion on what is computed is observer-relative. Different observers associate different computations with one and the same process, when they observe it. However, the two observers described above agree in their judgment that something is computed. Both regard the writing of the symbols as a computation.

A third observer might attribute still another computational meaning to this process. Or, if he is not familiar with this method of addition, he might not be able to interpret the process in any way as a computation. So clearly, there is not one computation intrinsic to this calculation. It becomes a specific computation only by virtue of the observer's interpretation.

But if nothing is computed at all in the view of the third observer, is there computation –not a specific one– intrinsic to the process that is observed? One could argue, that this is not the case, and this is roughly the point of view of Searle [12]. He actually argues that this cannot be case at all, since computation is not an intrinsic feature of the world. It only arises, when somebody –the observer– gives a meaning to the process that is observed.

On the other hand, one might argue that the observer here is more than a mere observer. Even translating the string 32 into a number requires the computation $3 \cdot 10 + 2$ in the decimal system; similarly 65 in base eight is translated into a number via $6 \cdot 8 + 5 = 53$. For the decimal system we usually do not need to think about this kind of calculation, since we are so used to this system. But even for a two digit octal number most humans will need to compute seriously, which illustrates more clearly that even reading and understanding a number requires some computation. It is worth noting that addition and multiplication, which are necessary here, are not mere transductions, i.e. they cannot be done by standard transducers. So more than regular computational power is necessary to execute them.

Thus the difference in interpretations of the process from above is not merely in the observations. Rather, we have computations that consists of two parts. The first part is done by the observed computer, the second one by the observer, which is just another computer. Since the second part is different for the two observers, we actually deal with two different computations that just share one common part. As soon as we arrive at a number, that is, the idea of the number in the observer's mind, there cannot be any ambiguity any more. From this point of view, there is nothing observer-relative about computation.

However, this kind of computation exists only for the agent thinking about it. To communicate about it with others, he would have to translate it into some kind of syntax that others can see. And they would have to be able to interpret this syntax just as the observers in our example from above.

The question here is what we call a computation. Do syntactical manipulations as as the addition above count as computations? Or is a computation only the entire mapping from one mathematical concept to another? And must the assignment of meaning to the syntax be completely direct, or is a little bit of computation allowed in this process? Of course, the answer to these questions is fundamental for answering the question, whether all computation is observer-relative.

From our example we deduce the following: most people would say that the main part of the computation was done by the agent writing down the symbols, not by the observer. So it makes sense to allow some simple calculations to be done by an observer, if we want to capture the intuitive meaning of computation. Now the role of the observer is a crucial factor, when we speak about observer-relativeness. What is an observer allowed to do? What does it mean to observe? Observing the digit 5 lets us think of the corresponding number in a more direct way than observing 5534; and for octal numbers the way from reading to understanding the number is still less direct. Where do we draw the border between observation and computation?

We will now try to give some formal answers to these questions. Before we start, we should note that our example is still a crude simplification of the situation for Observer Systems, which we will now introduce. In all of the above interpretations, $XY + Z = ZX$ is just one instantiation of the function called addition. But an instance of a model of computation normally specifies an entire function with its infinitely many possible inputs.

2 COMPUTING BY OBSERVING

First off, let us again point out the following about the example from Section 1. In order to recognize that something has been computed, we need to look at all the notes that have been taken. Since they are relatively complete, we can reconstruct the rules that the computer followed.

If we do not understand the rules behind what is happening, it is hard to distinguish a computation from just random symbol manipulations. We may as well say that for us nothing is computed unless we know what is computed, because otherwise we cannot know the meaning of the result. Thus we need to see the process in a rather complete way. Suppose the computer in the introductory example had done a bigger part of the calculation only in his head. We could only give meaning to what we observe by doing the calculation again ourselves.

A somewhat similar situation is the standard setup of an experiment in the natural sciences. Take, for example, the relation between the population sizes of hunter and prey. From observing the population numbers over a long period of time, one can infer the rules that this dynamic equilibrium follows. This role of an observer that logs certain values produced by a process is essential to most experiments. Without this extraction of data there would be no extraction of knowledge.

