Chapter 11

GENETIC PROGRAMMING OF AN ALGORITHMIC CHEMISTRY

W. Banzhaf[1] and C. Lasarczyk[2]
[1]*Memorial University of Newfoundland;* [2]*University of Dortmund*

Abstract We introduce a new method of execution for GP-evolved programs consisting of register machine instructions. It is shown that this method can be considered as an artificial chemistry. It lends itself well to distributed and parallel computing schemes in which synchronization and coordination are not an issue.

Keywords:

> Informally, an *algorithm* is a well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*. An algorithm is thus a sequence of computational steps that transform the input into the output.
>
> (Introduction to Algorithms, TH Cormen et al)

1. Introduction

In this chapter we shall introduce a new way of looking at transformations from input to output that does not require the second part of the definition quoted above: a prescribed sequence of computational steps. Instead, the elements of the transformation, which in our case are single instructions from a multiset $I = \{I_1, I_2, I_3, I_2, I_3, I_1, ...\}$ are drawn in a random order to produce a transformation result. In this way we dissolve the sequential order usually associated with an algorithm for our programs. It will turn out, that such an arrangement is still able to produce wished-for results, though only under the reign of a programming method that banks on its stochastic character. This method will be Genetic Programming.

A program in this sense is thus not a sequence of instructions but rather an assemblage of instructions that can be executed in arbitrary order. By randomly choosing one instruction at a time, the program proceeds through its transformations until a predetermined number of instructions has been executed. In the present work we set the number of instructions to be executed at five times the size of the multiset, this way giving ample chance to each instruction to be executed at least once and to exert its proper influence on the result.

Different multi-sets can be considered different programs, whereas different passes through a multi–set can be considered different behavioral variants of a single program. Programs of this type can be seen as artificial chemistries, where instructions interact with each other (by taking the transformation results from one instruction and feeding it into another). As it will turn out, many interactions of this type are, what in an Artificial Chemistry is called "elastic", in that nothing happens as a result, for instance because the earlier instruction did not feed into the arguments of the later.[1]

Because instructions are drawn randomly in the execution of the program, it is really the concentration of instructions that matters most. It is thus expected that "programming" of such a system requires the proper choice of concentrations of instructions, similar to what is required from the functioning of living cells, where at each given time many reactions happen simultaneously but without a need to synchronicity.

Even if the reader at this point is skeptical about the feasibility of such a method, suppose for the moment, it would work. What would it mean for parallel and distributed computing? Perhaps it would mean that parallel and distributed computing could be freed from the need to constantly synchronize and keep proper orders. Perhaps it would be a method able to harvest a large amount of CPU power at the expense, admittedly, of some efficiency because the number of instructions to be executed will be higher than in deterministic sequential programs. In fact, due to the stochastic nature of results, it might be advisable to execute a program multiple times before a conclusion is drawn about its "real" output. In this way, it is again the concentration of output results that matters. Therefore, a number of n passes through the program should be taken before any reliable conclusion about its result can be drawn. Reliability in this sense would be in the eye of the beholder. Should results turn out to be not reliable enough, simply increasing n would help to narrow down the uncertainty. Thus the method is perfectly scalable, with more computational power thrown at the problem achieving more accurate results.

We believe that, despite this admitted inefficiency of the approach in the small, it might well beat sequential or synchronized computing at large, if we

[1]Elastic interactions have some bearings on neutral code, but they are not identical.

imagine tens of thousands or millions of processors at work. It really looks much more like a chemistry than like ordinary computing, the reason why we call it algorithmic chemistry.

2. Background

Algorithmic Chemistries were considered earlier in the work of Fontana (Fontana, 1992). In that work, a system of λ-calculus expressions was examined in their interaction with each other. Due to the nature of the λ-calculus, each expression could serve both as a function and as an argument to a function. The resulting system produced, upon encounter of λ-expressions, new λ-expressions.

In our contribution we use the term as an umbrella term for those kinds of artificial chemistries (Dittrich et al., 2001) that aim at algorithms. As opposed to terms like randomized or probabilistic algorithms, in which a certain degree of stochasticity is introduced explicitely, our algorithms have an implicit type of stochasticity. Executing the sequence of instructions every time in a different order has the potential of producing highly unpredictable results.

