Self-Adaptation and Dynamic Environment Experiments with Evolvable Virtual Machines

Mariusz Nowostawski[1] and Lucien Epiney[2] and Martin Purvis[1]

[1] Department Information Science
University of Otago, Dunedin, New Zealand
(mnowostawski,mpurvis)@infoscience.otago.ac.nz
[2] Swiss Federal Institute of Technology
EPFL, Lausanne
lucien.epiney@epfl.ch

Abstract. Increasing complexity of software applications forces researchers to look for automated ways of programming and adapting these systems. Self-adapting, self-organising software system is one of the possible ways to tackle and manage higher complexity. A set of small independent problem solvers, working together in a dynamic environment, solving multiple tasks, and dynamically adapting to changing requirements is one way of achieving true self-adaptation in software systems. Our work presents a dynamic multi-task environment and experiments with a self-adapting software system. The Evolvable Virtual Machine (EVM) architecture is a model for building complex hierarchically organised software systems. The intrinsic properties of EVM allow the independent programs to evolve into higher levels of complexity, in a way analogous to multi-level, or hierarchical evolutionary processes. The EVM is designed to evolve structures of self-maintaining, self-adapting ensembles, that are open-ended and hierarchically organised. This article discusses the EVM architecture together with different statistical exploration methods that can be used with it. Based on experimental results, certain behaviours that exhibit self-adaptation in the EVM system are discussed.

1 Introduction

Existing evolutionary computation techniques, such as genetic programming (GP) [3], linear genetic algorithms (GAs) [13], and others [2], have proved to be successful in a broad range of optimisation problems and applications. These methods, however, are operating on a predefined, fixed fitness landscape and therefore are very difficult or even impossible to be used in multi-task dynamical environments. In this article we propose a new model of evolutionary computation that can be used in highly dynamic environments. Moreover, our model can be used with traditional linear GA evolutionary learning, with random search, and with many other stochastic search methods. Our framework consists of a set of independent computing cells that compete for limited resources. These computing cells are able to dynamically change their functionality and functional dependency to meet changes in their environment. They form a web of interacting computational agents that exhibit self-organisation and self-adaptability without direct user interaction.

2 Computation and biological inspirations

Current research in EC emphasises information-centric methods that are inspired by 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 [1; 17].

The Evolvable Virtual Machine architecture (EVM) is a novel model for building complex hierarchically organised software systems. In this article we describe how the original abstract EVM model [9] has been extended by the elements of symbiogenesis, that allow independent computing elements to engage in symbiotic relationships.

From the biological perspective, the abstract EVM model is primarily based on Darwin's principle of natural selection, which is widely used in current computational models of evolutionary systems for optimisation or simulation purposes, and in evolutionary computation (EC) 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 [1].

Our work proposes an evolutionary model inspired by the theory of hypercycles [1], autopoiesis [7], and symbiogenesis [8]. A system which connects autocatalytic or self-replicative units through 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.

The EVM system consists of an interconnected network of processing units (cells or agents) that can only interact with their neighbours. These processing units are autonomous, they do not have pre-assigned functions, can specialise in different tasks and can utilise the processing structures of their neighbours. Initially, each single processing unit potentially benefits from Universal Turing Machine (UTM) equivalent computing capabilities. In time, some of the cells can specialise in tasks requiring different (lower) computing power, i.e. specialisation of the virtual machines occurs. On the other hand, each processing unit can make use of other processing units, via a symbiosis-like relationship, thus creating a web of interconnected machines.

2.1 Symbiogenesis and specialisation

Proponents of symbiogenesis argue that symbiosis is a primary source of biological variation, and that acquisition and accumulation of random mutations alone are not sufficient to develop high levels of complexity [5; 6]. K. Mereschkowsky [8] and I. Wallin [14] were the first to propose that independent organisms merge (spontaneously) to form composites (new cell organelles, new organs, species, etc). For example, important organelles, such as plastid or mitochondria, are thought to have evolved from an endosymbiosis between a Gram-negative bacterium and a pre-eukaryotic cell.

Another (less speculative) phenomenon that occurs at all levels of biological organisation from molecules to populations, is specialisation. It is the process of setting apart a particular sub-system (reducing its complexity) for the purpose of better performance and/or efficiency of a particular function. Our working hypothesis is that specialisation, together with symbiosis, is necessary to reach higher complexity levels.

