Life-Like Computing Beyond the Machine Metaphor

George Kampis

Dept. of Theoretical Chemistry, University of Tübingen, Tübingen

and

Dept of Ethology, Eötvös University, Budapest

Introduction

The question, what models, if any, can serve to represent the complexity of life, is a very important one. The application of biological ideas to novel software or hardware designs, or the seemingly opposite but in effect closely related task of using existing computers for the study of life-like phenomena requires an at least partial clarification of what computers can do. The subject of study of this paper is a foundational question of this kind.

Following a few earlier writings [Kampis 1991a, Kampis 1991b] we attempt here to give a short nontechnical summary for the nonspecialist of a set of general ideas about computer modelling, and to present an account of an operational modelling methodology for dealing with models of life, and in particular, with models of evolving systems.

Evolvability is perhaps the most distinctive characteristic of living systems. Many biologists like J. Maynard Smith [1975, 1986] or R. Dawkins [1985] consider this to be the key to life. Evolution produces novelty; so it seems quite natural to say that one of the primary problems of the modelling of life is that of the representation of novelty.

It will be suggested in this paper that in order to represent novelty one has to go beyond what is usually considered to be the computer approach or, rather, the *computer metaphor*. Afterwards, we shall outline a new model system, which is realizable by computations in an *approximative sense*, and can possibly serve as a useful tool for the study of novelty.

To describe the idea in a nutshell: we will point out that existing computer-based models of life correspond to encapsulated or enframed systems, the identity of which is subordinated to operations that delimit their complexity and thereby prevent them from incorporating new information. In contrast, the proposed new approach will make it possible to let the model components develop unexpected relations, and to extend their property spaces by means of what will be recognized as a *self-modifying behavior*.

Relationships of Complexity and Novelty

In an intuitive sense, we may say that a simple system is one for which the characterization does not require much effort. Such a system is readily exhaustible by an analysis which is not computationally intensive. A complex system is, on the other hand, computationally intensive and is not readily exhaustible. Nonlinear phenomena of dynamical systems, as exemplified by chaotic or cellular automata systems, are widely held to be complex, and nicely illustrate this principle which we may call "complexity-as-nontriviality".

It is useful to consider more abstract formalizations of complexity. Kolmogorov complexity, or information complexity [for a good review of this field, see Löfgren 1977 or 1987] and computational complexity [Garey and Johnson 1979, Wagner and Wechsung 1986] are two precise mathematical concepts that express essentially the same idea. What is deemed complex by these definitions is what requires a large amount of computer resources, that is, where much memory and/or execution time is needed to cope with the given system. Information complexity deals with the length of programs. In terms of information complexity, the most complex objects are those which cannot be described by any computer program shorter than an explicit list of the original object. In terms of computational complexity, the question is how computation speed depends on problem size. Those algorithms which require more than polynomial time for their execution are practically intractable because of their extensive demand of resources.

It is easy to see that these ideas are naturally related to that of novelty. A system that produces and utilizes novelty should become, in general, more complex. This is so because the novel elements that appear in a system also require a description and they also need time and memory if they are to be executed as programs; in either way, there will be an increased need for resources to cope with these elements. This reinforces the primary intuition that can be gained from the study of biological evolution. There, the appearance of novelty is usually associated with what is called *progress*, and this most frequently brings with it the increase of structural and functional complicatedness.

Now, in order to represent systems whose complexity increases we must face the problem of the *complexity bottleneck*. This notion was introduced and made precise by G. Chaitin [1987]. The complexity bottleneck involves the paradoxical issue of producing, in Chaitin's original words, a *"hundred-pound theorem"* in a *"ten-pound axiom system"*: something more complex in a system that is less complex. Chaitin has shown that such a production is not possible and hence the problem is unsolvable in this direct form. For us this suggests that, as is the case with Chaitin's Gödelian systems, also in evolution we may reach the limits of formal computability. Just why this is so, and what can we do next is exactly what we shall focus on.

What is a Computation?

Computers are central to our discussion. Therefore, and also in order to fix things, let us characterize computations very briefly. Consider, for instance, the so-called

Weak Church thesis : "Every effectively definable function is Turing-computable" [e.g. Yasuhara 1971, Rogers 1967].

What this means for the modeler is that there is an immediate connection between the more familiar notion of Turing Machines (TM-s) and the functions of logic and mathematics, and further, that, in a loose sense, all known models of computation are equivalent, and what is even more, they are directly equivalent to TM-s and to our present-day computers (papers in [Herken 1988] discuss various aspects of this curious equivalence).

A TM is a finite-state control system equipped with a reading and writing head as well as a tape, which can be extended without limits in at least one direction and is operated on by the control mechanism (Figure 1.). All a TM must be able to do, in oder to be

a universal computer, is to let the head move left and right, to stay where it is, and to replace the symbol of the tape under the head by a new one. The perhaps simplest abstract models of such universal computers were given in [Trakhtenbrot 1973].

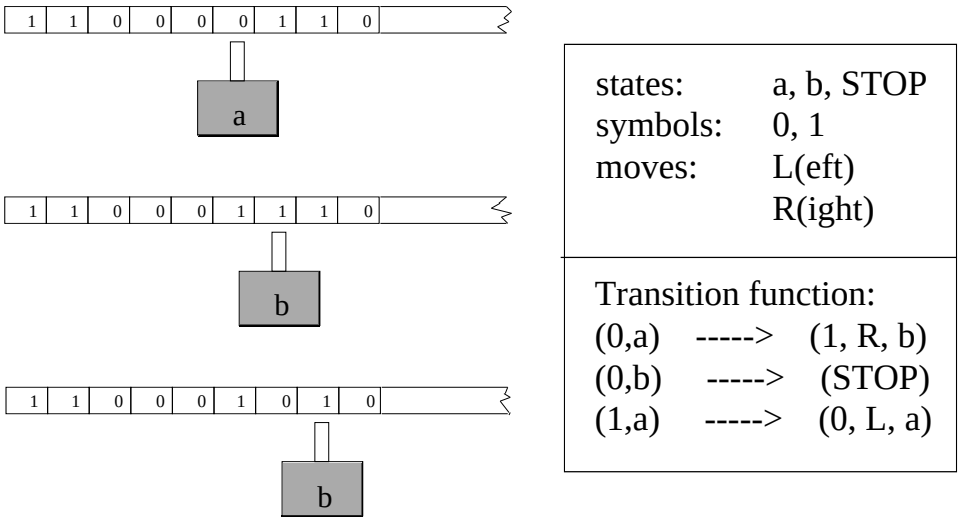

Figure 1.
Example for a Turing Machine

TM-s represent an attractive formalization of computing because the tape-and-controller system offers itself for a relatively easy study; from the Church thesis it follows that we can learn *everything* about computers by studying nothing but TM-s.

