An architecture for self-organising evolvable virtual machines

Mariusz Nowostawski and Martin Purvis and Stephen Cranefield

Information Science Department
University of Otago, Dunedin, New Zealand
(mnowostawski,mpurvis)@infoscience.otago.ac.nz

Abstract. Contemporary software systems are exposed to demanding, dynamic, and unpredictable environments where the traditional adaptability mechanisms may not be sufficient. To imitate and fully benefit from life-like adaptability in software systems, that might come closer to the complexity levels of biological organisms, we seek a formal mathematical model of certain fundamental concepts such as: life, organism, evolvability and adaptation. In this work we will concentrate on the concept of software evolvability. Our work proposes an evolutionary computation model, based on the theory of hypercycles and autopoiesis. The intrinsic properties of hypercycles allow them to evolve into higher levels of complexity, analogous to multi-level, or hierarchical evolutionary processes. We aim to obtain structures of self-maintaining ensembles, that are hierarchically organised, and our primary focus is on such open-ended hierarchically organised evolution.

1 Introduction

The rapid growth of complexity in different areas of technology stimulates research in the field of engineering of self-organising and adaptive computation systems. Adaptive software models refer to generic concepts such as *adaptability* and *evolution*. This, on the other hand, inherently leads to fundamental questions about the nature of open-ended uniform evolutionary processes: their essential properties, minimal requirements, architectures, models, and evolution of evolvability. Answers to some of these fundamental questions will lead to progress in automatic evolutionary design of computational machines and in engineering techniques for self-organising and self-adaptable software systems.

1.1 Traditional methods

In common English usage *adaptation* means the act of changing something to make it suitable for a new purpose or situation. In software systems, the term adaptation is being used mostly, if not exclusively, with a second semantic meaning. What is usually meant by software adaptation is that the system will continue to fulfil its original purpose in new or changing circumstances, situations or environments. The adaptability in such software systems may be achieved by a feedback loops between the system, the controller monitoring and changing and adapting the system, and the environment

itself. The system purpose is pre-defined in advance as a set of specifications, which are kept within the controller. The behaviour of the system is automatically altered if the expected outputs are outside of these pre-defined specifications. Such models operate analogously to the way automatic control systems work (15). Most of them are based on top-down design and work well in limited environments, where changes in the environment can be predicted and constrained in advance (21). Such adaptive systems are tuned to particular kinds and specific levels of change in the environment.

Most of the adaptability in traditional software systems is achieved via control mechanisms like in automatics. There is a central system, with a set of sensors and actuators, a controller, and an environment. Sensors sense an environment, the system and controller are tied via a set of feedback loops, and the controller tries to keep the system within pre-defined boundaries. This model can be implemented in a straightforward fashion, however it is static and must be applied in situations where it is possible to predict in advance all the changes and variations of the environment. To make things more robust and flexible, it is possible to implement into the controller an ability to learn, so the rules for changing the system become more dynamic, thereby enabling the entire ensemble to follow changes in more dynamic environments. Yet such systems still suffer from drawbacks associated with the simple control model. Even though the system shows some adaptability on another scale of complexity, there are limits of environmental change with which the system can cope. And these limits are pre-established with the structure of the learning mechanism itself.

1.2 New requirements

Contemporary software systems, especially open multi-agent distributed systems, eg. (26), that may potentially be spread around the globe and interact with various changing web-services and web-technologies, are exposed to demanding, dynamic, and unpredictable environments where the traditional adaptability mechanisms may not be sufficient.

To imitate and fully benefit from life-like adaptability in software systems, that (at least in theory) might come closer to the complexity levels of biological organisms, we seek a formal mathematical model of certain fundamental concepts such as: life, organism, evolvability and adaptation. In this work we will concentrate on the concept of software evolvability.

The landmark step in understanding the evolutionary process of living organisms in natural life was done by Darwin (4), who proposed mechanisms by which purposeful adaptive changes take place via processes of random mutation and natural selection. Darwinian mechanisms postulate reproduction, the statistical character of change processes, and the process of elimination (after elimination the organism ceases to exist, i.e. is not alive anymore).

1.3 Computation and biological inspirations

In this work we use a theory of evolvable virtual machines, which exhibits adaptability and self-organisation. The model has been inspired by ideas that have been developed over the last decades. The roots of the proposed model can be traced back to the work of John von Neumann (38; 39), who submitted that a precise mathematical definition

should be given to basic biological theories. This has been most prominently continued and extended by Gregory Chaitin (2; 3).

Some current research in evolutionary computation (EC) is emphasising information-centric methods that mirror Darwinian theory of random mutations and natural selection. This is visible in well-established computational optimisation methods, such as Genetic Algorithms (GA), Genetic Programming (GP), and their variations, such as assorted Artificial Life systems. Despite some successes, the typical simple single-layer evolutionary systems based on random mutation and selection have been shown to be insufficient (in principle) to produce an open-ended evolutionary process with potential multiple levels of genetic material translation, see e.g. (5; 41).

Our work proposes an alternative path, based on the theory of hypercycles (5) and autopoiesis (20). The intrinsic properties of hypercycles allow them to evolve into higher levels of complexity, analogous to multi-level, or hierarchical evolutionary processes. We aim to obtain structures of self-maintaining ensembles, that are hierarchically organised, and our primary focus is on such open-ended hierarchically organised evolution.

2 Computational evolution

2.1 Information-centric approach

It is believed by some that the information-centric approach is a correct, if not the only possible, path to pursue the research and make progress in the field of theoretical and computational biology (23; 3). In our work, we use some of the basic concepts of the information-centric approach, and throughout this work we will use two basic notions of information as introduced by Shannon (33) and in Kolmogorov-Solomonoff-Chaitin algorithmic information theory (18). We will refer to the Shannon notion as *information* and to the Kolmogorov-Solomonoff-Chaitin notion as *algorithmic information*.

There have been many more or less formal attempts to define life, complexity, organism, organism boundaries, and information content (32; 25; 19). Some authors have attempted to give rigorous quantitative definitions of these concepts, in a formal deductive form (39; 3; 10). Interestingly, authors coming independently from different sets of basic definitions and assumptions reached the same or very similar conclusions (e.g. (3) and (10)). According to theoretical and experimental work of most authors, the process of improvement in individuals and ensemble growth are best accomplished by carrying along all, or almost all, of the previously developed structures while new pieces of an ensemble structure are being added (34). Simulations and statistical analysis in the fields of Artificial Life experimentally confirm the efficiency of this approach. Recent work in incremental reinforcement learning methods also advocate retention of learned structures (or learned information) (e.g. (30)). The sub-structures developed or acquired during the history of the program self-improvement process are kept in the program data-structures. It therefore comes as a bit of surprise that this general procedure is not being exhibited by any of (standard) evolutionary programming models (7) such as: Genetic Programming (GP) (16)) or Genetic Algorithms (GA) (40). Although these evolutionary programming models are inspired by biological evolution, they do

not share some significant aspects that are recognised in current evolutionary biology, neither can they be used (directly) in incremental self-improvement fashion.

We are seeking a new, robust, and flexible evolutionary model, that can accommodate meta-learning and incremental self-improvement, as well as hierarchically organised evolutionary processes.