Some recent work in incremental reinforcement learning methods also advocate the retention of learnt structures (or learnt information) [10]. The sub-structures developed or acquired during the course of the program self-improvement process are retained in the program data-structures. It is therefore surprising that this general procedure has not been exhibited by any of the (standard) evolutionary programming models, such as GP or GAs [13]. 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 an incremental self-improvement fashion.

2.2 Abstract self-organising application architecture

Simplified, the model presented here can be depicted schematically as in Figure 1, with inputs and outputs connected to the external environment. The environment consists of a set of current tasks to be solved, with an appropriate resource pool associated with each task. The environment also keeps track (via the tasks) of the current resource utilisation for each given task. The user can dynamically plug-in new tasks, and remove existing ones, at will. In Figure 1 the resource marked as `R1`, is utilised by 5 computational cells (cells with same gray hue on the grid). Similarly, the other two resources `R2` and `R3` are utilised by cells indicated by other matched shading types.

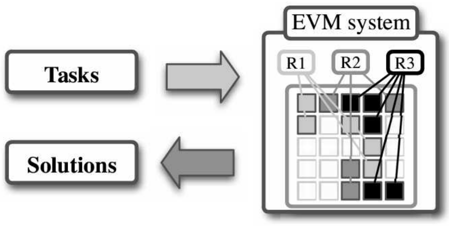

Fig. 1. EVM system from an end user point of view.

3 EVM – Evolvable Virtual Machines

The programming language used for search in EC plays an important role. Some languages are particularly suited for some, but not for all, problems. An appealing aspect of a multi-level search process is that, in principle, it is possible to specify a new base level and a new programming language that is specialised for a given task at that level. We want the EVM to exploit this property.

In our work we deal with programs capable of universal computation (e.g. with loops, recursion, etc.). In other words, the virtual machine running our programs must be Universal Turing-machine equivalent.

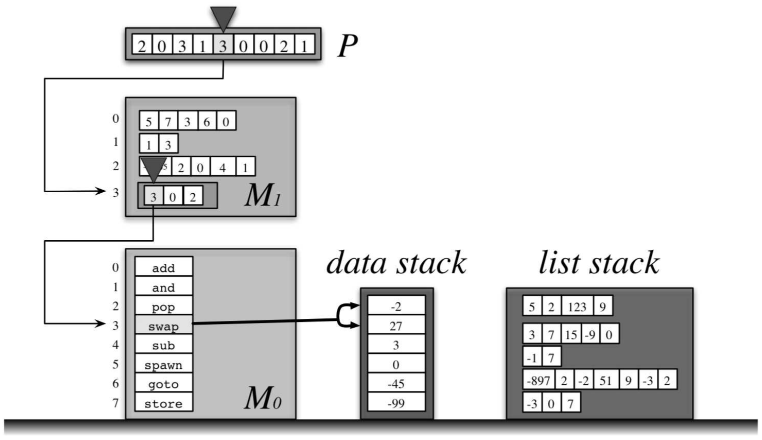

Fig. 2. Schematic representation of two-level machine execution. The execution frame is executing the 5^{th} instruction of program P, that calls program 3 of machine M_1. The first instruction of the 3^{rd} program of M_1 is the primitive instruction `swap` that will swap the two topmost elements of the data stack.

None of the existing languages used in EC provides mechanisms to manipulate machine levels – a property needed for our EVM implementation. There are other features we would like the base machine language to possess that none of the existing languages have. For example, it is desirable that a language be capable of redefining itself. Thus the primitive instruction set must allow the evolutionary process to restructure and redefine that set. We would also like to have a programming language that is highly expressive, that is, we want solution programs to typically encountered tasks to be expressible in compact form. Moreover, we believe that there are efficiency and expressibility advantages for a language with solution spaces that are highly recursive.

Some existing languages possess some of these desired properties, but no single one possesses all of them. For this reason we have designed our own specialised programming language, called ’EVM assembly’. The principal objective of this programming language is to facilitate searches for languages specialised for a given set of problems.