Now, if we analyse in detail what the TM model entails, we find that such a machine consists of a set of primitives (such as the permissible symbols of the tape and the states of the controller), a syntax for prescribing how the primitives are related, and a transition scheme to operate on them. Therefore, nothing a TM can ever produce will be different from logical tautology, *"tautology"* being a technical word now used for expressions obtained by repeated substitution. Wittgenstein was the first (in his correspondance with Bertrand Russell) to make this point clear; for an account, see his [1939]. (In fact Russell believed computation can produce something not contained in the premises and Wittgenstein pointed out that this is not the case.)

The elements that constitute a TM define a one-dimensional or string language. That is, ultimately, every computer is a *serial processor*; even cellular automata and other complicated systems like computer networks, some of which are intuitively felt "multidimensional" in fact restrict themselves to serial or string-processing modes. This is a somewhat counterintuitive fact.

The best way to see why every computation is a $1D$ process is to recall the fact that a computer is, according to one of its many equivalent formulations, a calculating machine or "number cruncher". In other words, whatever we do in a computer, it can be expressed

for the computer as numbers. Now, numbers inhabit simply the real line, no additional concepts are needed. K. Gödel has developed a technique generally called *Gödel numbering* which makes it possible to automatically execute the procedure of transforming everything to numbers and back. In other words, computer programs do nothing but elementary arithmetics - making ones from zeroes or back. In this sense, computational processes are extremely simple and rigid. All the complexity a computer can possess is the complexity of the numbers; all novelty it can represent is that of a *"one"* where previously there was a *"nought"*.

The Problem of Representation

Advanced or adventurous computing style can hide from the eye this underlying structure of a computational process. Unlike the above impoverished picture, on the screen of our desktop computers we find very flexible tools. There are invisible encodings behind them, that do a good part of the job for us, and it is only these encoding procedures, or rather, the programmers who develop them, who must be concerned with the Gödel numbers and the transition functions.

The reason why it is perhaps not pointless to recall these largely trivial facts about computing is that they indicate that it is not a computation in itself but the chosen *representation* (in other words, the fancy "blinking" of the screen rather than the numbers in the memory) that carries the relevant information to the user.

Let us consider this question: How is the complexity of a computation, understood in the terms discussed earlier, related to those intuitive complexity sensations which can be obtained by using a computer through the terminal? We can also reformulate the question as this: Can the use of suitable interfaces *enlarge* the capabilities of computers with respect to their ability to deal with complexity, or to produce new complexity? (That the interfaces do increase the strength of computers with respect to their "handiness" is obvious.)

Unfortunately, the answer must be negative. It is not the case that by changing representations, we can increase complexity; in fact mathematical complexity is defined in a representation-independent way. Or, maybe the question of representation is to be re-thought radically.

At the end of this paper we shall discuss how we can use "tricks" in order to form new kinds of representations which can in *some* sense go beyond the complexity which the underlying computers (i.e. the embedding machines and their programs) themselves offer. But the way typical computer models and typical computer embeddigs are defined today is rather self-delimiting; a new approach is needed.

Addressed directly, a computer must "know" in advance, what it should do; that is exactly what present-day programs are for. This gives us all the freedom (because, in a computer, unlike in the physical Universe surrounding us, only what *we* want will happen), but this also imposes a significant constraint. The embedding, or the interpretation, of every object that appears in a computer must be pre-specified, either directly, as in a look-up table, or indirectly, by means of some generating function that will be instantiated when necessary. In short, we have to think about every possible embedding before we let it work. We have to invent the future before it will be computed.

Object-oriented programming [e.g. Peterson 1987], a widespread current methodology for dealing with computational objects as *objects* on their own, provides good examples for what this current modelling strategy imples. Object-oriented languages like *C++* or *Oberon* [Reiser 1991] offer structures similar to those previously developed in AI under the name of *frame systems.* In both cases, the main idea is to represent objects as members of a class represented as a structured template that defines the properties, the inheritance, and the relations of its elements in some invariant scheme [Reichgelt 1991]. That is, we end up with an informationally closed, atomistic model.

Non-Computational Effects: Formal versus Physical Systems

Of course, whether there is anything beyond computers is a question many people are interested in, and recent debates about Chinese rooms and Emperor's minds put this question in the foreground. But in fact the line of work that deals very seriously with this very question goes back to A.N. Whitehead, the eminent mathematician and philosopher, co-author of the famous *Principia* [Russell and Whitehead 1912]. What I suggest to recall now owes a lot to him, in particular to his [1929].

Expressed in one sentence, what he pointed out was that even computations are not always *quite* like computations. As a consequence, the usual notion of computation turns out to be too restrictive, a little known idea that requires attention. (To be precise, Whitehead never spoke about computations, because there were no computers in his time. He spoke about using mathematics. However, by the Church Thesis, that makes no difference.)

A natural process is, strictly speaking, never reducible to a computation, Whitehead said. Perhaps the best way to show this is by pointing to the *physical* systems which we use in order to realize the *formal* computations. A basic fact is this: besides performing a computation, these systems always do *something else, as well.* One particular form in which they often do something else is what we call 'errors', as if Nature could go wrong. What happens actually when a computer makes an error is that *another process* interferes with the computation, in such a way which was not foreseen (included or accounted for) in the definition of the given computation. That is why we feel the computation breaks down. In other words, every natural process, even if specifically built for doing *nothing* but computations, does a lot more: there is a potential for further interactions in it.

At the moment when these interactions are "turned on" we transcend the initial computational framework. But of course nothing prevents us from *redefining* our initial computation now, so as to include these error-producing processes, as well; in this sense, the computational framework can be saved. (To put it differently, the idea is that the second physical process that embeds the original computation and is responsible for the errors of it can be incorporated in another, "bigger", and hence more complicated computation.)

What this all means can be illuminated by using a terminology pertaining not to computer science but to systems research [Klir 1985].

By confining ourselves to an initial computational description of a natural system, we define a *model.* A model is just exactly what we spoke about so far: it is a kind of an abstract construct, which permits a limited characterization which works in a "typical" case, but for the correctness of which there is no guarantee. That is, strictly speaking, computation is but a fruitful *idealization.* By enlarging the definition of computation to

include more, we define another model, another idealization — presumably, a better one. So the implied question is, how well can we do by using these sequentially refined models for the approximation of physical systems.

What Whitehead and others have noticed is that we encounter an infinite regress here. This refinement game is a race against Nature that cannot be won. No matter how much we have already included in a given mathematical supermodel, what we get is just another computation which would then suffer from new 'side effects', and new 'errors', and so on. Even a quantum description would not be an exception. Unexpected effects can come not only from "deep down" but from "the side" or "far away", as well: no model can be "fool-proof" against everything that affects its original. One can easily imagine such sudden effects as lightning, the entering of hackers in the room, or technicians who alter the hardware of a computer.

"Error" is but one instance of a wide class of unexpected new interactions that can occur within a system, Whitehead recognized. These interactions can drive a system into various new modes of processing we cannot account for in advance. Technically speaking, Whitehead concludes, in every *process* we deal with a potential infinitude of variables, whereas (expressed in modern terminology) in any *model* we use but a finite subset.