With this we come to the field of Natural Computing. Its goal is the use of mechanisms present in nature for computing. Biochemical reactions changing DNA strands are an example for a candidate mechanism for building a whole new kind of computer. Many times it is even claimed that there is already some computation present in nature that we only need to discover. For example, Landweber and Kari say that '…ciliated protozoans of genus Oxytricha and Stylonychia had solved a potentially harder problem using DNA several million years earlier' [10]. Since they probably do not attribute any intention or understanding to the ciliate cells, this sounds as if for them computation was an intrinsic feature of the processes that happen in these cells. Later though, they slightly relativize this point of view and say that 'in principle, these unicellular organisms may have the capacity to perform at least any computation carried out by an electronic computer.'

Normally, the new models for computation that are based on biochemical phenomena present in nature follow the standard paradigm of Computer Science: an input is transformed into an output by some kind of process(or) that follows a certain programme or a set of rules. The output constitutes the result. This is the common approach from Adleman's seminal experiment [1] to the many theoretical models that have been designed since then [11, 8].

So there is a big difference between the ways in which computer scientists and researchers from experimental sciences "use" biochemical systems or abstractions thereof. Computer Science employs biochemical reactions, but does so in a different way from those scientists that have dealt with these phenomena for a much longer time. In the light of this, Matteo Cavaliere and the present author asked themselves, how one could formalize the role of the observer in an experiment that is so important, when scientists want to gain information about processes in nature. Or in other words: how could one compute with biochemical systems using the methods that are used by those people who have dealt with these kinds of systems for

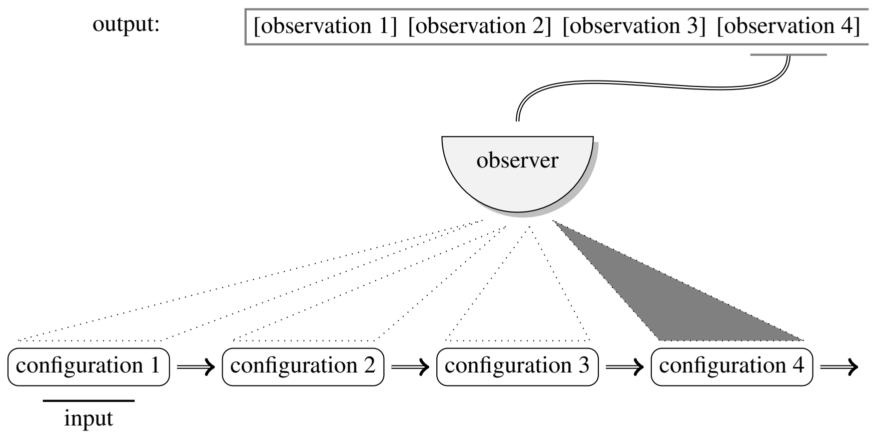

Figure 1. Schematic representation of a transducing observer system.

a much longer time than the ones who are aspiring to build biocomputers or to discover computation in nature?

The result was the paradigm called *Computing by Observing* that is inspired by the setup of experiments in the natural sciences. Figure 1 depicts the role of a separate observer in this architecture. As in most models of computation –be they standard ones or bio-inspired ones– there is a system that evolves from one configuration to another in discrete steps. This system starts working on the input. However, the output is not produced directly by this system. Rather, each configuration of the system is read by an observer that maps it to a letter. The concatenation of all these letters until the underlying system stops is then the result of the computation. Just like a sequence of (pairs of) numbers is the result of observing hunter and prey populations.

Thus the computation of the underlying system can work on data in any kind of format as long as the observer can read this format. For example, in the initial work on the topic Membrane Systems formed the underlying systems [5]. Their configurations are represented by multisets. Just as well, there could be a Turing Machine in the role of the underlying system. Then the observer would look at its state and its tape after every step that the machine takes.

In some sense the result that is produced by the observer is an abstraction of the entire computation. We assign one (or none) of finitely many letters to each one of the infinitely possible configurations of the underlying system. Thus we put them into one of finitely many equivalence classes. The final result is then isomorphic to a sequence of classes of configurations rather than to the sequence of configurations. Thus we lose some information, which in the best case is like losing the intermediate notes of a calculation and retaining only the final result.