3.1 Implementation

Our current implementation of the EVM architecture is based on a stack-machine, such as Forth, or the Java Virtual Machine (JVM). In fact, with small differences, it is comparable to an integer-based subset of the JVM. The implementation is written entirely in Java, and developers can obtain it from CVS [3].

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 *the data stack*, or for short *the stack*. The 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* itself. Unlike the JVM, our virtual machine does not provide any operations for creating and manipulating arrays. Instead, EVM facilitates operations on lists. There is a special stack, called *the list stack* for storing integer-based lists.

Execution frames are managed in a similar way to the JVM, via a special execution frames stack. 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. 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 (primitive instructions for that machine), the sequence must be executed on another lower-level machine. The Base Machine implements all the primitive instructions that are not reified further into more primitive units.

Potentially, EVM programs can run indefinitely and therefore each thread of execution has an instruction time limit to constrain the time of each program in a multi-EVM environment. Each execution thread (a single program) has a maximum number of primitive instructions that it can execute. Once the limit is reached, the program unconditionally halts.

The EVM offers unrestricted reflection and reification mechanisms. 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 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. In addition, 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 near limitless flexibility and capabilities for reifying EVM execution.

3.2 Extensions

One possible way of extending current EVM implementation is by adopting bias-optimal search primitives [4], or the incremental search methods [11]. 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. A more detailed description of the abstract EVM architecture is given elswhere [9].

[3] `http://www.sf.net/projects/cirrus`

4 Specialisation of an individual machine

Given an initial set of instructions I, the specialisation mechanism will aim to find programs from I^* (the set of all possible sequences of instructions) that solve tasks defined by the environment. We have tried, independently, three different search methods for our EVM model: 1) random search; 2) GA with variable lengths of chromosomes; and 3) stochastic search based on a probability distribution of individual instructions.

For preliminary testing of these different search methods we used programs that control an agent moving on a two-dimensional discrete grid. The cells on the grid may contain rewards. Grids for experiments 1, 2, and 3 are depicted in Figure 3. In experiments 1 and 2 the rewards are persistent on the grid. In experiment 3, the agent can obtain each reward only once from a given cell (i.e. volatile rewards). This constraint has been added to force the agent to follow a path. The initial position of the agent is always the top left corner. Execution time is limited to 100 time steps for the first two experiments. This means, the ideal program for experiment 1 would be 100 instructions long, and will collect 100 reward units. For experiment 2, the perfect program would be again 100 instructions long, and would collect 99 reward units. Note, that there is a trade-off between the total number of rewards and the length of the program. For a program of length 6 that utilizes loop, the total number of rewards for experiment 2 is 66. For third experiment the time limit is set to 12 time steps and the total number of possible reward units collected is 12. The base instruction set has been extended with 4 special instructions: `down`, `up`, `right`, and `left` to move the agent on the grid.

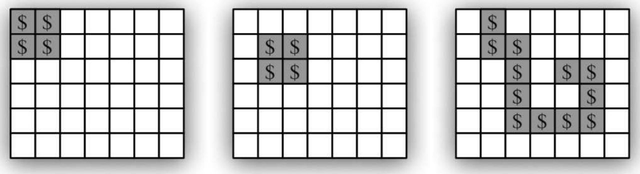

Fig. 3. Grid configuration for experiments 1, 2, and 3 respectively (the actual size of the grid is 100x100 instead of 7x7). Dark cells contain rewards

4.1 Specialisation with the use of Random Search

Random search is the simplest mechanism to specialise a machine. For each single cell it generates random programs. If one of the programs is successful, it will stay; and if it is not successful, then a new program will be randomly generated and tested[4]. On one

[4] In our implementation this is implemented by the following algorithm. First step: create a machine with a randomly selected program. On request return that program, and if that program received a reward, *freeze* it, i.e. always return the same program to the evaluator. If the program

hand, random search cannot take advantage of regularities in the fitness landscape. But on the other hand, it has no parameters, is fast, and needs very little memory.

For arithmetical problems, the landscape basically consists of one big peak with a steep slope (rewards are either all or nothing). In such circumstances, random search is appropriate and performs quite well when compared with other methods. Moreover, by implementing the simplest possible search mechanism for every cell, it is possible to focus on macroscopic behavioural patterns, i.e. how cells interact to compute a complex solution.