Whether this infinity (understood in the sense of the number of the potentially relevant variables being 'unbounded') has to be taken seriously is another question. Nature often appears to cooperate. For instance, many systems appear to be 'meaningfully' stratified, and that means that we can split them safely into well-defined and invariant subsystems that correspond to *levels of resolution*. The level-based approach to mathematical modelling is well-elaborated in [Klir 1985] and elsewhere.

It is easy to understand now, that the processes we call "computations" are typically processes confined to one level. That is, usually we do not have to bother with all the variables, other than just a few of them, maybe a handfull, which we have selected: these are the ones we include in a computational model or in the definition of a hardware. In particular, if we consider now a Turing Machine (such as the one on Fig. 1.) as a physical system, we find that it uses but a fragment of the available information carriers for actually representing information: for instance, in the *reading frame* defined by the interaction between the head and the memory tape, the majority of the material complexity is omitted (Figure 2.).

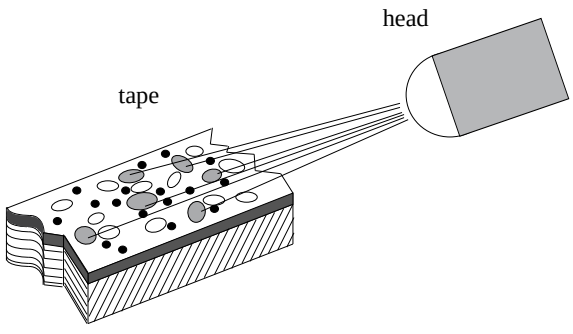

Figure 2.
Turing Machines use a fragment of their physical information potential

Shifting Reading Frames

Now, can we find any concrete process of interest, apart from maybe subatomic processes, that does not conform to some version of those reduced computations we have discussed so far? For even the molecular biologist finds it convenient to treat molecules as *information processors* that can be analyzed in terms of computer jargon: the very notions of "genetic code" and "genetic program" reflect this.

At the same time, there are various systems in biology that utilize what can be rightly called *shifting reading frames* and resemble the non-computational effects discussed above. That is, there are systems that can change and redefine their own primary use of information carriers. As the choice of information carriers or system variables is what makes computations different from physical systems, the shift of these variables is exactly like crossing the levels or changing the models in the Whiteheadian sense, or using a new reading/writing head in Fig. 2.

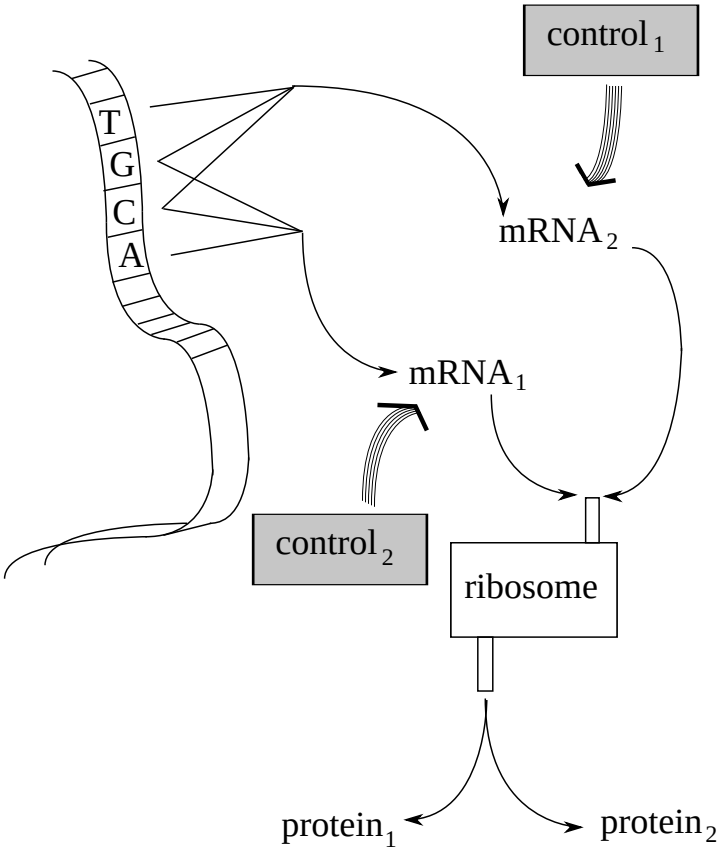

Figure 3.
The virus $\Phi X174$ with its actively selected coding methods

Let us take the example of DNA coding. Although the genetic code is universal, DNA or RNA sequences can nevertheless be interpreted in many different ways. For instance, the interpretation of a given DNA chain may depend on where we start to read its sequence. If we have an mRNA subchain $...AGACUG...$, it is still an open question, what counts as a triplet, is it now AGA, ACU, CUG, or is it something else? The way this selection is

made, that is, the way the biological readout frame is matched to the string, determines what amino acid sequence will be produced in turn. But this assignment is not unique, and there is no physical preference for all its variants. The selection of this interpretation is biologically controlled and it can be actively changed. As a result, the DNA (or, more precisely, the mRNA formed on the DNA) can use several code systems simultaneously.

Bacteria have a ring-shaped DNA that would offer itself for similar *"autonomous transformations of meaning"*. Yet perhaps the most profound example is provided by a virus called $\Phi X174$ in which the same piece of DNA code stands for two distinct structural proteins (Figure 3.) The choice is mediated by other enzymes that activate this or that pathway. In a permanent readout frame, this information content cannot be expressed. The lesson is that codes sometimes overlap and may involve several levels of organization.

Another typical example for the use of shifting frames is provided by catalytic proteins. Allosteric enzymes control biochemical production chains in the following way: they can be "blindfolded" by specific molecules in the absence of which the enyme works, and in the presence of which the ezyme is "silent". A mechanism called *end product inhibition* operates by using the same molecule for two different functions (that is, as catalyst and as a blindfold). In other words, this is another example where the structure is unique, but its use is not. How many uses a molecule can have mainly depends on how rich its environment is.

In the organismic realm, we have a perfect analog of these phenomena in the case of what is called the *principle of function change*. Popularized as "evolutionary tinkering" by F. Jacob [1981], this mechanism involves the change in the usage of organs. A standard example is the evolutionary formation of the lungs from parts of the digestive tract. In the same way, feet can turn into wings or fins, and so on. It seems everything may become something else in evolution. A remarkable fact is that this transformation can take place without much structural change or sometimes even without *any* structural change: it is just a matter of starting to use things differently. Later evolution can refine these organs to fit better for the new purpose, but they can be viable without such optimization. Besides being an important evolutionary mechanism, this phenomenon indicates a degree of autonomy of the phenotype in evolution. The phenotype is what is "visible" from the organism — better said, what is visible under the given interactions with the internal and external environment.