2.2 Information measure

Information is a measure based on a selection from a set of available choices. *Algorithmic information* is a uniform measurement of encoding information relative to the given computing machine (virtual machine).

The amount of information is based on the ability to make a correct selection from a given set. Let us consider a unique code $k \subset X \times Y$, i.e. $y = k(x)$, where $x \in X, y \in Y$, x represents a given condition, and y represents the correct selection. X and Y can be any sets, but in the context of finite state machines and discrete computation, one can treat them as sets of program blocks. Let us use index g to indicate a particular selection of x for a given y (goal). This model can be expressed now as $x_g = k^{-1}(y_g)$. Selection of a single unique condition x_g gives us all necessary information to obtain the output y_g: $I(x_g) = -log\, p(x_g)$, where $p(x_g)$ is the probability of picking a correct condition x, and I is the information content of a particular x_g (33; 1).

One of the possible ways to refer to the probability distribution $p(x)$ is to compare it with the reference distribution of the system. We can take as a reference a system that makes all possible choices with equal probability. Such a system would have maximum entropy (equivalent to the thermodynamical state of equilibrium). By *entropy* we mean an information theory measure which, when applied to an information source, determines the maximum channel capacity to transmit the source encoded according to a particular signal alphabet. The state of maximum information entropy we will refer to as a system in *abiotic equilibrium*. In such a state the probability of a correct selection of a given condition for a given output is uniformly distributed across all the possible conditions, and therefore all selections are equally probable. By calculating the difference between the actual $p(x)$ and this uniform abiotic distribution, one can calculate the information content needed to make a correct selection. (This is the difference between the given channel and one with maximum information capacity.)

In the context of incremental search methods through program search-space, as in the case of (31), $p(x)$ can be interpreted as the probability of executing a particular instruction during the course of program execution. The program itself can accumulate information about its environment and requirements by adjusting these probability distributions. A program can modify the probability distributions for different instructions at runtime. The difference between abiotic probability distributions (the initial uniform distribution) and the given probability distribution of a given system will be the measure of acquired information.

2.3 Darwinian systems

Darwin's principle of natural selection is widely used in current computational models of evolutionary systems for optimisation or simulation purposes (in fact in Evolutionary

Computation in general). Some authors regard natural selection as axiomatic, but this assumption is not necessary. Natural selection is simply a consequence of the properties of population dynamics subjected to specified external constraints. The main objective of the work of Darwin and Wallace (4) was to provide some basic insights into the process of evolution and the phylogenetic interrelations among species.

There are some inherent properties of conventional computational Darwinian systems which are sometimes overlooked. Darwinian systems rely on the concept of an environment with embedded self-replicating entities competing for resources and reproduction. For the model to be consistent, one has to postulate a stable species which competes for selective preferences and a stable reproduction of the best adapted species. In other words the model postulates that the selection operates purely on the individuals, hence there is only a flat single level of individuals which the evolutionary processes operate on. In such a model, there is a limit to the amount of the information content a stable species can have. Therefore the evolution of such a system is limited to a certain level of complexity defined by the threshold for maximum information content for a given setup/configuration of species. To overcome this threshold, further levels of selection and evolutionary information translation would need to be introduced into the system.

A second important aspect concerns random mutations. In real biological systems, due to overly complex and dynamic environments, the mutations can simply be a function of the environment. There is no need to postulate an external, god-like source of randomness. However, in computational models, the environments are highly regular and fixed, and the only way to introduce the necessary noise to the search process is by introducing an external source of randomness. This, as with certain random-search optimisation methods, can be useful, and in fact works quite well for some classes of problems (with the main computational techniques employed being Genetic Algorithms, Genetic Programming, and other evolutionary computation optimisation methods).

However in the context of adaptable, self-organising and open-ended evolutionary software system that exist in dynamic environments, it makes little sense to introduce additional external sources of randomness. By definition, the environment should supply all the randomness for the adaptable software system. If the environment stabilizes, the system should stabilize as well. It is a simple result of maximisation of the system aptness to the given environment. On the other hand, the learning mechanism involved in adapting the system to the given environment, or to the changes in the environment, may internally need to use some sort of probability distribution (that is represented internally, or externally via the environment itself). This is, however, a different matter to an artificially introduced external source of randomness, as in evolutionary computation.

One possible way of dealing with that is via bias-optimal search methods (17; 29), or via incremental search methods (31). To narrow the search, one can combine several methods, for example it is possible to construct a generator of problem solver generators, and employ multiple meta-learning strategies. We will discuss some of the details further in the following sections.

3 Autopoietic hypercycles

3.1 Hypercycle

Lets consider a sequence of reactions in which products with or without the help of additional reactants undergo further transformations. The *reaction cycle* or *cycle* is such a sequence of reactions in which some of the products are identical with the reactant of any previous step of the sequence. The most basic is a three-membered cycle, with a substrate, enzyme, and a product. The enzyme transforms a substrate into enzyme-substrate and then enzyme-product complexes, which in turn is transformed into a product and free enzyme. See Figure 1.

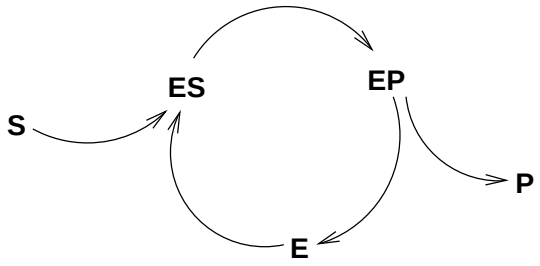

Fig. 1. An example of three-membered catalytic cycle: the free enzyme (E), the enzyme-substrate (ES) and the enzyme-product (EP) complexes all demonstrate a catalytic cyclic restoration of the intermediates in the turnover of the substrate (S) to the product (P).

The cycle as a whole works as a *catalyst*. Unidirectional cyclic restoration of the intermediates presumes a system far from energy equilibrium. This can be associated with a dissipation of energy into the environment. Equilibration occurring in a closed system would cause each individual step to be in balance: catalytic action in such a closed system would not be microscopically irreversible.

Lets now consider a reaction cycle in which at least one of the intermediates themselves is a catalyst (see work of Kaufmann on autocatalytic nets (14)). The simplest representative of this category is a single *autocatalyst* (or a self-replicative unit).

A system which connects autocatalytic or self-replicative units though a cyclic linkage is called a *hypercycle*. Compared with a simple autocatalyst or a self-replicative unit (which we can consider here to be a "flat" structure) a hypercycle is self-reproductive to a higher degree. This is because each of the intermediaries can itself be an autocatalytic cycle.

3.2 Autopoiesis

Following Maturana and Varela terminology (20), machines are unities which are made out of components. All components are characterized by certain properties. Machine components must operate according to certain relationships among their interactions and transformations that define the operation of the machine. The details of properties

other than those participating in the interactions and transformations which constitute the machine are not relevant.

The *organisation* of the machine is defined as all the relations that define a machine as a unity and determine the dynamics of interactions and transformations which it may undergo as such a unity. The organisation of a machine does not specify the properties of the components which realise a concrete machine. The organisation of a machine is independent of the properties of its individual components, which can be any, and a given machine can be realised in many different manners by many different kinds of components.

The *structure* of the machine is defined as the actual relations which hold among the components which realise a concrete machine in a given space. A given machine (machine with fixed organisation) can be realised by many different structures. An organisation may remain constant by being static, by maintaining its components constant, or by maintaining certain relations between components constant which are otherwise in continuous flow or change.