In the general case, though, most problems (like the maze experiments) display regularities in the fitness landscape. For that reason, we seek more complex search mechanisms that can take advantage of these regularities.

4.2 Specialisation with the use of Genetic Algorihtms

For our implementation of the genetic algorithm (GA) module, chromosomes are represented as lists of integers. There is an extra mutation operator that inserts or deletes individual instructions into the program, and for all the experimental runs the probabilities of addition and removal were set to the same rate. The initial size of the program, as well as all the other mutation and crossover probabilities, varied, so it was possible to hand-tune the parameters for a given problem to achieve satisfactory GA performance. For experiment 1 we used a population size of 1000 and up to 2000 generations, with the probability of crossover set at 0.8 and the mutation rate set at 0.01 (the add/remove probability was set at 0.05). The GA-based search did not have any trouble finding suboptimal solutions that utilize loops, but it was unable to find a global optimum.

Trade-off between exploration and exploitation. The main drawback of the GA-based search was that it does not dynamically adapt to different exploration strategies for different environments. Exploration in GAs is basically performed by the mutation operation. For example, in experiment 2, often a single run of GA-based search was unable to find any rewarded sequence for 3000 generations with 1000 individuals. Thus, the success of the GA run depended purely on the initial randomly generated population. If it was initialized without any good *building blocks*, the population was unable to discover any useful subsequence purely by mutations alone[5]

Converging to suboptimal solutions. The GA-based search always prefers shorter solutions to the longer ones, because shorter ones are statistically more stable. The heavy use of loops prevents GA-based search from finding global optima (apart from experiment 3, where GA-based search in half of the runs was able to find the global solution).

Stable prefix and bloat of introns at the tail. GA-based search generated solutions that are characterized by a relatively stable prefix, and a very long chaotic tail of introns.

solves a task, it will be rewarded and its cumulative rewards will be higher than zero. If the program does not solves the task, subtract a fixed amount from the cumulative rewards. Return the same *frozen* program for each request, until the frozen program cumulative rewards are zero, and the program is starving. Then, go back to the first step, and create different random program.

[5] The probability of such an event is around 0.0000000164: there are 78 instructions in total, and the minimal length of a rewarded program is 2.

Introns are simply instructions that are either not executed at all, or, when executed do not have any negative or positive side-effects for the program. Again, exploration of the program space tends to concentrate on the end of the program structure. The closer to the beginning, the more stable the instruction become. This is very similar to the "freezing" mechanism discussed in the next section in regard to stochastic search.

4.3 Specialisation with the use of stochastic search

For the stochastic search we assumed that the number of instructions per program, and the number of programs per machine were limited. The basic idea is that by limiting the size of the machine we can assign a probability distribution to each instruction of the machine M. For each instruction of M, there is a probability distribution over its possible values (Figure 4).

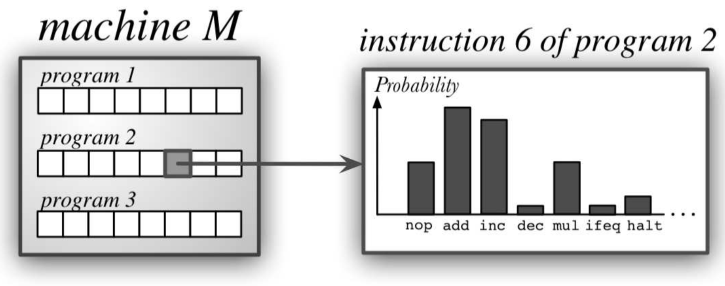

Fig. 4. Every instruction of a machine M contains a probability distribution over its possible values

To evaluate a machine, specific values are randomly picked representing the instructions of M according to their probability distributions. The result is an instance m of M, which will be used in an attempt to solve a problem. Programs of m will then be executed until a solution is found. Depending on their success, probabilities of programs of m will be increased or decreased. Every time a reward r is gained with a program p, the probabilities of p's instructions are increased. On the other hand, if p was not successful (implying no reward), the probabilities of its instructions were decreased slightly.