Another important point is that we usually give the observer the possibility to stop a computation. That is, after reading a configuration it cannot only produce an output letter, but also a special output $\perp$ that immediately invalidates the entire result. In this way, the observer can separate good or meaningful computations from bad ones.

The first central question addressed in work on Computing by Observing has been the following: Is it possible to go beyond the computational power of the components by combining them in this way? Only then the additional effort would be justified. For example, if we already had Turing Machines as underlying systems, adding the observer could obviously not lead to an increase in computational power. The further the underlying system is from being computationally complete, the bigger the increase can be. So the question was if the architecture can lead to a gain in computational power and how big this gain can be?

Many different bio-inspired models and also plain string-rewriting systems were used to explore possible increases in power. Without entering into detail on this topic, the most common pattern was the following one: regular observers with underlying systems of context-free power sufficed to attain computational completeness, for example in the cases of grammars and string-rewriting systems [6, 7]. There were two exceptions to this. For sticker systems [2] and insertions systems [9] already systems of regular power were sufficient to compute everything that is Turing-computable. And, of course, sometimes there is no increase in power.

These differences raised another question: can we identify key features in the underlying systems that are crucial for a big increase in computational power? There seem to be two very important features. The first one is the ability to expand without limit the space that is used. Just like any class of Turing Machines with a space bound cannot be computationally complete, an observer system needs the ability to use an unlimited amount of space. For example, If this space is fixed or linearly bounded for a string-rewriting system, then this system and any regular observer can easily be simulated by a linear bounded automaton.

The second feature that is always present in the underlying system, when we observe a big increase in computational power is unlimited re-usability of the working space. For instance, for a string-rewriting system this means that the contents of a position can change arbitrarily often. In such a system, the only processing is the rewriting done by the rules. So information that is not rewritten any more cannot have any influence on the further evolution of the computation. Because we have only finitely many letters, frequent rewriting will, of course, eventually lead to a repetition in that position. But the relevant information is not only in the symbol itself, but also in the sequence it runs through. So unlimited access to the information that has been produced is essential.

Looking again at a linear bounded automaton, if we put a constant bound on the number of times that it can rewrite its cells, this further limits its computational power. So the key features for attaining computational completeness via Computing by Observing can also be recognized in classical models like the Turing Machine.

3 THE ROLE OF THE OBSERVER IN OBSERVER SYSTEMS

Without going into technical details, we have stated that the observers that were used always had regular computational power. So they were quite simple. They could not even do the additions and multiplications that are necessary to convert a decimal representation of a number into a number as we have mentioned in Section 1. Now we can ask the question whether there are cases, in which computations of the underlying system can be interpreted in different ways like in our introductory example.

Before answering this question, we again want to point out a weakness of the introductory example: The ambiguity works only for specific combinations of digits. By looking at several different additions we could eventually see that there is only one number system where all of them work. Already the number of distinct digits that appear would tell us with high probability, in which base the calculation is done.

For making our point, the example was good enough. But now we want to say that a system computes different functions depending on how it is observed. This means that these distinct observers must map every input to the corresponding output of the function that is computed. We give an example for such a case. To this end we need to enter into a little more detail about the definition of the instantiation of the Computing by Observing architecture that we will use.

An *accepting observer system* consists of a string-rewriting system and a monadic transducer. So the input is a string, and the rewriting-system works on that. It consist of rules of the form $u \to v$. Their application replaces an occurrence of u in a string by v. For example, $aabb$ can be converted to $abab$ by the rule $ab \to ba$. For more detail on this topic the reader may refer to the monograph of Book and Otto [3].

As described in Section 2, a monadic transducer must map every intermediate string to a symbol. It is just a finite automaton, with a one-letter output associated to each state. After reading a string, it outputs the letter associated to the state it has stopped in. The concatentation of these symbols determines, whether the input string is accepted or not. That is, if the observation belongs to a given regular language, then the corresponding input is accepted, otherwise not. As usual, an input is accepted if there exists at least one accepting computation for it. For more details on accepting observer systems the reader may consult the article of the present author with Matteo Cavaliere, where these systems were introduced [7].