An *autopoietic machine* is defined as a unity by a network of production, transformation, and destruction of components which: (i) through their interactions and transformations continuously regenerate and realise the network of relations that produced them, and (ii) constitute the machine as a concrete unity in the space by specifying the topological domain of its realisation as such a network. An autopoietic machine is an homeostatic or rather a relations-static system that has its own organisation as the fundamental variable which it maintains constant.

In contrast, a machine in which organisation is not autopoietic does not produce the components that constitute it. The products of such a machine are different from the machine itself. The physical unity of such a machine is determined by processes that do not enter into its organisation. Such a machine is called *allopoietic* (20). Allopoietic machines have input and output relations as a characteristic of their organization: their output is the product of their operation, and their input is what they transform to produce this product. The phenomenology of an allopoietic machine is the phenomenology of its input-output relations. The realisation of allopoietic machines is determined by processes which do not enter into the organisation of the machine itself.

Self-organisation An autopoietic system is considered to be a unity in the physical space. It is an entity topologically and operationally separable from the physical background. It is defined by an organisation that consists of a network of processes of production and transformation of components, molecular and otherwise, that through their interactions: a) recursively generate the same network of processes of production of components that generated them; and b) constitute the system as a physical unity by determining its boundaries in the physical space.

The important aspect of an autopoietic system is that it remains invariant in its organisation. The system itself can be deformed by external circumstances, but its internal organisation remains invariant. In other words, the self-organisation and self-maintenance of defining relations is inherent in the autopoietic model. Any change in the autopoietic organisation beyond a particular threshold is equivalent to the loss of identity, and the system disintegration.

Thus, an autopoietic system is defined as a unity by its autopoietic organisation, and all the transformations that it may undergo without losing its identity are transformations in which its organisation remains invariant. All autopoietic systems are therefore homeostatic. They maintain their own organisation constant through their operation. All the various unitary phenomena of an autopoietic system are constitutively subordinated to the maintenance of its autopoiesis.

Hierarchies In conventional Darwinian systems all self-replicative units competing for selection are non-coupled. In other words, the selection forces operate purely on a single level: the level of individuals. This simply leads to a conservation of a limited amount of information, which cannot pass above a specified threshold. In hypercyclic systems, as distinct from conventional Darwinian systems, we deal with similar selective pressures. Note however, that in the hypercyclic case we also deal with integrating properties, and this allows for cooperation of otherwise competing units. Hypercycles are capable of establishing higher-order linkages. When inter-cyclic coupling is established, individual hypercycles may form hierarchies. In other words, the basic unit of selection may not be a single hypercycle, instead a whole chain of interrelated hypercycles. This is an important aspect of our work — exploiting the hypercyclic integrating properties and multi-level selective pressures.

If the autopoiesis of the component unities of a composite autopoietic system conforms to allopoietic roles that through the production of relations of constitution, specification, and order define an autopoietic space, the new system becomes in its own right an autopoietic unity of second order. The most stable condition for coupling appears if the unity organisation is precisely geared to maintain this organisation — that is if the unity becomes autopoietic. Therefore there is an ever present selective pressure for the constitution of higher order autopoietic systems from the coupling of lower order autopoietic unities.

In the theory of autopoiesis, unlike many other theoretical models of the process of life, the process of evolution is simply a side-effect, a consequence of limited resources, not the prerequisite. Whenever we deal with restricted resources, we have the selection and evolutionary pressures naturally occurring within our computational models. It is however important to recognize that life (or precisely: autopoietic systems) would still exist even if the process of evolution were not to occur in a system.

4 Evolvable machine

4.1 Hierachical Computation

Some scholars believe that all sufficiently complicated systems are modelled best by hierarchical models (11; 27; 36). In system sciences and cybernetics any system under investigation is thought of as a composition of multiple subsystems, each of which can itself be decomposed into subsystems, and this follows all the way down to a basic, fundamental level (34). Hierarchies help us deal with complex phenomena by decomposing them into more manageable subsystems and investigating the interactions

between these subsystems, one interaction at a time. The emphasis is placed on investigation of properties on different levels, mutual dependencies, and interactions between and within the hierarchy levels. Hierarchical decomposition of the problem space deals with complexity in a way that is natural and intuitive to humans.

4.2 Virtual machines

Hierarchically organised virtual machines can be used as a specific computation model. Such a model is based on the traditional notion of computing machines, but extends it in certain aspects. The model discussed here provides a flexible and robust platform for experimentation with self-organisation and self-adaptability. It allows for a detailed analysis of different aspects of hierarchical complex system decomposition, together with the analysis of interactions between and within different hierarchical levels. This may help to understand a modelled problem or phenomenon better, giving us at the same time a robust and adaptable computing framework.

Formal definitions The following formalism is inspired by typical models of computing machines. More theoretical foundations for computing models from a programming perspective can be found in (24; 13).

From the Church-Turing Thesis we expect that all models of discrete computation, including the one presented here, will have the same properties as any other model of computation with respect to uncomputability and undecidability. This fact has some interesting and fascinating implications, see e.g. (6). All the well known properties from computational complexity (18) are naturally exhibited by to the computational model presented here. This includes, for example: undecidability, the halting problem, and the concept of non-computable functions.

Definition 1. *A virtual machine or a computing machine (or just a machine for short) is a tuple $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$ where K is the set of states and $s \in K$ is the initial state. Σ_{in} and Σ_{out} are sets of input and output symbols, respectively, referred to as input and output alphabets. δ is a function that maps $K \times \Sigma_{in}$ to $K \times \Sigma_{out}$, and is called the program. We say δ (or the program) runs on machine M. Remember that formally δ is an integral part of the machine itself. The notation $M(x)$ represents the output of machine M given the input sequence x. $M(x, y)$ represents the output of machine M given the input sequence x followed by the input sequence y.*

Definition 2. *Suppose that f is a function from $(\Sigma_{in})*$ to $(\Sigma_{out})*$, and let M be a machine with input and output alphabets Σ_{in} and Σ_{out} respectively (the symbol $*$ has the usual meaning of "set of all possible sequence from a given alphabet"). We say that M computes f if for any string $x \in (\Sigma_{in})*, M(x) = f(x)$. If such machine M exists, f is called a recursive function. We also say that function f is computed by machine M.*

Definition 3. *If for machine $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$ there exists a machine $M' = (K', \Sigma'_{in}, \Sigma'_{out}, \delta', s')$ which computes δ, we call machine M a recursive virtual machine or recursive machine for short. We call program δ' an interpreter of M, and we say an M interpreter runs on machine M'. We have $\forall x \in (\Sigma_{in})*, M(x) = M'(\delta, x)$.*

Definition 4. *Suppose we have a machine $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$ and there exists machine $M_c = (K_c, \Sigma_{in_c}, \Sigma_{out_c}, \delta_c, s_c)$, where $\Sigma_{in} \subseteq \Sigma_{in_c}$ and machine $M' = (K', \Sigma'_{in}, \Sigma'_{out}, \delta', s')$ where $\Sigma_{out_c} \subseteq \Sigma'_{in}$ and $\Sigma_{out} \subseteq \Sigma'_{out}$. If $\forall x \in (\Sigma_{in})*$, $M(x) = M'(M_c(x))$ then we say that δ_c is an M compiler, and we say M_c compiles M into M'.*

The emphasis in the conceptual framework presented above is to treat algorithms and running programs as *machines* (*recursive virtual machines* to be precise). This along with the notions of compilers and interpreters is discussed at length in (13). The above definitions do not make any assumptions about the number of states a given machine can have, nor about the storage capability. All possible models of computations, and different computer/algorithm architectures fit the above definitions. For example one could use $\Sigma \subseteq Real$ to perform analog computation on real values. It can be shown that the proposed conceptual framework is a simple extension of the theoretical models of computation such as Turing machines and Universal Turing machines (12; 24).