In addition to finding the solution to a particular problem, this mechanism is aimed at *remembering* solutions. The intent is to store these solutions in the machine's list of programs. For instance, a machine M can specialise in solving arithmetical operations. The environment provides several arithmetical problems. M will more or less randomly explore the space of possible programs until it starts finding the first solutions. Then by solving the same tasks again and again, probabilities will increase, and solutions will progressively be stored in M's programs.

Correlation between instructions. The primary drawback of the stochastic approach (and to a certain extent GAs, as well) is that it does not take into account correlations between instructions. The program's measure of quality highly depends on all its instructions taken together. Changing one of them may disrupt the entire program and penalize the other instructions, even if these are likely to be beneficial. Storing conditional probabilities might be a potential solution to that problem. Ideally, we could store successful subprograms' *patterns* of any length in a probability tree. It would then be equivalent to a Markov-chain based stochastic search. Experiments are currently being performed to evaluate this approach.

Freezing mechanism. Since instructions are randomly selected according to their probability distributions, there is always a probability of not selecting an important instruction i. To be sure i is in the program, the search mechanism tends to increase its probability in consecutive placeholders as well. Therefore, the search wound up confined to local optima such as

`right down right down down right down` (3 reward units)

for experiments 1 and 2. This is especially disappointing, because during the evolutionary process, the search mechanism does find some good solutions (up to 66 reward units in the 2nd experiment) but fails to remember them. The search process can be enhanced by introducing a *freezing mechanism* that will progressively "freeze" values in a program's instructions according to the following quasi-algorithm:

1. Assign $n \leftarrow 1$ (start from the first instruction)
2. If the probability of the n-th instruction has been almost maximal (within predefined threshold) for the number of iterations (another predefined constant), set the probability to 1 and don't modify it anymore (*freeze* it). Set $n \leftarrow n + 1$
3. Reset all the probabilities of the tail of the program, i.e. for all the instructions $n+1$ and above.
4. $n \leftarrow n + 1$ and return to step 2 until the entire program has been frozen.

The freezing mechanism has proved to be highly effective in many different problems we have tried, and it has always outperformed the stochastic search without freezing. In experiments 1 and 2, indeed, the search converges to solutions containing a loop, either with 2 or 4 agent movements instructions inside it (the more movement instructions, the higher the total reward). For instance, the second experiment produces these solutions:

`down right right left const_2 goto` (50 reward units)
`down right right down left up const_2 goto` (66 reward units)

The first program moves the agent: down, right, right, then left. After that sequence, a value of 2 is placed on top of the stack (instruction `const_2`, and instruction `goto` moves back to the third instruction in the program (`right`). The sequence of agent movements: `right`, `left` is executed indefinitely in the loop.

Long programs. Even with the help of the freezing mechanism, it becomes very difficult to build up longer programs, especially when a shorter (yet less rewarded) one is possible. The third experiment demonstrates this. Instead of finding the better 12-instructions-long program, all of our search methods are more likely to attempt to find a pattern to insert in a loop, e.g.:

```
right down right down down const_0 goto (7 reward units)
```

Since this program is shorter, it is more likely to be picked. It gets smaller rewards than the best solution, but it gets them more often, and for this reason it dominates. Again, this issue may be corrected by a probability tree.

5 Experiments with a web of interacting agents

Suppose that several machines lie on an n-dimensional grid (all executed asynchronously). In addition to primitive instructions (like `add`, `swap`,. . .), a machine can also access its neighbours' programs. There is no specific constraint on the grid's topology. It can be n-dimensional, and the neighbourhood can be as big as desired. In the extreme case, we could imagine a grid or web in which every machine can access every other machine. So far we have run experiments with two-dimensional grids with four neighbours, but other experiments with different topologies are planned to investigate further the impact of neighbourhood relation and locality.

A program now can look like the following:

`add dup` *program2ofLeftNeighbour* `mul` *program1ofRightNeighbour*

Moreover, if a program gets a reward, it will share it with any neighbour's program used to compute the solution. Both of them will benefit from their relationship. In other words, symbiotic relationships may appear between programs. This ability to access neighbours' programs has thus opened the door to complex hierarchical organization. As a consequence, machines are now able to collaborate to solve complex problems.