Classical neo-Darwinism thought that phenotypic function is uniquely determined by the genes and hence that evolution must always operate at the level of the genes. In other words, it was assumed that there is always an invariant and well-defined relation between the phenotype and the genotype, exactly as in the naive sense we assumed that there was a unique relation between a physical system and a computational process. The actual relation turns out to be more blurred, in both cases. In ontogeny, the genotype-phenotype relation can be retransformed or shifted by the environment and by other, partly internal, factors [Goodwin and Saunders 1992].

These mechanisms are important for understanding how life operates and how it produces new complexity, and they are promising candidates to be studied when dealing with biologically motivated computations.

"Non-Turing Machines" and the Idea of Context

When a reading frame is shifted a dramatic change occurs. In the cell, a new metabolic pathway is opened. In the computer, a new interaction takes places that invalidates a previous computation.

By this analogy, we find that the biological mechanisms discussed produce novelty that transcends a given set of computational rules. This idea, borrowed from molecules and evolution, is easy to generalize now in terms of Turing Machine-like constructs.

We can define a "non-Turing Machine" as one that operates with essentially the same structure as a Turing Machine (i.e. it has a finite state transition system, read and write operations, and elementary tape moves), except for one thing: that it uses a shifting reading frame instead of a predefined one. It is like traditional computers that permanently produce "errors", but unlike them does not break down, instead, incorporates these errors in its new basic definition. Such a machine could be realized, for instance, by finding a mechanism that develops new reading and writing methods (i.e. new tape handling methods) in runtime.

This possibility has interesting consequences for information theory besides modeling. In effect, this amounts to suggesting that the number of bits stored on a tape can be increased by adding new methods of storage. For instance, we can consider a machine, on the tape of which we tie a knot (Figure 4.). By developing a new reading method for the knot, the additional information, that is, the existence/nonexistence of the knot, can be accessed. However, and that is very important, *every* tape has a certain amount of knots on it (usually a zero amount as a special case). We are led to the conclusion that the overall information content only depends on the reading frame, and we don't even have to transform the tape. Again, the structure can remain invariant, but the function can change.

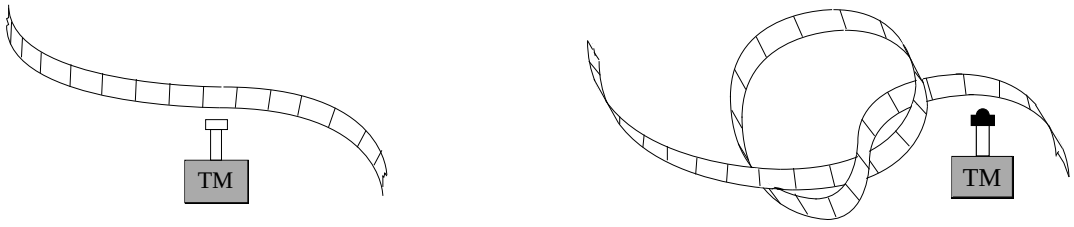

Figure 4.
Beyond Turing Machines: the use of a shifting reading frame

From Natural to Formal Systems

For the production of new contexts, and thereby of new information, there is an endless variety of candidates in perhaps every natural system. In actuality these possibilities can not be explored or even catalogued in advance. Yet we can make an active use of this

infinite regress foreseen (but not used) in customary computations. We can use it for the redefinition of the processes that will realize new computations.

We have an equivalent of the modelling process in terms of a hierarchy of Turing Machines with various reading heads. There exist Turing Machines with multiple heads which may therefore realize multiple reading frames and multiple function modes (that is, one for each head). These machines could try to simulate the shifting systems by simply switching between their predefined modes. But these machines must cling to a limited and predefined variety of frames. Technically speaking, the code sets of these machines must be *bounded*. There is a better alternative. It consists in organizing the information differently: not to rely on a fixed variety, and not to *store* the information about the reading methods. Unlike in usual Turing Machines we can try to reproduce them dynamically.

Component-Systems

We have claimed that certain biological mechanisms realize processes beyond ordinary computation, and that in principle this can be used for the redefinition of computers. The next question is: is there any method for making this idea operational, or should it remain a source of possible criticism against current modelling methods that use a more restricted classical notion of computation?

The biological examples we considered suggest that systems that define newer and newer properties of their constituting objects can be constructed "effectively", effectiveness understood in the material sense. Indeed, nothing prevents us from realizing the same biochemical processes as the ones discussed to produce shifts.

In other words, we have *examples* at hand; the problem is, how to generalize them in order to get the definition of a *class*. Let us turn to this question now.

Work is done on the theory of related systems in various forms (e.g. [Rosen 1985], [Minch 1988], [Conrad 1989]). One particular formulation is that of component-systems [Kampis 1991a].

The idea of component-system generalizes the known properties of macromolecular systems. A component-system is defined as one that *produces* and *destroys* its own components, which form an unlimited pool of component-types. By producing a new component, not only a new *copy* but often also a new *type* of component will be constructed. The idea is that, by introducing new types of components, new reading frames can be defined. This is so because new components can interact with old ones in new ways. That induces new properties in both. In short, the possibility for the emergence of new information processing pathways arises. Therefore, a component-system can be recognized as a variant of what we have informally called a "non-Turing Machine".

The theory of component systems has been developed in detail in [Kampis 1991a].

The Realization of Component-Systems on a Computer

Instead of giving an abstract characterization of component-systems, we are going to discuss their possible realizations here.

A straightforward method for the approximation of component-systems by means of computer models is to use a population of programs that operate on each other, so as to

produce new programs, in such a way that these "interactions" redefine or extend the functionality of the participating programs. In other words, we use programs as components, and we change the properties of components every time they interact with new ones. The interpretation is that we take the variants of a given program to be the context-dependent "facets" of a bigger, and invisible, underlying object, and we take the historical envelope of these variants to be the approximation of the object in question. (In this way, objects are fully constructed only at the *end* of a simulation; this is exactly the opposite of what is usually done today.)