Proposition 1. *Let machine $M = (K, \Sigma_{in}, \Sigma_{out}, \delta, s)$, with finite input and output alphabets $\Sigma = \Sigma_{in} = \Sigma_{out}$, $\{\sqcup, \triangleright\} \in \Sigma$ and $\{h, y, n\} \in K$. In other words the alphabet contains two special symbols, the blank and the first symbol, and there are three extra state symbols, namely: h the halting state, y the accepting state, n the rejecting state. We define three additional symbols, representing cursor directions: $\leftarrow$ for "left" and $\rightarrow$ for "right" and $-$ for "stay". If δ maps $K \times \Sigma$ to $K' \times \Sigma$, where $K' = K \times \{\leftarrow, \rightarrow, -\}$ then we say that machine M is a Turing machine.*

5 The architecture

We can model artificial and naturally occurring phenomena as a chain of virtual machines. One possible perspective on artificial life or evolutionary systems is to focus on a tower of compilers and/or interpreters. The concepts of chaining and stacking compilers and interpreters is discussed in detail in (13). The other approach is to use more traditional functional decomposition. All computing programs, including all evolutionary computation models can be represented as a chain of compilers and/or interpreters, with different functional partitioning on each level. The way this chain is constructed and how all its elements interact with each other is a principal concern of our hierarchical computing architectural approach.

5.1 Vertical and horizontal decomposition

Following the formal definitions, a machine can be statically represented as a program string, consisting of a prefix, together with some instructions following this prefix. The prefix itself can be decomposed into another prefix and another program, and so on. This is called *vertical decomposition*, or a *vertical hierarchy*. Another type of decomposition is based on dividing a given machine into interacting parts – this is called a *horizontal decomposition*. Formally, a vertical hierarchy is based on stacking interpreters and/or compilers (13), see Figure 1. A horizontal decomposition is based on splitting a single machine into two or more machines, see Figure 2.

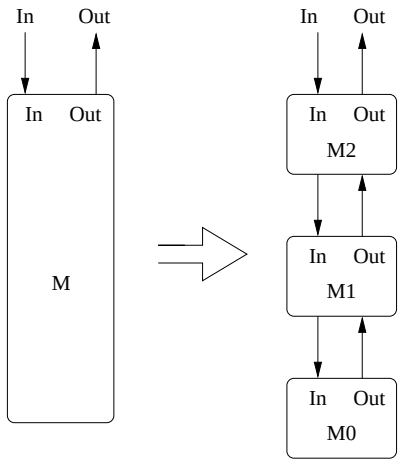

Fig. 2. A vertical split of machine M into a tower of machines M0, M1, M2.

Existing examples of vertical hierarchies are all sorts of (real-life) interpreters and compilers. For example given a Pascal interpreter written in Java we would have: program written in Pascal $\longrightarrow$ Pascal virtual machine (written in Java) $\longrightarrow$ Java virtual machine (written for example in C) $\longrightarrow$ C virtual machine $\longrightarrow$ etc., where the arrow reads as "runs on" as defined in Definition 3.

An example of horizontal partitioning would be a functional partitioning of a single individual virtual machine. Let us imagine that we have a machine that can compute two operations on the natural numbers domain: addition and multiplication. If we perform functional partitioning, we can end up with two virtual machines, each computing a single operation, multiplication or addition, respectively. The union of these two gives us the original single machine.

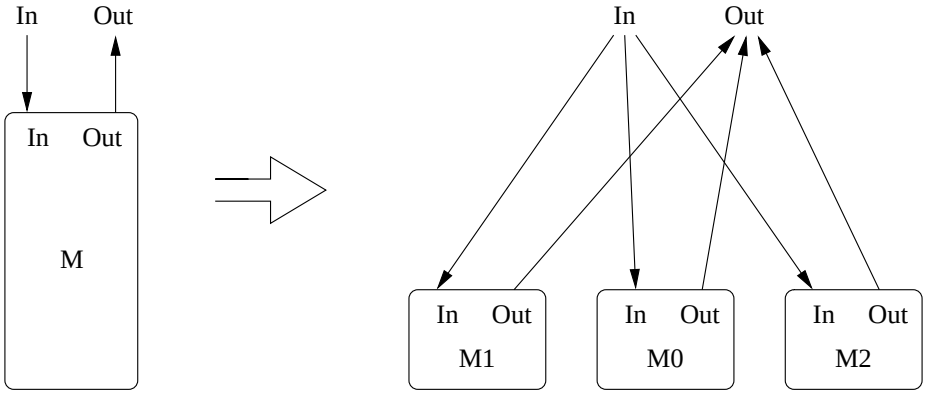

Fig. 3. A horizontal split of machine M into machines M0, M1, M2.

One can enumerate through all the machine levels, starting from the base (fundamental) level M_0, up to the final highest-level machine, M_n. The actual input (instructions) are fed to the machine M_n. It is important to remember that, in fact, there is no special distinction between the *program* running on a virtual machine and the program emulating a particular machine itself.

All the interacting virtual machines are connected by their input/output streams. The hierarchical structure of that composition can have different forms, depending on the particular phenomena at hand. It can be a simple linear structure, or it can be a tree-like structure. In general it is a directed graph, with cycles, with self-referencing nodes, and possibly with complicated interdependencies (see Figure 5.1).

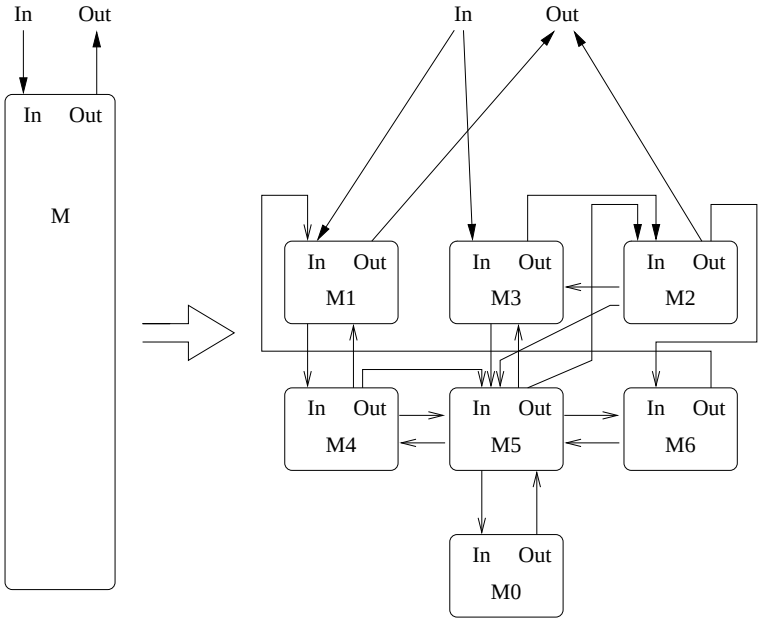

Fig. 4. The example of possible dependencies between machines after decomposition

5.2 Partial equivalence

Some machines can be fully or partially equivalent to others; for example a Pascal virtual machine written in C and a second one written in Java are always perfect and fully equivalent Pascal virtual machines, even though they use completely different machines on the lower level. Note that even though these two Pascal virtual machines have different machines below them, they can have exactly the same virtual machine one level down, for example a virtual machine for a particular operating system.