Consider the following example. On the grid are some machines (agents) specialised in list manipulation, and some specialised in recursion problems. The task is now to solve a problem involving both list manipulation and recursion. Cells surrounded by machines specialising for these two problems have the opportunity to find solutions to the task by accessing programs of their neighbours. Some machines will specialise in solving these simpler tasks and might be used by their neighbours to solve a more difficult task. A simple case, with two simple arithmetic tasks ($2x$ and $3x$) and one more complicated ($3x + 2y$), is depicted in Figure 5.

This computational model is thus well-suited for incremental problem solving. If solutions (or some part thereof) to some tasks can be reused to solve more complex ones, the EVM will take advantage of it. Indeed in such conditions, useful neighbours are more likely to appear.

5.1 Environment

From a machine learning point of view, the environment consists of a set of tasks to be solved (*multitask learning*). For the search process to be efficient, the model should fulfill the following requirements:

1. All tasks must be solved (eventually).
2. Solving difficult tasks should lead to greater rewards than easy ones.

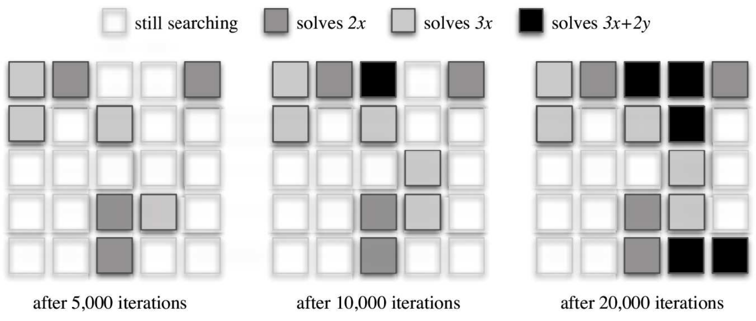

Fig. 5. Typical run exhibiting self assembly. After 5,000 iterations, several machines can solve the two simple tasks ($2x$ and $3x$). After 10,000 iterations, one machine ($m_{1,3}$) uses its neighbours to solve the hard task ($3x + 2y$), and all of the machines share the rewards (symbiosis). Shortly afterwards, some of its neighbours take advantage of it: they simply solve the task by calling $m_{1,3}$ (parasitism). Finally, after 20,000 iterations, we can observe another cluster of solutions at the bottom of the grid.

3. Computational resources (i.e. agents not currently solving any task) should focus on unsolved tasks.
4. Solutions must not be forgotten, as long as they are useful.
5. Knowledge diffusion across the web of interacting machines should be facilitated.
6. Dynamic environments should be supported: tasks can be added and/or removed at any time, dynamically.

From an artificial life point of view, one can view the environment as having to manage food *resources*, that are to be dispatched to the agents trying to consume them. Every resource represents a task to be solved. Every resource has two attributes: *quantity* and *quality*. Values for these attributes specify how much food (reward) will be given to the cell that consumes from the given resource.

The parameter $QUANTITY$ (capitalized to highlight its static nature) represents the abundance of resources in the environment. This value is set as an *a priori* conjecture by the modeller and is the same for each of the resources. It allows us to tune the amount of agents that will be able to survive.

The resource's quality has to reflect how difficult a task is. It facilitates a mechanism to give more rewards for hard tasks (requirement 2). There are several ways of measuring the difficulty of a task. Some are *a priori* (using expert knowledge), but it is more interesting to adjust it dynamically based on the observed difficulty. For example, the resource's quality may be set based on the observed average time it takes to solve it, or on how many agents can solve it, etc. We decided to set the resource's quality to the current minimal number of cells required to solve the task. It will reflect dynamically the task's complexity without depending on randomness and without the use of extra parameter that would need to be tuned for the search process.

When a cell consumes a resource, it gets the following amount of food:

$$food = \frac{QUANTITY\ quality}{consumers},$$

where $consumers$ is the number of agents eating the resource. Moreover, a cell has to share its food with all the neighbours it used to solve the task. Every cell used will get the same share of food [6].