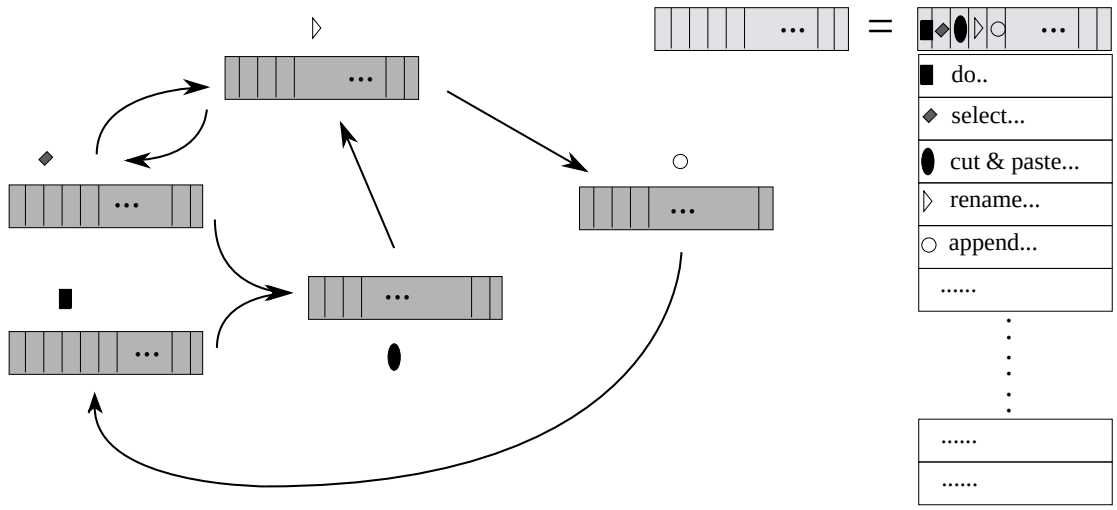

Figure 5.
The idea of a component system

A method by which the redefinition can be achieved is simply by *changing program codes during a run.* The production of new program code (i.e. the addition or alteration of program statements in a program list) will be made equivalent to the development of new properties for a component. Such a system can be conceived as a modified version of a *Markov normal algorithm* [Markov 1988] or of a *list processor* (Figure 5.). Also Markov algorithms and list processors are programs that can operate on programs.

In a setting like this, it is easy to embed shifting reading frames. Reading frames have to do with object-object interactions. We can now say that an interaction occurs if two programs "meet" in the sense that one or both emit a specific *call* for the other. To every call we maintain a separate list of statements defined in the given program (as, for instance, a production system in the sense used in *AI*), and this list will be activated upon a call. And if a given call is new, that is, if it's not on the list of the calls of the given program yet, a new set of program statements can be generated and added (a situation that will certainly occur sooner or later, if we keep on producing new programs that emit their new calls in turn). In this way, by adding self-modification we can simulate Nature's way of utilizing an endless store of "hidden properties", but without having to construct

the systems materially, or having to define these properties in advance.

Although fairly primitive and limited, such a system can already demonstrate the idea of information generation in an otherwise effectively defined system, and, as we shall discuss later, it can produce viable behaviors such as "natural selection" in a simple form.

Elements of a Modelling Philosophy

Let us make a few remarks about the very idea that we have just introduced.

Taken literally, one cannot exactly *realize* a concrete, real-world component-system on a computer, because that would require a knowledge of all the system variables and their interactions in advance. That would not be a component-system any more, since the components-systems are those that produce these modes themselves. What we do is to make computers behave like *artificial* component-systems. By studying these, we can study natural component-systems (or at least some of their generic properties) indirectly.

In other words, it will be difficult, if not impossible, to model the exact way a given evolving biological system defines and uses its new variables and properties. In this way, the model, understood in the strict sense, is given up by our methodology. Yet we can build another system, which probably does every detail differently, and where there is no direct correspondance between real-world events and model events, still, in our hope some basic characteristics of the original are retained. In other words, instead of representing one particular system in detail, we can try to build a model for a *class*.

Accordingly, a method for realizing component-systems as programs amounts to the utilization of several *independent* programs, built with a 'don't care' philosophy (that is, without specifying or pre-designing their interactions by hand). The idea is, don't design things beyond what is absolutely necessary, then lean back, and let the system work. This corresponds to the philosophy of having new and previously *unknown* properties that emerge in a material interaction in a natural system. In a biological system the novelty is bound to the material carrier, but this does not mean that there should be any relation between the old and the novel elements, or that we should consider the latter as pre-programmed: we can just as well think of them as if they were simply *random*, with respect to each other.

That is, for realizing component-systems we can apply some form of artificial chemistry with *new properties added randomly whenever new components interact*. To that, the only thing we need is random property generators which fill the interaction/property matrix and increase its dimensionality – or random program code generators, for that matter. The idea of random addition exemplifies the above "don't care" philosophy in a simplest way.

Here we understand that the computer can assume a new kind of role, not found in customary simulations. Unlike in traditional computing, where *arithmetic* and *logical* abilities of the computer are in the foreground, here the computer, as an embedding framework serves *combinatorial* purposes. The computer relates different sources of information. What these pieces are, where they come from, and what they do, should not be the given simulation program's concern: that can be pushed "behind the veil". What matters, is the way these elements are *organized*.

An Early Computer Model

The first computer model which utilized an early version of a similar philosophy was developed in 1983 and published in 1987 by the author and V. Csányi together [Kampis and Csányi 1987].

Figure 6.
A sample run of the self-modifying grid model

The model simulated an evolutionary process producing complex permanent structures in a system with changing composition. The system consisted of a two-dimensional

grid, the points of which corresponded to component-types in a functional rather than geometrical space. Components could interact with each other along the grid. The interactions were of two kinds. One of them modeled some aspects of chemical catalysis, whereas the other interaction stood for a less specific production of new components (in other words, it served for a random redistribution of matter in the phase space). Properties of the components (grid points) corresponded to the existence/nonexistence of a catalytic activity towards their neighbours. These properties could be changed by random errors and by the interactions themselves, according to a simple rule: if a component was produced anew so that its amount changed from zero to a positive number, it was assigned new random properties; in case of an opposite transition (where the result was the disappearance of a formerly existing component) we removed all its properties, so that the next time it was produced it could assume new ones. In the model, "catalysis" took simply the form of removing components from randomly selected locations of the functional space, and putting them to the grid point whose production was "catalized".

The model was able to produce results that indicated the emergence of self-selecting evolution and showed the spontaneous formation of "meaningful" stable structures, in spite of the ongoing self-modification (realized as the random change of component properties) and the other noise-like elements. This was the first proof that (1) self-modification of the discussed type is realizable, and that (2) it can lead to "interesting" behaviour. Permanent catalytic sets (similar to what are called Kauffman sets) were found to form as self-consistent solutions of the self-modifying process on the grid.

Figure 6. shows various stages of such a simulation, based on [Kampis and Csányi 1987]. The model has been significantly developed en extended since then.

The SPL System

A more recent model, which in fact is still under active development, will now be discussed in detail. It is based on a general string-processing language called SPL, developed by the author together with M. Vargyas. What will be presented here is a report of the system's basic structural construction and some current experiments that aim at the study of the principles outlined in the earlier sections.

SPL is part of a complex modelling environment that makes possible the interactive development and simulation of a wide class of biologically relevant computational models, including semi-computational simulations of component-systems.

The modelling environment provides tools for the automatic maintenance and the simultaneous execution of a population of programs that operate on each other and on themselves. Besides these new tools, there are several ways for introducing more conventional "errors"(like noise), and this gives the system the potential for a random search in a functional space. This latter property is shared by many recent evolutionary simulations, and it is one known to be applicable to optimization tasks based on biological mechanisms (as exemplified by genetic algorithms, and other systems). SPL, however, focusses on something else (as in fact there is no inherent need in SPL for self-reproduction, which is *the* key factor studied in GA's and other evolutionary simulations).