One can have a partial Pascal virtual machine that accepts a subset of all possible programs generated in Pascal. This is referred to as *specialisation*. On the other hand

it is also possible to have a Pascal virtual machine accepting a subset of expressions from the C language, in addition to normal Pascal programs. The process of adding features to the language and enhancing the input language for a given machine is called *conservative extension* (13).

Some machines can be recursively executed on themselves, for example a Java virtual machine interpreter written in Java and executed on a Java virtual machine interpreter. Some machines can be functionally equivalent even though they use completely different language syntaxes or alphabets, for example expressions in prefix, postfix or reverse Polish notations. All these properties are well known in computer science, in which specific languages, interpreters and compilers flourish.

Suppose the problem at hand is coded in such a way that the solution can be expressed as a string of symbols from some language, L. For some languages finding a solution string is easier than for others: coding the problem is the key issue in solving the problem. In a sense the language L captures and exploits some of the properties of the problem. This is one of the main features of the proposed approach. With recursive virtual machines we have the necessary framework to model the transformations of a problem representation from one language to another, and we are able to translate the original problem into a more easily solvable equivalent.

5.3 Decomposition limits

A particular level from the hierarchy is treated as a virtual machine that provides some functionality to the other level immediately above it, and uses the level below to have the computation performed. In other words a particular machine accepts input from one level, uses other levels to perform computation, and then returns the results back to yet another adjacent level. The highest level of the chain of machines accepts some input (instructions), interacts with the level below it by sending/receiving some input/output, and returns some outputs (results) back. Similarly to the base level, what we consider the highest level is also arbitrary. There is always a virtual machine feeding the instructions and accepting the results (e.g. a computer program or a human operator).

A given machine in a chain is formally equivalent to an interpreter or compiler of another machine located above it. The first, the base level is the very first interpreter, which we assume as being executed on some universal virtual machine (UVM). In the case of digital computers (and for the sake of simplicity) we can without loss of generality assume that the base level machine is equivalent to the Universal Turing Machine (12). Of course, this is an arbitrary choice, and the decomposition could be carried further, treating the UVM itself as a virtual machine, running on some software/hardware platform and so on, all the way down to electrical and/or chemical reactions and some physical processes[1].

[1] Actually, according to (9) we have no reason to stop there, and we can decompose the system further, based on the idea that physical phenomena itself are running on some (digital, in the case of Fredkin's theory) virtual machine.

6 EVM implementation

6.1 Yet another language?

There exist many programming languages developed within the field of Evolutionary Computation. Many employ usual higher level programming languages designed for human programmers (such as Lisp for the original formulation of tree-based Genetic Programming (16)); some are developed with an evolutionary process in mind (35; 28), and others are developed for machine processing and recursive program manipulations (31). Some of the languages are highly specialized, and provide the evolutionary mechanisms with a bias towards a particular solution subspace. However, none of these languages provides mechanisms to manipulate levels – a property needed for our EVM implementation. There are other features we want our base machine language to possess, that none of the existing languages have. For example, we want the language to be capable of redefining itself. That is, the primitive instruction set must allow the evolutionary process to restructure and redefine itself. Also, we want a programming language that is highly expressive, that is, we want solution programs to typically encountered tasks to be short. And also, we believe that there are efficiency advantages for a language whose solution spaces are highly recursive.

A programming language used for search in Evolutionary Computation plays an important role – some programming languages are particularly suited for some, but not for all, problems. One of the appealing aspects of a multi-level search process is that, in principle, it can define a new base level and a completely new programming language that is specialized for the given task at hand. We want to exploit this property.

Some of the existing languages possess some of these desired properties, but no single one of them possesses all of them. This is why we have designed our own specialized programming language. The principal objective of the overall programming language is to facilitate searches for specialized languages for a given set of problems, and we want the EVM to facilitate that. Even though there is currently a concrete implementation of the base machine for the EVM system (the primitive instruction set), we treat it only as a temporary list. We are working on redesigning the base machine to better suit and to help with the search of program generators. So far, we have obtained some results suggesting the need of some of more computationally intensive primitive instructions to be included into the base machine. On the other hand, some of the existing instructions are rarely being used, and will be removed in the next iteration of our implementation.

6.2 Computing model

The hierarchical computing model presented in the previous sections can be implemented in multiple ways and in many physical programming languages. It should be understood that the implementation presented below is only one of many possible implementations, and the choice for this particular implementation as distinct form other computing architectures, is somewhat arbitrary. On the other hand, we have paid considerable attention in order to make the implementation as flexible and robust as possible

and to facilitate different configurations and different experiments in order to fine-tune the instruction set and the overall computing architecture for general-purpose use.

Our initial implementation of the EVM architecture is based on a stack-machine, such as Forth (22), or Java Virtual Machine (JVM) (37). In fact, with small differences, it is exactly the same as an integer-based subset of a JVM.

The main architectural component, similar to the JVM, is the so called *execution frame*. The schematic view of the execution frame is presented on Figure 6.2.

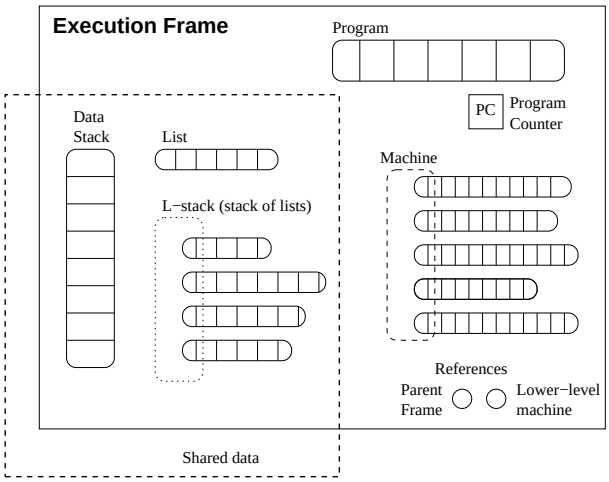

Fig. 5. Schema of the execution frame

The basic data unit for processing in our current implementation is a 32-bit signed integer. The basic input/output and argument-passing capabilities are provided by the operand Stack, called here *Data Stack*, or for short *Stack*. Data Stack is a normal integer stack, just as in a JVM for example. All the operands for all the instructions are passed via the Stack. The only exception is the instruction `push`, which takes its operand from the Program List itself. Unlike the JVM, our virtual machine does not provide any operations for creating and manipulating arrays. Instead, it provides instructions for and facilitates operations on lists. There is a special stack, called *L-stack* for storing integer-based lists. The L-stack is implemented as an a-stack (a-stack is a special way of implementing stack, such that its top element is stored as a special register/variable). The L-stack top element is stored in a special list called *List*. In other words, *List* contains the "actual" top element of the L-stack, and the top element that is on the stack of the L-stack is the second topmost element, and so on. The decision to implement L-stack based on a-stack is purely for efficiency purposes. However it also makes all the instructions that operate on a List more natural and more intuitive, as they operate on an actual list, not on the top element of the L-stack. Both, Data Stack and L-stack (together with the List), are shared between multiple Execution Frames that share a common thread of execution.