At every iteration, a cell needs to eat a certain amount of food: F_{NEEDED}. If it eats more, it can makes provisions by storing it. On the other hand, if it eats less it will die from starvation once its provisions are empty.

$$provision_t = provision_{t-1} + food - F_{NEEDED}$$

5.2 Parameters and their impact

The two main parameters: the resource's quantity and the food needed for a cell to survive can be represented as one parameter $DENSITY$.

$$DENSITY = \frac{QUANTITY}{F_{NEEDED}\, SIZE},$$

where $SIZE$ is the total number of cells. This simplifies the model, because only the respective ratio is really important. $DENSITY$ controls the utilisation of the cells on the web. Figure 6 depicts two different settings for that parameter.

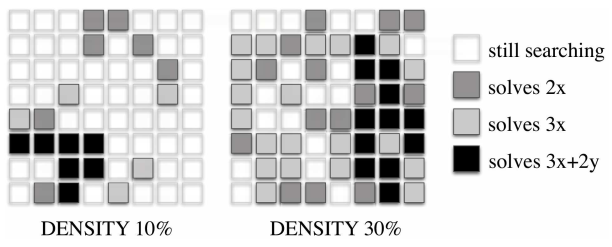

Fig. 6. Two different settings for the $DENSITY$ parameter (left: 10%; right: 30%).

Equilibrum/stability Another parameter, $PROVISION_{MAX}$, has been added. It sets a maximal bound for provisions stored by a cell. Its value drastically affects the dynamism of the web. If $PROVISION_{MAX}$ is high, most of the cells are stable and only a few appear and disappear (scenario A). If $PROVISION_{MAX}$ is low, we observe much more dynamic structural patterns on the web, with cyclic episodes similar to a kind of *catastrophe* scenario [12]. Good solutions spontaneously appear in the web, and after a while there are too many cells competing for the same resource. As a consequence, the quantity of the resource they are consuming decrease below the F_{NEEDED} threshold. Since they don't have enough provisions, they will soon almost all disappear. New cells can then start a new cycle (scenario B).

[6] For these early experiments, we have chosen a very simple reward mechanism. More complicated models will be investigated in our future work.

There seems to be no smooth transition between these two dramatically different scenarios. Scenario A represents a stable and fixed solid state, similar to Wolfram's class 1 of cellular automata (CA) classification [15]. Scenario B represents a cyclic state, and is similar to Wolfram's CA class 2. Wolfram's Classes 3 and 4 can be achieved by tuning the F_{NEEDED} parameter.

Knowledge diffusion. There is another interesting behaviour of interacting machines that can be observed. When a cell C_s solves a difficult task for the first time, the solution is almost immediately parasited by its neighbours. That phenomenon (see Figure 7) enables to diffuse a solution around the successful cell C_s, thus rendering this solution accessible to an increasing number of agents. Since some agents may need this solution to compute a more difficult problem, knowledge diffusion is highly desirable. Competition between parasites is very intense. They usually appear, survive a couple of iterations, disappear, after a while appear again, and so on. The dynamism exhibited looks like C_s is trying to reach something in its neighbourhood. For instance, if the diffusion manages to reach the neighbourhood of a cell C_1 that needs it, it will be used and thus the whole chain of parasites from C_s to C_1 will receive a lot of rewards and survive.

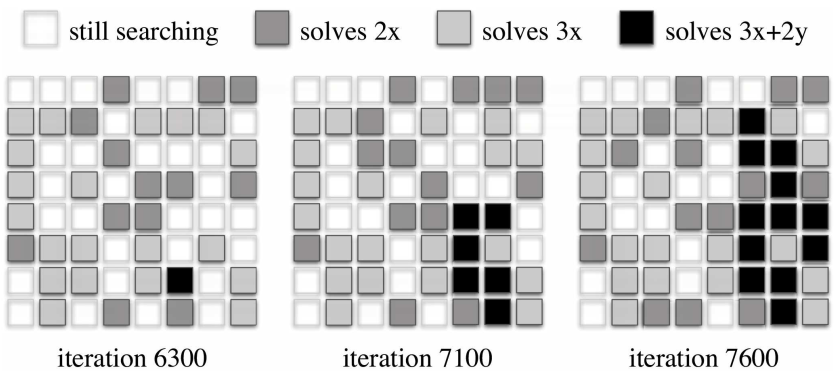

Fig. 7. Knowledge diffusion.