The structure of SPL

SPL is based on a machine-code-like instruction set designed to allow universal computation besides serving as the substrate for the biological simulation†.

The *SPL* instruction set is recapitulated in Figure 7. and will be discussed later. A formal definition of the language is given in the Appendix.

An important characteristic of SPL is its coherent *string-processing* philosophy. It realizes what was anticipated in Figure 4. Among other things, this means that, strictly speaking, there is no physical address associated with the program instructions. Rather, there is a *pattern-matching process* where the arguments of the operations are identified as labels directly on the executing program lists. Informally, this is much like the spatial mechanism by which molecules can find each other directly.

A typical operation of this system is a substitution, in the population of program strings, of the form $A_i \rightarrow B_j$, where A and B are patterns. This is a feature which allows for the use of techniques that are usually excluded from most other programming languages but are of central importance for us. (The actual execution of an *SPL* substitution takes more than one instruction – it takes at least two, such as a DEL and an INS – but that makes little difference.)

Another feature which needs discussion and which is, indeed, the expression of the central idea behind the whole *SPL* development, is this. In *SPL* we introduced two "metaphysical" operations RND and WISH. The effect of RND is to insert random bytes at a specified place (more precisely, at a given pattern), whereas that of WISH is the insertion of an entire random program, in the same way. The design of the *SPL* syntax together with the suitable construction of the embeddig environment makes every random program functional and executable in this system (that does not mean, of course, that such a random program would necessarily do something "useful"). This makes random progamming an easy job.

Formally, the introduction of the "metaphysical" operations renders *SPL* equivalent to a Turing Machine with random inputs. Methodologically, however, we use these random inputs in a novel way. The reason they are called "metaphysical" is that they can be directly applied to changing the system that uses them, and hence they express the idea of self-modification in the sense discussed earlier‡.

† Those familiar with the evolutionary model *Tierra* of Tom Ray or with its several relatives and predecessors, such as the *CoreWar* systems based on "Redcode", will recognize SPL as part of a series of efforts towards a general simulation language.

‡ Note that some primitive form of "self-modification" is provided by every list-processing language where codes that get altered during execution can be written. But this is different from the mechanism of introducing new, independent code when a program is called by another.

```
                        String Processing Language

registers (16 bit):                     [stack: STCKSIZE x 16 bit]
S1      address register                S2      address register
I1      index register                  I2      index register
D1      data register                   D2      data register

instruction set (8 bits, 28 instructions):
------------------------------------------------------------------------
control instructions
ADR     find pattern or empty room, write in (S1, I1)
JMP     goto pattern or [S1, I1]
IF      goto pattern or [S1, I1] if D1 != 0
CALL    subroutine call at pattern
RET     return to last CALL
REP     repeat D1 times
EREP    end of repeat block
------------------------------------------------------------------------
register instructions
LD      load [S1,I1] or @pattern to D1 (i.e. "read")
DL      download D to [S2, I2] or to pattern (i.e. "write")
PUSH    D1 to stack
POP     stack to D1
MOV     next byte into D1
XCH0    exchange (S1, I1) <--> (S2, I2)
XCH1    exchange D1 <--> I1
XCH2    exchange D1 <--> S1
XCH3    exchange D1 <--> D2
SWP     exchange bytes of D1
------------------------------------------------------------------------
arithmetical and logical instructions
ADD     add [S1, I1] or @pattern to D1
SUB     sub [S1, I1] or @pattern from D1
RADD    add stack to D1
RSUB    sub stack from D1
NAND    bw. D1 and stack
SHL     D1 = 2*D1
------------------------------------------------------------------------
transfer instructions
COPY    block copy from pattern to [S2, I2]
DEL     delete pattern
INS     insert pattern at [S2, I2]
------------------------------------------------------------------------
metaphysical instructions
RND     set D1 random
WISH    insert rnd string of rnd length to [S2, I2]
```

Figure 7.
The SPL instruction set

Besides these, the rest of SPL is concerned with customary computations. A working memory is provided in the form of registers, and we have also introduced the usual arithmetic and logical operations together with instructions for flow of control (like branching or cycles). This part is fairly standard and need not be detailed here.

How these instructions can be used for constructing biological models is briefly discussed next.

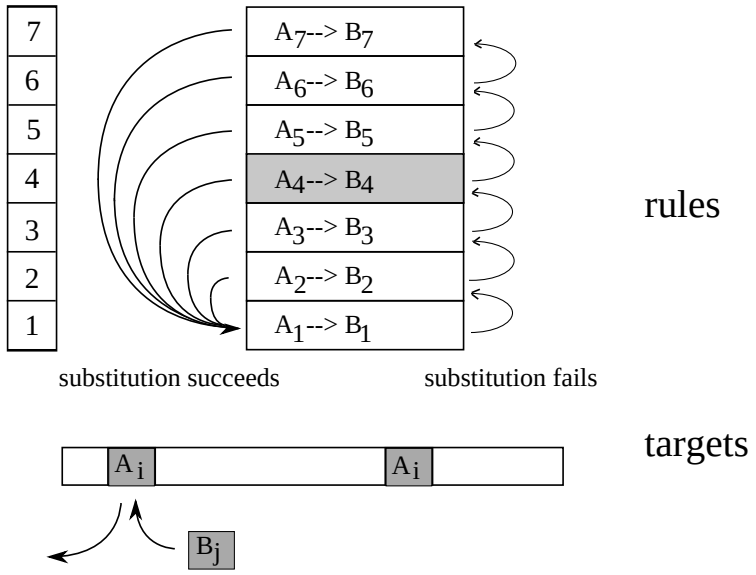

Figure 8.
Markov normal algorithm

A Hierarchy of Simulation Goals in SPL

Of particular interest is the following hierarchy of goals that can be achieved in this environment. Each step brings us closer to component-systems, the last element of the list.

Universal Turing Machines

A Universal Turing Machine can be realized in this system in several ways. One is by writing a single special *SPL* program with no random effects, no self-programming, and no use of the list-processing instructions. The key idea is, in any case, that the pattern matching operations should be used for establishing and removing a potentially unlimited amount of freely addressable temporal memory in the form of *new strings uniquely labelled by patterns*. These constitute the extendable tape of the Turing Machine.

"Data eaters"

Systems nicknamed "data eaters" are parallel nonrandom programs that use a population of strings for the simultaneous execution of arbitrary tasks and communicate by leaving nonexecutable traces (i.e. labelled data strings) to be picked up by other programs. In this way one can easily realize any parallel computation, in a design that shows some similarities to D. Gelernter's *Linda* system [Carriero and Gelernter 1990].

"Program eaters"

This is a more liberal structure where there is no difference between data and program. In *SPL* such differences can only be maintained by avoiding the use of certain operations; if

we program the string system with no restrictions we obtain programs that read and write each other's code deliberately. We call them "program eaters" because other programs are their "food", to be transformed ("digested") or to be destroyed ("used up").