There is a lower-level machine handle attached to each of the execution frames. This is a list of lists, where each individual list represents an implementation of a single instruction for the given machine. So, in other words, the machine is a list of lists of instructions, each of which implements a given machine instruction. Of course, if the given instruction is not one of the Base Machine units, the sequence must be executed on another lower-level machine. The Base Machine implements base instructions that are not reified further into more primitive units. These instructions will be discussed later in this section.

The program in the Execution Frame is represented as a list of integers. The program counter (PC) points to the current instruction in the program list. The PC is itself an integer, and that limits the theoretical length of any single program to 2^{31}. This is however lowered by the maximum length of any list in the system, which is currently set to 100000. Each instruction in the program list points to the appropriate instruction in the machine. If the value of a given instruction in the program is bigger than the number of instructions in the given machine, the index of the instruction is calculated as $index = instruction \ mod \ machine_{size}$, where $machine_{size}$ is the number of instructions in a given machine.

Each Execution Frame can reference the parent frame. The parent frame is responsible for creating and initializing the given frame. Base-level frames are those top-level frames that do not have any parent frame (the reference is $null$).

6.3 Execution model

As described in the previous subsection, the EVM program is represented within the Execution Frame as a list of integers. Each integer (modulo Machine number-of-instructions) points to an individual machine instruction, that implements a particular behaviour of a given program instruction.

There are two possible situations. First, the instruction of a machine may be a primitive instruction, or *EVM operation*. In that case, the execution of a program instruction is simple: the behaviour just happens within the current Execution Frame. For example, if our program contains an integer, that points to `add` operation on the Base EVM Machine, this operation will take two arguments from the Stack, will add them together, and will put the result back on the Stack.

The second situation is when the instruction of the machine is a composite instruction, i.e. a list of instructions for lower-level machine. In that case, the execution of a program instruction proceeds as follows. First, a new Execution Frame is created. This new Execution Frame is initialized with the Stack of the parent Execution Frame (all Execution Frames in the same thread of execution share the same stack for parameter and return value passing). The PC of the new Execution Frame is set to `zero`, and the Program List is set to the Machine Instruction List. Then, the control is passed to this new Execution Frame, which executes this “subprogram”. Once done, the control is switched back to the parent Execution Frame.

The EVM is Turing equivalent, therefore there exist EVM programs that can run indefinitely. Each thread of execution has an instruction time limit, to constrains the time of each program in a multi-EVM environment. That is, each execution thread

(single program) has a maximum number of primitive instructions that it can execute. Once the limit is reached, the program halts.

6.4 Instruction set

As noted before, the instruction set is modelled after the Forth (22), and Java Virtual Machine (37) instruction sets.

The instruction set is divided into several categories, which we describe here briefly. The first category is "Stack and general operations". This includes pushing constants onto the Stack, popping, swapping, rolling, duplicating, etc. Some of the example instructions are: `const1, const0, pop, swap, roll, dup`.

The second category is L-stack operations. There are operations for appending, removing, and manipulating lists on the L-stack. It also contains operations to transfer lists between L-stack and Stack. Some of the example instructions are: `lpop, lpopn, lswap, ldup, ldepth`.

The third category contains List operations, and includes transferring elements between List and Stack, and manipulating List elements. Some of the example instructions are `prepend, append, load, store, length, rmf, rml, rm, rmn`.

The fourth category contains operations for manipulating the Machine list of lists. This includes similar operations to L-stack operations, but here they refer to Machine. Some example instructions are: `mappend, mprepend, mload, mstore, mrmf, mrml, mrm, mins`.

The fifth category comprises the three level-related operations. These are `spawn`, `up` and `down`. We will discuss them in more detail below.

The sixth category contains all the control instructions. This list is based on the JVM control operations, and contains the following instructions: `ifeq, ifneq, iflt, ifle, ifgt, ifge, goto, jmp`. There are three extra instructions. `exec` takes the content of the List, and instantiate new Execution Frame and executes as if List is a program to be executed. This is equivalent of executing dynamically created subroutine. The other two instructions are two "search and jump" instruction. They take the element from the operand stack, and search forward (`jmpsf`), or backward (`jmsb`) to find same element in the program list and jump to the next instruction following that element.

The seventh and eighth categories contain all the Logic and Arithmetic operations. Logic operations are: `shl, shr, ushr, and, or, xor,` and `not`. Arithmetic operations are: `add, inc, sub, dec, mul, div, rem,` and `neg`.

6.5 Multi-level computation

The key feature of the EVM, apart from its clean and elegant "zero operand architecture" (8), is that it offers multi-level processing. This is like having unrestricted reflection and reification mechanisms built-in for the virtual machine itself. The computing model is relatively fixed at the lowest-level, but it does provide the user with multiple computing architectures to choose from. The model allows the programs to reify the very virtual machine on the lowest level. For example programs are free to modify, add, and remove instructions from or to the lowest level virtual machine. Also, programs can

construct higher-level machines and execute themselves on these newly created levels. Not only that – a running program can switch the context of the machine, to execute some commands on the lower-level, or on the higher-level machine. All together it provides unlimited flexibility and capabilities for reifying EVM execution.

Let us consider a particular example. Imagine, that we are tasked with writing a program to add a given number N of integers. All N integers are provided on the Data Stack, and the result is expected to be on the stack. We make it an incremental problem, by iterating from $1 \dots N$. On the base machine we only have arithmetic operations, such as `add`. The `add` operation takes two arguments from the data stack, adds them together, and puts the result back on the top of the stack. The task of adding two numbers can be solved by a program with one instruction `add`. The task of adding three numbers requires two `add` instructions, and so on. If we generalise it for N, the simplest program would look like this:

```
add add add ... add   /* (nth-1 instruction) */
```

Given that the solution for N would be provided as a prefix for the program that must solve the task for $N+1$, the probability of randomly generating the postfix code for the $N+1$ problem would be $1/|BM|$, where $|BM|$ represents number of instructions in the base level machine.

With the multiple-levels, the prefix however can specialise a higher level machine for the "adding numbers" problem. This can be easily achieved by creating a higher level machine, with only one instruction, that adds two numbers. The program would look like that:

```
push add   /* pushes code for ADD instruction */
depth      /* pushes 1 on the stack */
popn_l     /* appends 1 element into the list*/
const_1    /* pushes 1 on the stack */
spawn      /* creates a higher level machine */
up         /* changes to the higher level */
0  ...  0  /* instruction repeated nth-1 times */
```

Note, that it takes only 7 instructions to construct and switch to the higher level machine, whose sole purpose is number addition. The higher level machine is a specialized machine that can only add numbers. It does not matter what the $N-1$ instructions are that are actually appended to the end of the program. The program will always correctly add N numbers. In this case, the probability of solving the $N+1$ problem becomes 1, as any of the added instructions would map to the single instruction on the higher level machine.