In our example, the underlying system is the string-rewriting system with the four rules $a \to A$, $A \to b$, $b \to B$, and $B \to C$, and the input words consist only of the letters a and b. Configurations of such a system are simply strings. First we construct an observer, with which this system can recognize words of the form $a^n b^n$, i,e. a number of a followed by the same number of b.

The interesting computations in this case mark the left-most a by rewriting it to A; then they do the same to the left-most b by rewriting it to B. If both types of markings can be done the same number of times, obviously the original number of a and b was the same. The role of the observer consists mainly in filtering out those computations, where the string-rewriting rules are not applied in exactly the manner just described. This is why all the rewritings are done in two steps, for example $b \to B \to C$ instead of directly rewriting $b \to C$. In this way, every rewriting leaves a kind of trace (in this case the letter B) in a configuration and can thus be detected by the observer.

In detail, the observer realizes the following mapping:

$$\mathcal{O}(w) = \begin{cases} I & \text{if } w \in a^+b^+ \\ 1 & \text{if } w \in b^*Aa^*c^*b^* \\ 2 & \text{if } w \in b^*Aa^*c^*Bb^* \\ 3 & \text{if } w \in b^*a^*c^*Bb^* \\ 4 & \text{if } w \in b^*a^*c^*b^* \\ \oplus & \text{if } w \in b^*c^+ \\ \lambda & \text{if } w \in b^*Bc^+ \\ \perp & \text{else} \end{cases}$$

Here we use regular expressions to specify the languages and λ denotes the empty word. The words that are mapped to I are input words that have the correct order of symbols, i.e. only a followed by only b. The words mapped to numbers 1 to 4 correspond to the four phases of marking and rewriting one a and one b as described above. If we arrive at a string mapped to $\oplus$, the numbers of a and b were the same. Then it remains to rewrite all b to c, because the system has to stop in order to accept. Since there is no rule that rewrites c, a string of only c brings the system to a halt.

So the observations that lead to acceptance of the input string have the form $I(1234)^*\oplus^+$ and thus form a regular language. The language accepted by the system is $\{a^n b^n : n > 0\}$. Now we want to use the same string-rewriting system with a different observer to accept the following language: all the strings a^p where p is a prime. We use the facts that multiplication is repeated addition and that addition is just concatenation for unary numbers. Thus for every number k the string a^k can be factored into $a^i \cdot a^i \cdots a^i$ for every divisor of k. Therefore k is a prime if and only if such a factorization with $1 < i < k$ can be found.

The string-rewriting system realizes the following algorithm: guess a divisor d of the length of the input word. Rewrite the first d letters a to A and then to b. Now rewrite the left-most a to A, the left-most b to B and so on until all b are gone. Then all A are rewritten to b and all B to c. In this way we again arrive at a suffix of d letters b followed only by a. Iterating this, we rewrite the entire word to c with a final suffix of b only if the length is a multiple of d.

In this case the observer mapping is as follows:

$$\mathcal{O}_2(w) = \begin{cases} I & \text{if } w \in A^*a^+ \cup b^*A^+a^+ \\ S & \text{if } w \in bb^+a^+ \\ 1 & \text{if } w \in c^*b^*Bc^*a^* \\ 2 & \text{if } w \in c^*b^*Bc^*Aa^* \\ 3 & \text{if } w \in c^*b^*c^*Aa^* \\ 4 & \text{if } w \in c^*b^+c^+b^*a^* \\ F & \text{if } w \in c^+b^+a^+ \\ \oplus & \text{if } w \in c^+b^+ \\ \lambda & \text{if } w \in c^+Bb^+ \\ \perp & \text{else} \end{cases}$$

While the divisor is guessed, strings are mapped to I. When a string of class S is reached, the check for divisibility starts. The part bb^+ guarantees that we have not guessed the trivial divisor one, the presence of more a guarantees that we have not guessed the number itself. The rewriting of the letters b is started at the right end of the block. Otherwise the border between the two blocks would not remain clear after the first a is rewritten to b. We run through the phases 1 to 4 until the last iteration where the final configuration is mapped to F.