Once some other cells in the web solve the same task as C_s by their own (without parasiting), it becomes more and more difficult for the parasites of C_s to survive (as they always have to share their food with the cells they use). As a consequence, knowledge diffusion will progressively decrease.

6 Film analysis vs. quantitative studies

The study so far was mainly based on running simulations and preparing (off-line) the 2D movies (animation sequences) of the dynamics of the evolving web of cells. Once the movie has been generated, it has been observed and analysed by the researchers.

Most of the observed phenomena would be very difficult to be discovered by any other means. Of course, observations performed by the movie technique are very subjective, and do not represent statistically meaningful results. This is, however, the first step. Once certain phenomena are identified, then it is possible to prepare experiments and collect enough statistical data to confirm the initial observations. For the initial investigations, where some of the phenomena are not really expected or even completely unknown, the movie technique proved to be very successful.

It is our believe that utilisation of video sequences in this study is a valuable and essential element. It is one of the ways, if not the only way, to capture complex spatio-temporal aspects of certain phenomena, that cannot be predicted in advance. This technique has been used with great success in the field of cellular automata e.g. [16]. For 1D cellular automata the spatio-temporal aspect can be captured by a 2D image of the evolution of the single line of cells. However, for 2D almost all the studies must be done with video sequences (on-line or off-line).

7 Summary

An architecture of dynamic hierarchically organised virtual machines as a self-organising computing model has been presented. It builds on the Turing-machine-based traditional model of computation. The model provides some of the necessary facilities for open-ended evolutionary processes in self-organising software systems. The EVM system exhibits self-adaptation and self-maintenance. The EVM components are autonomous, they are executed asynchronously, they have no assigned specific function and can interact with the local neighbours. The EVM system exhibits emerging properties and hierarchical organisation via selection mechanisms, and symbiotic relationships between components.

The EVM architecture is particularly effective when applied to problems with an inherently well-organised structure. The results obtained so far suggest (although much more extensive experimentation is needed) that it can outperform random search, GA and Markov-chain based search techniques, for problems that exhibit well organised structural patterns, and when the problem search can be split into subspaces that can be explored independently/incrementally by the EVM web of agents. In general, it appears to perform as well as a bias-optimal search techniques (and as well as standard GA). During our experiments, for specific problems, with well-defined incremental subproblems, it outperformed both GAs and stochastic search.

More experimental data needs to be collected, and more formal comparisons with existing program search techniques is planned for the future. We also plan to investigate different topological environments, environments with resource locality, and investigating the influence of introducing mobility of cells, to boost diffusion.

Nevertheless, we believe that this computational mechanism can be successfully applied to a broad range of tasks, and through its inherent hierarchical organisation, can prove to be well suited for managing highly complex computational systems. Combined with existing evolutionary search techniques, like GAs, it offers unique ability of collapsing abstraction levels and managing dynamically interdependencies between computing agents.

Bibliography

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

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

[3] John R. Koza, Forrest H. Bennett, David Andre, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, 1999.

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

[5] Lynn Margulis. *Origin of Eukaryotic Cells*. University Press, New Haven, 1970.

[6] Lynn Margulis. *Symbiosis in Cell Evolution*. Freeman & Co., San Francisco, 1981.

[7] 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.

[8] Konstantin Sergeivich Mereschkowsky. Über Natur und Ursprung der Chromatophoren im Pflanzenreiche. *Biol. Zentralbl.*, 25:593–604, 1905.

[9] Mariusz Nowostawski, Martin Purvis, and Stephen Cranefield. An architecture for self-organising evolvable virtual machines. In Sven Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos, and Radhika Nagpal, editors, *Engineering Self Organising Sytems: Methodologies and Applications*, number 3464 in Lecture Notes in Artificial Intelligence. Springer Verlag, 2004.

[10] 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.

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

[12] René Thom. *Structural stability and morphogenesis*. Benjamin Addison Wesley, New York, 1975.

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

[14] Ivan Wallin. *Symbionticism and the Origin of Species*. Williams & Wilkins, Baltimore, 1927.

[15] Stephen Wolfram. Universality and complexity in cellular automata. *Physica D*, 10:1–35, 1984.

[16] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Inc., first edition, May 2002.

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