As a special example, a wide variety of computer viruses and virus defense mechanisms can be written and tested in SPL, without making harm to the embeddig system. Work on the study of this possibility is under progress.

Molecular computers

Molecular computers are information processing systems that utilize principles of molecular interactions [Conrad 1985]. Of the various formalizations of such systems that can be naturally realized in SPL, we now consider the Liberman molecular computers [Liberman 1979].

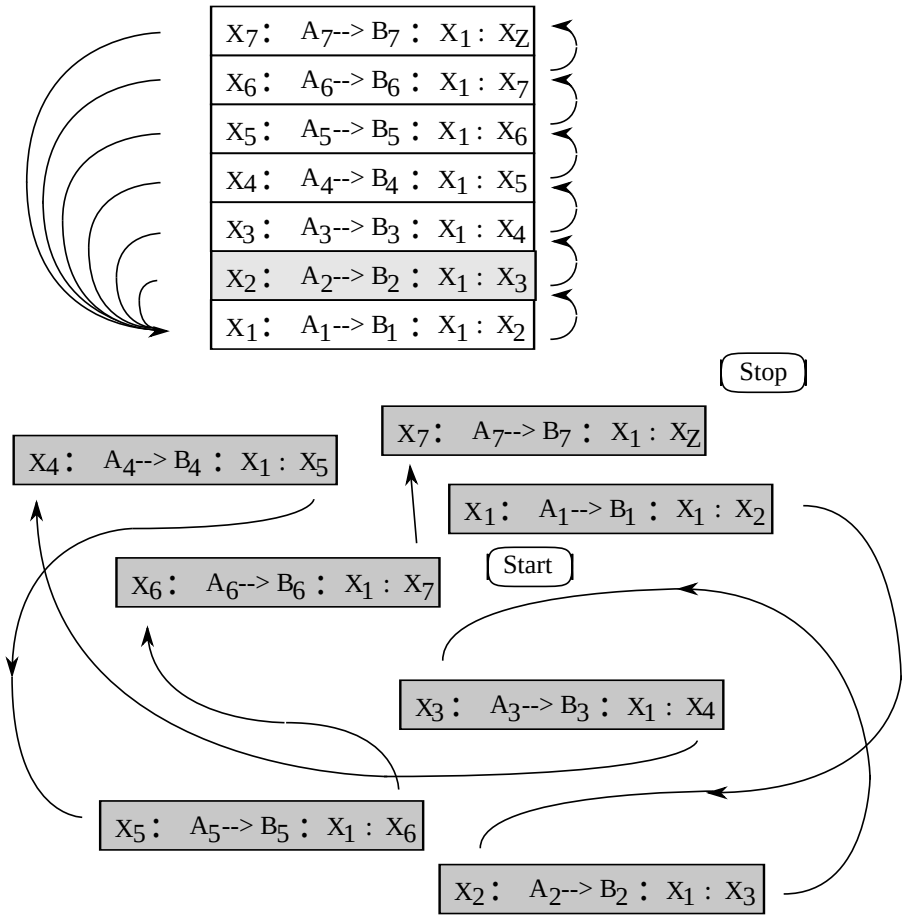

Figure 9.
A Liberman computer

A Liberman molecular computer is a pattern-matching and rewriting system with a Markov control flow. Recall from Figure 8. that the Markov control flow is this: If substitution $S_i : A_i \rightarrow B_i$ fails, goto S_{i+1}; if it succeeds, go back to S_1. In a string processing system this can be realized with a slight modification. A string C_i is supplemented with

a header H_i, and the substitutions are executed in strings with the respective headers (S_1 in H_1, S_2 in H_2, etc.). The control flow then takes this form: if S_i is successful, rewrite H_i asH_1; if it fails, rewrite it as H_{i+1} (Figure 9.). It is easy to see that such a system can be realized with the SPL instruction set: using patterns as headers and using SPLs pattern-directed rewriting rules as substitutions.

Tierra-like systems

Although this was not a goal in the SPL project to achieve, the system can easily embed evolutionary models of the Tierra type [Ray 1992]. In fact in SPL both simpler and more complex self-reproducing programs can exist than in Tierra and, much as in Psoup, another simulation program [de Groot 1992], these can form automatically from an initial population which contains no self-reproducing programs at all. For the sake of example, the simplest SPL program with such Darwinian reproduction and evolution capabilities we have found was

```
"GK ; define header for identification
ADR ; make room for offspring
XCHO ; change registers for COPY
COPY ""GK* ; copy the whole program from header
```

This is a truly trivial program of no interest — apart from the facts that it exists in SPL and that it can serve as a template for a large family of similar, autonomously developing programs..

Evolving "data eaters"

An interesting class of evolving programs, beyond the capability of the above simple selectionist system where mutation and reproduction are the only controls, is one where the evolving programs can also behave like "data eaters" at the same time. This makes the autonomous introduction (in runtime) of further selection criteria possible. The executed programs can develop new criteria for their own evolution by letting the reproduction of a given program depend on labelled data strings left behind by some other programs. As these latter programs (just as the evolution criteria themselves) can also be subjects of mutation and selection, we can get an intriguingly complex process where evolution has a coreography mutually defined and changed by the evolving subjects. The same idea can be applied to "program eaters", too.

Evolving problem solvers

A next logical step is to allow programs to pose problems for each other by means of communication in "data eater" mode. The target program can find and transform the data, and the source program can evaluate, in turn, the solution. Based on this, it can print out another data string that will be used by the target as an reproductive condition, and so on: there is an endless variety of interaction variants. As this process can proceed on a mutualistic basis, the evolving programs can learn how to solve the problems they have together invented.

Our experiments with evolving "data eaters" and evolving problem solvers are under way. Early results show that the difficulty is not in the writing of suitable SPL code

but in making the program population robust enough in order to last long enough until something interesting evolves: a problem not uncommon in "wet" biology either.

WISH-programming

There is a possibility for using the WISH and RND instructions for the writing and rewriting of programs. Specifically, the WISH programming philosophy suggests an unusual way for the filling of the environment with an initial population of random programs. A simple program called *"devil's factory"* can do that. As this program works indefinitely, another program is required to kill it after some time has passed.

With the combination of WISH programming with the other methods we discussed it is easy to write very short and surprising programs. As an exercise the reader may write a program which uses WISH or RND instructions for the the setting of cycle length in wait cycles (or string length in copy cycles etc.) to achieve random reproduction. Or, in an evolutionary simulation these mechanisms can be used for producing *"hot spots"* for the evolution. Hot spots are places where random "errors" occur much above the normal mutation rate. But we can also use similar strategies for more advanced applications:

Component-systems

Most importantly, SPL allows for the simulation of component-systems by using the strategy of WISH-programming.

A method for realizing component-systems is combining Liberman computers with WISH elements. When a substitution is not possible, a new header and a new program code will be generated by WISH for the target string. In this way we can obtain a sequence of programs that grow bigger and bigger as they encounter more and more substitutions that correspond to the specific "calls" we discussed. Selectively delayed delete instructions can then help these programs to get rid of their never used program parts, and so on: string programming gives all the flexibility required to such a delicate modelling task. All the programming can be done from within the components, and it can be done autonomously: after an initial startup, no intervention is needed.