If there are no more a left, which means we have found a divisor, the resulting observation is $\oplus$. Then we only need to replace the remaining b by c so the systems stops, just like in the example above.

So a word is accepted if the observation belongs to the language $I^+S[(1234)^+(123F)]^*[(1234)^+(123\oplus)]\oplus^*$. If the input is not of prime length, then at some point there cannot be a 2 after a 1. So only the desired words are accepted.

What we have seen is that we can accept very different languages with the same string-rewriting system and different observers. So we have generalized the example from Section 1 from the calculation of one specific addition to the general notion of function as it is used in computability theory.

It is worth noting that the string-rewriting system we have used is extremely simple. Its rules only replace one single letter by another. The context cannot play any role, and the length of the string remains constant, when such a rule is applied. This type of string-rewriting systems is called a *painter* system. Taken by itself, such a system cannot compute much. To be more precise, a language is accepted by a painter system, iff it can be represented as A^*BA^* where A and B are arbitrary subsets of the alphabet. In the Computing by Observing architecture, however, all context-sensitive languages can be computed with these systems [7]. It is also clear that nothing more can be computed, because the working space is fixed and cannot be expanded.

After these two examples, it is clear that the string-rewriting system used above could also be used to accept many other languages. Straight-forward examples are $\{a^nb^na^n : n > 0\}$ or the language of all words that contain the same number of a and b. In general, there is no limit to the number of languages that can be accepted with the same underlying system.

The strongest result that generalizes our example is Theorem 3 in the work of Cavaliere, Frisco, and Hoogebom [4]. They construct a single rewriting system S that is universal in the following sense: for every Turing-computable language L there exists an observer, such that L is computed by S in combination with the specific observer. It is worth noting that the underlying rewriting system by itself, i.e. without the observer interpreting its derivations, is by no means Turing-complete. It has only context-free computational power.

So it is really the combination of the two components that yields the power. The underlying system cannot, for example, decide in a first step non-deterministically which language it computes, and then compute it like most universal Turing Machines do. The computational completeness really arises only from the interplay between S and the different observers that allow only the simulation of one specific Turing Machine each. But for every machine there is one fitting observer.

The result of Cavaliere, Frisco, and Hoogebom tells us that one and the same process can be used for doing the main part of any computation that is possible at all in the sense of Turing. In other words: more observer-relativeness is not possible in this context. Note, however, that the system itself is only a context-free grammar and thus very simple in its structure. It uses only two kinds of rules.

There are the very simple rules of the form $a \to b$ that only replace one letter by another. This is the kind of rule we have used in our example above and that is called painter rules. Secondly, as we have mentioned above, there must be a way of expanding the working space in order to reach computational completeness. This is done by rules $a \to bc$ that replace one letter by two letters. Nothing more is necessary to obtain the universal system. Actually, it is sufficient to have a single rule $a \to a\square$, where $\square$ is the new space that is introduced, and the other symbol stays the same. Apart from this we only need painter rules.

$a \to b$ and $a \to bc$ are two very simple patterns that –with a little phantasy– can be found in almost any place in nature. For instance, we can take biochemical reactions. $a \to bc$ could be the splitting up of a larger molecule, $a \to b$ could be a change in the folding of a protein. Looking only at the molecules and disregarding matters like energy that is set free or bound, this would be quite direct implementation of the rules. If we monitor just one organic molecule in a solution of many molecules, the rule $a \to a\square$ mentioned above could simply be the appending of another molecule at a certain point, for example the addition of one more base to a DNA strand.

Another instance would be cloud formation, droplets and ice crystals of different sizes form, collide, combine etc. $a \to b$ could formalize freezing, melting, or a change in the crystal formation or the kinetic energy; we only have to divide these features in into discrete intervals to obtain a finite number of symbols. The splitting up of a droplet or crystal gives us a rule $a \to bc$.

Of course, in both cases we have to abstract away from many details to see certain phenomena as instantiations of our rules. However, the same kind of abstraction is necessary when we look at human or electronic computers and interpret their syntactical manipulations as computations. Thus the result of Cavaliere, Frisco, and Hoogebom gives string support to Searle's claim that just about anything can be viewed as a digital computer.