Of course, this is an extremely simple case, but it demonstrates the specialization capabilities of a multi-level computing machine. The governing idea is to create a custom specialized language and to solve the problem in that language, instead of trying to solve it in the original language of the base machine.

6.6 EVM and its expressiveness

We were inspired by the expressive power of a simple programming language designed by Schmidthuber (31). However, we noted, that some of the recursive functional constructs he introduced in his language could be done more simply (i.e. making them shorter) in our own language for EVM.

For example, in Schmidhuber's language a recursive call to define and to calculate the factorial of N, assuming N is placed on top of the data stack, takes 14 instructions and has the following form:

```
c1 c1 def up c1 ex rt0 del up dec topf dof mul ret
```

In our language it is only 8 instructions, and the program looks like this:

```
dup halt0 dup dec lpush_p mappend mlcall mul
```

Note, our `dup` is equivalent to Schmidhuber's `ex`, `halt0` is equivalent to `rt0`, `mdepth` to `topf`, `mlcall` to `topf` and `dof` executed together. `lpush_p` copies the current program list into the List, and `mappend` appends the newly defined program from the List as a new primitive instruction to the base machine. `mlcall` executes the last instruction from the current machine instruction list. Our program is shorter, because (a) we do not need to define the number of arguments and result values, (b) we do not need to explicitly call `ret`, and (c) in our language defining a new program based on the current program in the List takes one instruction, and based on the program in the Program List, takes only two instructions.

Similarly, in the case of a context free grammar problem described by Schmidhuber (31), his solution is 5 instructions long, and looks like this:

```
defnp c1 calltp c2 endnp
```

Our code takes only 3 instructions:

```
c1 rwhile c2
```

`rwhile` is a "recursive while" instruction, that works in the following way: it checks the top of Data Stack; if there is value 0, it halts, otherwise, it forks to a recursive call back to the original program.

Because the EVM is more expressive, any search method, including Schmidhuber's optimal problem solver, should find the appropriate solutions faster.

6.7 Program ontogeny

Let us view a program input as a sequence of integers on the Data Stack, and consider a single program loaded into the Program List in the Execution Frame. Both, the input, and the program, can be divided into two subsequences indexed 1 and n, in such a way, that the first subsequence of a program, P_1 (program with index 1) reads all the data from the D_1 subsequence. P_1 will produce some results on the Data Stack, and it can also manipulate the program list itself. Hence, the remaining subsequence on the Data Stack is now longer, and the remaining program on the program list may differ from

the original program list. If this process is repeated recursively, the final program, and the final data that this program reads, will be remapped from whatever was originally on the Data Stack, and inside the Program List. This process is referred to as Program Ontogeny. It demonstrates the development of the final (mature) stage of a program from some initial (larval) stage, through a sequence of transformation steps.

6.8 EVM and hypercycles

In the current implementation of the EVM architecture, we employ two initial designs that facilitate the hypercyclic dependencies. One of them is based on the notion of self-replication of the EVM programs. The other is based on the notion of cyclic data flow. They each, in a way, complement each other. We will describe them below based on simple examples.

Self-replication. If a given program produces an output, and this output is identical to the program that produced it, we have a self-replicating EVM program. In other words, we have a program that can calculate (produce) itself. If a program P_1 produces another program P_2, such that P_2 is not equivalent to the original program P_1, but in turn, P_2 produces a program P_1, than we have a a hypercycle. Depending on the complexity of each of the individual programs, and their ontogeny, it may exhibit interesting autopoietic dependencies.

Data flow cycles. Each program within a multi-EVM environment fulfills its function in a narrow spectrum of data inputs, and produces its outputs again, in relatively narrow spectrum out of all possible outputs. For example a solver for "n-addition problem" cannot be given different input than it expects, otherwise it will not work as an "n-addition" problem solver. However, if the output of P_1 is connected to the input of P_2, and the output of P_2 to the input of P_1, then we have a cycle. If the cycle keeps the data flow within expected and desired ranges of values, we have an autocatalytic hypercycle. Together with the actual programs they represent an autopoietic system.

7 Self-organisation by means of evolutionary computation

7.1 Requirements

There are some inherit properties that the self-adaptive and self-organising software system may exhibit. These properties facilitate effective processes to help and guide evolutionary mechanisms. Our current EVM implementation facilitates some of these properties.

Split and splice. It is desirable that different individual functional units are freely manipulated. It means that one can put different components together, and then split them apart, always producing valid functional units within the system. This is supported by the EVM. Each program can be cut in the middle, and the parts will always form valid programs. Programs can be joined together, always producing valid programs. Actually, any sequence of integers is a valid program in a EVM.

Cyclic behaviour. All individual components of the software system must carry out their activities in a cyclic manner. That means that the functionality is organised in such a way that tasks are repeated over and over again so that tuning, self-organisation, and adaptability can take place. If a given task were to be designed to be performed only once, there would be no room for improvement, since the given component would only have a single opportunity to perform.

Many agents on many levels. There are benefits from having many independent interactive components acting on many different levels. Reflection and recursion among components can facilitate shorter and more robust solutions to given tasks performed by components of the system. Some components will just perform tasks, some will monitor others performing tasks and provide necessary feedback for improvement, and others will improve the "improvers", etc.

The system must be open to external signals. This simply means that the system has to interact with the "outside". A software system which does not exchange any information with the environment "outside" the system itself cannot evolve into a more complex system than the original one. Without being exposed to new information the system can only refine itself, and is unable to acquire new capabilities. Such information must be provided from the outside environment.

7.2 Evolving recursive virtual machines

The field of evolutionary computation is mainly based on experimentation, and so far it is primarily a trial and error approach. In light of all the advances in theoretical computer science and given the conceptual framework of recursive virtual machines, it is now possible to introduce a more systematic approach. Within EVM, each different evolutionary system is an example of a virtual machine, each language is an example of a different search space, and each system is an example of the interplay between different aspects of the hierarchical organisation.

Probably one of the closest existing systems using the concept of a virtual machine in the form of a hierarchy is the grammatical evolution system (28). In this system, a top-level search is performed on strings of integers. A string containing integers is fed into a particular machine to produce a computer program coded in a particular language as output. This code is then fed as input to yet another machine, which in turn returns a final result. Each of the levels is relative to the level below it; this relativity means that the same top-level string of integers will produce a completely different result when used in combination with another machine. The top-level machine accepting the strings of integers is designed in such a way that it can "plug-in" to any possible second-level machine, and the model will still work. This is a human designed feature, but it is inspired by many naturally occurring phenomena. The multiple levels of indirect influences seems to be the most powerful mechanism at work here.

Instead of designing such machines, and all the indirection levels, by hand, we believe that with our approach this process can be automated, and the virtual machine suitable for a particular class of problems can be discovered automatically.

7.3 Seeds and solution growing

Let us take a grammatical evolution system (28) as an example of the solution growing concept. The solution for a problem at hand is effectively a proper hierarchy of machines (in this case a BNF-encoded language grammar) and a string of integers as a symbolically encoded solution, which we refer to as a *seed*. In the case of a grammatical evolution system, the hierarchy of machines is designed by a human programmer before the search for the proper seed is started. However, the hierarchy of machines needs to be discovered as well. sought-after solution itself.