Discussion

The system as described here became functional in February 1993 and was presented at two conferences since then. The system is equipped with an assembler/disassembler, a visual output generator, and a set of tools such as a dump tracer.

We have early results concerning several forms of the above hierarchy of goals. These results indicate that SPL is a viable tool for experimenting with information-producing systems. As the buildup and analysis of complex simulations is a very time-consuming activity, it takes a long way from here to go until the point where "meaningful" self-organizing structures such as the ones on Figure 5. can develop in SPL (and will be understood, once developed).

The SPL interpreter and the programming environment are available upon request from the author in MS-DOS format.

Acknowledgment

The paper was written with the support of research grant OTKA 2314 obtained from the Hungarian Academy of Sciences. The support is gratefully acknowledged. Part of the work was carried out during the author's stay at the Dept. of Theor. Chemistry, University of Tuebingen, Germany. The author wishes to thank Professor O.E. Rössler for his hospitality and co-operation. He thanks Mr M. Vargyas for his work on SPL and related matters. This paper uses figures and excerpts from other, noncopyrighted manuscripts of the author.

References

1. Carriero, N. and Gelernter, D. 1990: *How to write parallel programs: a first course*, MIT Press, Cambridge, Mass.
2. Chaitin, G.J. 1987: *Information, Randomness and Incompleteness*, World Scientific, Singapore.
3. Conrad, M. 1985: On Design Principles of a Molecular Computer, Comm. ACM **28**, 464-480.
4. Conrad, M. 1989: The Brain-Machine Disanalogy, BioSystems **22**, 197-213.
5. Dawkins, R. 1986: *The Blind Watchmaker*, W.W. Norton, New York.
6. Goodwin, B. and Saunders, P.T. (ed.) 1992: *Theoretical biology: epigenetic and evolutionary order from complex systems*, Johns Hopkins University Press, Baltimore.
7. de Groot, M. 1992: Psoup Manual. Obtainable from marc@os.com
8. Garey, M.R. and Johnson, D.S. 1979: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco.
9. Herken, R. (ed.)1988: *The Universal Turing Machine: A Half-Century Survey*, Oxford University Press, Oxford.
10. Jacob, F. 1981:*The Possible and the Actual*, University of Washington Press, Seattle.
11. Kampis, G. 1991a: *Self-Modifying Systems in Biology and Cognitive Science: A New Framework for Dynamics, Information and Complexity*, Pergamon, Oxford-New York, pp 543.
12. Kampis, G. (ed.) 1991: *Creativity in Nature, Mind and Society*, Gordon and Breach, New York. (Also available as Special Issue of World Futures: The Jnl. of General Evolution, **32**/2-3, 1991.)
13. Kampis, G. and Csányi, V. 1987: A Computer Model of Autogenesis, Kybernetes **16**, 169-181.
14. Klir, G. J. 1985: *Architecture of systems problem solving*, Plenum, New York.
15. Liberman, E.A. 1979: Analog-Digital Cell Computer, BioSystems 11, 111- 124.
16. Löfgren, L. 1977: Complexity of Descriptions of Systems: A Foundational Study, Int.J.General Systems **3**, 197-214.
17. Löfgren, L. 1987: Complexity of Systems, *in*: *Systems and Control Encyclopedia* (ed. M. Singh), Pergamon,Oxford, 704-709.
18. Markov, A.A. 1988: *The Theory of Algorithms*, Kluwer, Dordrecht.
19. Maynard Smith, J. 1975: *The Theory of Evolution*, Penguin, London.
20. Maynard Smith, J. 1986: *The Problems of Biology*, Oxford UP, Oxford
21. Minch, E. 1988: Representation of Hierarchical Structure in Evolving Networks, PhD Dissertation, Dept of Systems Science, SUNY at Binghamton, NY.

22. Peterson, G.E. (ed.) 1987: *Tutorial, Object-oriented Computing*, Computer Society Press of the IEEE , Washington.
23. Ray, T. 1992: Tierra & Tierra documentation. (Although there have been numerous writings on Tierra in the press, it is hard to find a quality publication. Hence, currently the texts that accompany the PD versions of the software are of primary importance. The whole package can be pulled down from the Net through: life.slhs.udel.edu)
24. Reichgelt, H. 1991: *Knowledge Representation: an AI Perspective*, Ablex Pub., Norwood.
25. Reiser, Martin 1991: *The Oberon System: User Guide and Programmer's Manual*, ACM Press, New York.
26. Rogers, H. 1967: *Theory of Recursive Functions and Effective Computability*, Mcgraw-Hill, New York.
27. Rosen, R. 1985: *Anticipatory systems : philosophical, mathematical, and methodological foundations*, Pergamon, Oxford.
28. Russell, B. and Whitehead, B. 1912: *Principia Mathematica*
29. Trakhtrenbrot, B. A. 1973: *Finite automata; behavior and synthesis*, North-Holland, Amsterdam.
30. Wagner, K. and Wechsung, G. 1986: *Computational Complexity*, Reidel, Dordrecht.
31. Whitehead, A.N. 1929: *Process and Reality*, Free Press, New York.
32. Wittgenstein, L. 1976: *Lectures on the foundations of mathematics.* Wittgenstein's Lectures on the foundations of mathematics, Cambridge, 1939 (from the notes of R. G. Bosanquet, Norman Malcolm, Rush Rhees, and Yorick Smythies, edited by Cora Diamond), Cornell University Press, Ithaca.
33. Yasuhara, A. 1971: *Recursive Function Theory and Logic*, Academic Press, New York.

Appendix

Syntactic definition of the SPL language

```
<statement> ::= <single instruction> | <string> | <transfer> | <move>

<string> ::= <string instruction> <global pattern> |
             <string instruction> <local pattern> |
             <string instruction> <empty pattern>

<transfer> ::= <transfer instruction> <direct pattern> |
               <transfer instruction> <indirect pattern>

<global pattern> ::= " <pattern> <eos>

<local pattern> ::= <pattern> <eos>

<empty pattern> ::= " <eos>

<direct pattern> ::= <pattern> <eos>

<indirect pattern> ::= " <pattern> <eos>

<pattern> ::= <character> <pattern> | <character>

<move> ::= MOVE <byte>

<string instruction> ::= ADR | JMP | IF | CALL | LD | DL | ADD | SUB

<transfer instruction> ::= COPY | DEL | INS

<single instruction> ::= RND | WISH | REP | EREP | RET | PUSH | POP | XCH0 |
                         XCH1 | XCH2 | XCH3 | SWP | RADD | RSUB | NAND | SHL

<character> ::= ' ' | '!' | .. | '}' | '~'

<byte> ::= 0 | 1 | .. | 255

<eos> ::= 0
```