4 WHAT CAN WE LEARN FROM COMPUTING BY OBSERVING?

After this technical excursion we now come back to the question, whether computation is always observer-relative. We have claimed at the end of Section 1 that there must indeed be some observer, if we do not regard purely syntactical manipulations as computations. A meaning is given to them only by some observer.

And what is more, this observer must basically understand the computational process and do a little bit of computation itself. As a side note we add that this understanding can be replaced by belief or trust. This is what happens for instance with most computer programmes. Users believe what the instruction manual or the programme's window on the screen tells them about what the programme has actually computed or can compute. Only if they trust that this is true the result has a meaning for the users. So the observation is replaced by belief. If there is no such description or similar knowledge from other sources, then we cannot use the result of a computer programme, let alone judge whether really a computation has been executed.

But if the observer has some basic abilities, in our case just those of a finite automaton, then even very simple processes can be interpreted as computations. What is more, even all the computable functions can then be computed by one and the same process. Of course, we cannot claim that these simple observers really understand the computation they observe. Rather, this understanding must have been present in the person, who designed or programmed them. But our monadic transducers still give a formal description of the way, in which a human can assign meaning to the observed process.

It has become clear that one and the same process can serve for many different computations. This is true not just in one instance like in our introductory example, but for entire functions. So there is definitely not one computation intrinsic to each computational process. This indicates that computation is not an intrinsic property of physical objects or processes at all. Rather it depends on the knowledge and phantasy of the observer, whether he can discover computation in a given process.

ACKNOWLEDGEMENTS

The present author wishes to acknowledge the contributions of Matteo Cavaliere to the technical work that constitutes the basis for the observations laid out here. Figure 1 was drawn by Norbert Hundeshagen.

REFERENCES

[1] L.M. Adleman, 'Molecular computation of solutions to combinatorial problems', *Science*, **226**, 1021–1024, (1994).

[2] Artiom Alhazov and Matteo Cavaliere, 'Computing by observing biosystems: The case of sticker systems', in *DNA*, eds., Claudio Ferretti, Giancarlo Mauri, and Claudio Zandron, volume 3384 of *Lecture Notes in Computer Science*, pp. 1–13. Springer, (2004).

[3] R. Book and F. Otto, *String-Rewriting Systems*, Springer, Berlin, 1993.

[4] Matteo Cavaliere, Pierluigi Frisco, and Hendrik Jan Hoogeboom, 'Computing by only observing', in *Developments in Language Theory*, eds., Oscar H. Ibarra and Zhe Dang, volume 4036 of *Lecture Notes in Computer Science*, pp. 304–314. Springer, (2006).

[5] Matteo Cavaliere and Peter Leupold, 'Evolution and observation: A new way to look at membrane systems', in *Workshop on Membrane Computing*, eds., Carlos Martín-Vide, Giancarlo Mauri, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, volume 2933 of *Lecture Notes in Computer Science*, pp. 70–87. Springer, (2003).

[6] Matteo Cavaliere and Peter Leupold, 'Evolution and observation — a non-standard way to generate formal languages', *Theoretical Computer Science*, **321**, 233–248, (2004).

[7] Matteo Cavaliere and Peter Leupold, 'Observation of string-rewriting systems', *Fundamenta Informaticae*, **74**(4), 447–462, (2006).

[8] Jürgen Dassow, Victor Mitrana, and Arto Salomaa, 'Operations and language generating devices suggested by the genome evolution', *Theoretical Computer Science*, **270**(1-2), 701–738, (2002).

[9] Alexander Krassovitskiy and Peter Leupold, 'Computing by observing insertion', in *LATA*, eds., Adrian Horia Dediu and Carlos Martín-Vide, volume 7183 of *Lecture Notes in Computer Science*, pp. 377–388. Springer, (2012).

[10] Laura F. Landweber and Lila Kari, 'Universal molecular computation in ciliates', in *Evolution as Computation*, eds., Laura F. Landweber and Erik Winfree, Natural Computing Series, 257–274, Springer Berlin Heidelberg, (2002).

[11] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, *DNA Computing – New Computing Paradigms*, Springer-Verlag, Berlin Heidelberg, 1998.

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