In general, the solution to the problem (finding a computer program) will be a hierarchy of machines together with the seed. The actual computer program is then generated by feeding the seed through the system. In the case of grammatical evolution, speaking informally, the generation process is (in order): feeding the string of integers, generating the program listing, running the program for the given input, and then obtaining the final solution. The given input in this case depends on the "outer-level" virtual machine.

It is, however, possible to change or modify the machine hierarchy just before generating the computer program. If the hierarchy of machines, their connections and the initial states are subject to change, we refer to the process of generating a final solution as *solution growing*. In the case of searching for code, one can use the term *code growing* instead. It is possible, by varying the hierarchy of machines, to grow a valid solution from the same seed for a certain variation of the original problem. By simple re-mapping, one can achieve exactly the same result by varying the structure of the seed itself. This opens a new window of opportunities not yet used by the automatic code generation techniques. Again, it is a very commonly occurring phenomenon in nature.

Formally the idea of *code growing* is based on the notions of *bootstrapping* and *self-application*. This is analogous to more traditional compiler/interpreter bootstrapping and self-application (13).

8 Summary

An architecture of dynamic hierarchically organised virtual machines as a self-organising computing model has been presented. It builds on Turing-machine-based traditional models of computation. The model provides some of the necessary facilities for open-ended evolutionary processes in self-organising software systems. It allows stacking machines (vertical decomposition) in addition to more traditional functional hierarchical decomposition models. It can be used as a more systematic approach to different code generation techniques and self-adaptable software. Unlike existing models, the emerging levels of organisation can be either modelled directly as individual machines or can be indirectly captured for a formal analysis as a state of an individual machine.

Applications using the proposed architecture are possible and are planned as future work. Also, the formal model presented here allows for the preparation of an operational definition of a living system. However, further formalization of the framework is necessary, and is currently under investigation by the authors.

Bibliography

[1] Leon Brillouin. *Science and information theory*. Academic Press Inc., 1956.

[2] Gregory J. Chaitin. To a mathematical definition of 'life'. *ACM SICACT News 4*, pages 12–18, January 1970.

[3] Gregory J. Chaitin. Toward a mathematical definition of "life". In R. D. Levine and M. Tribus, editors, *The Maximum Entropy Formalism*, pages 477–498. MIT Press, 1979.

[4] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, 1859.

[5] Manfred Eigen and Peter Schuster. *The Hypercycle: A Principle of Natural Self-Organization*. Springer-Verlag, 1979.

[6] Gary William Flake. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. MIT Press, January 31 2000.

[7] David B. Fogel, editor. *Evolutionary Computation – The Fossil Record*. IEEE Press, New York, USA, 1998.

[8] Jeff Fox. Multiple Stack Zero Operand Computers Today. *Free Software Magazine*, (05), 2002.

[9] Edward Fredkin. A new cosmogony: On the origin of the universe. In *PhysComp'92: Proceedings of the Workshop on Physics and Computation*. IEEE Press, 1992.

[10] Andrzej Gecow and Antoni Hoffman. Self-improvement in a complex cybernetic system and its implication for biology. *Acta Biotheoretica*, 32(1):61–71, 1983.

[11] John H. Holland. *Adaptation in Natural and Artificial Systems : An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, reprint edition edition, April 29 1992.

[12] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley Publishing Company, USA, 1979.

[13] Neil D. Jones. *Computability and Complexity: From a Programming Perspective*. MIT Press, 1997.

[14] Stuart A. Kauffman. *The origins of order: self-organization and selection in evolution*. Oxford Press, New York, USA, 1993.

[15] M. Kokar, K. Baclawski, and A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, pages 37–45, May-June 1999.

[16] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[17] Leonid A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[18] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, New York, second edition, 1997.

[19] Lynn Margulis and Dorion Sagan. *What is life?* New York, Simon and Schuster, 1995.

[20] Humberto R. Maturana and Francisco J. Varela. Autopoiesis: The organization of the living. In Robert S. Cohen and Marx W. Wartofsky, editors, *Autopoiesis and Cognition: The Realization of the Living*, volume 42 of *Boston Studies in the Philosophy of Science*. D. Reidel Publishing Company, Dordrech, Holland, 1980. With a preface to 'Autopoiesis' by Sir Stafford Beer. Originally published in Chile in 1972 under the title De maquinas y Seres Vivos, by Editorial Univesitaria S.A.

[21] Alex C. Meng. On evaluating self-adaptive software. In Paul Robertson, Howie Shrobe, and Robert Laddaga, editors, *Self-Adaptive Software*, number 1936 in LNCS, pages 65–74. Springer-Verlag, Oxford, UK, April 17–19 2000. IWSAS 2000, Revised Papers.

[22] Charles H. Moore and Leach Goeffrey C. FORTH – A language for interactive computing. Technical report, Amsterdam NY: Mohasco Industries, Inc., 1970.

[23] L. E. Orgel. *The Origins of Life: Molecules and Natural Selection*. Wiley, New York, 1973.

[24] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc., 1994.

[25] Ilya Prigogine and Isabelle Stengers. *From being to becoming*. San Francisco, W. H. Freeman., 1980.

[26] Martin Purvis, Stephen Cranefield, Geoff Bush, Daniel Carter, Bryce McKinlay, Mariusz Nowostawski, and Roy Ward. The NZDIS Project: an Agent-based Distributed Information Systems Architecture. In Jr. R.H. Sprague, editor, *In CDROM Proceedings of the Hawaii International Conference on System Sciences (HICSS-33)*. IEEE Computer Society Press, 2000.

[27] Justinian P. Rosca. *Hierarchical learning with procedural abstraction mechanisms*. PhD thesis, University of Rochester, Rochester, NY 14627, USA, 1997.

[28] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of EuroGP 1998*, pages 83–96. Springer Verlag, 1998.

[29] J. Schmidhuber. The speed prior: a new simplicity measure yielding near-optimal computable predictions, 2002.

[30] Juergen Schmidhuber. A general method for incremental self-improvement and multiagent learning. In X. Yao, editor, *Evolutionary Computation: Theory and Applications*, chapter 3, pages 81–123. Scientific Publishers Co., Singapore, 1999.

[31] Juergen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–254, 2004.

[32] Erwin Schrödinger. *What is life? : the physical aspect of the living cell*. Cambridge, University Press, 1945.

[33] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[34] H. A. Simon. *The sciences of the artificial*. MIT Press, 1968.

[35] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.

[36] Lee Spector. Hierarchy helps it work that way. *Philosophical Psychology*, 15(2):109–117, June 2002.

[37] Bill Venners. *Inside the Java Virtual Machine*. Mc-Graw Hill, second edition edition, 1999.

[38] John Luis von Neumann. The general and logical theory of automata. In A. H. Taub, editor, *John von Neumann – Collected Works*, volume V, pages 288–328. Macmillan, New York, 1963.

[39] John Luis von Neumann and Arthur W. Burks. Theory of self-reproducing automata, 1966.

[40] Michael D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. A Bradford Book, MIT Press, Cambridge, Massachusetts/London, England, 1999.

[41] Sewall Wright. Evolution in mendelian populations. *Genetics*, 16(3):97–159, March 1931.