It will turn out, however, that even though the resulting computation is unpredictable in principle, evolution will favor those multi-sets of instructions that turn out to produce approximately correct results after execution. This feature of approximating the wished-for results is a consequence of the evolutionary forces of mutation, recombination and selection, and will have nothing to do with the actual order in which instructions are being executed. Irrespective of how many processors would work on the multi-set, the results of the computation would tend to fall into the same band of approximation. We submit, therefore, that methods like this can be very useful in parallel and distributed environements.

Our previous work on Artificial Chemistries (see, for example (Banzhaf, 1993, di Fenizio et al., 2000, Dittrich and Banzhaf, 1998, Ziegler and Banzhaf, 2001)) didn't address the question of how to write algorithms "chemically" in enough detail. In (Banzhaf, 1995) we introduced a very general analogy between chemical reaction and algorithmic computation, arguing that concentrations of results would be important. The present contribution aims to fill that gap and to put forward a proposal as to how such an artificial chemistry could look like.

3. The Method

Genetic Programming (GP) (Koza, 1992, Banzhaf et al., 1998) belongs to the family of Evolutionary Algorithms (EA). These heuristic algorithms try to improve originally random solutions to a problem via the mechanisms of recombination, mutation and selection. Many applications of GP can be described

as evolution of models (Eiben and Smith, 2003). The elements of models are usually arithmetic expressions, logical expressions or executable programs.

Here, we shall use evolution of a sine function (an approximation problem) and of a thyroid pattern diagnosis problem (a classification problem). We represent a program as a set of instructions only stored as a linear sequence in memory due to technical limitations. These instructions are 2 and 3 address instructions which work on a set of registers.

It should be noted that — in contrast to tree-based GP — each change in an instruction of this representation will have global effects. If, as a result of a change in an instruction, a certain register holds a different value, this will affect all registers making use of this register as input argument.

Linear GP with Sequence Generators

Here we shall use 3-address machine instructions. The genotype of an individual is a list of those instructions. Each instruction consists of an operation, a destination register, and two source registers[2]. Initially, individuals are produced by randomly choosing instructions. As is usual, we employ a set of fitness cases in order to evaluate (and subsequently select) individuals.

Figure 11-1 shows the execution of an individual in linear GP. A sequence

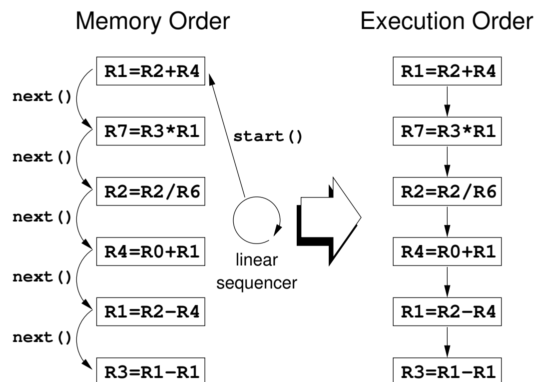


Figure 11-1. Execution of an individual in linear GP. Memory order and execution order correspond to each other. Arrows indicate returned values of calls to the sequence generator.

generator is used to determine the sequence of instructions. Each instruction is executed, with resulting data stored in its destination register. Usually, the sequence generator moves through the program sequence instruction by instruc-

[2]Operations which require only one source register simply ignore the second register.

tion. Thus, the location in memory space determines the particular sequence of instructions. Classically, this is realized by the program counter.[3]

1–Point–*Crossover* can be described using two sequence generators. The first generator is acting on the first parent and returns instructions at its beginning. These instructions form the first part of the offspring. The second sequence generator operates on the other parent. We ignore the first instructions this generator returns[4]. The others form the tail of the offsprings instruction list.

Mutation changes single instructions by changing either operation, or destination register or the source registers according to a prescribed probability distribution.

A register machine as an Algorithmic Chemistry

There is a simple way to realize an chemistry by a register machine. By substituting the systematic incremental stepping of the sequence generator by a random sequence we arrive at our system. That is to say, the instructions are drawn randomly from the set of all instructions in the program[5]. Still, we have to provide the number of registers, starting conditions and determine a target register from which output is to be drawn.

As shown in Figure 11-2 the chemistry works by executing the instructions of an individual analogous to what would happen in a linear GP–System (cf. 11-1), except that the sequence order is different.

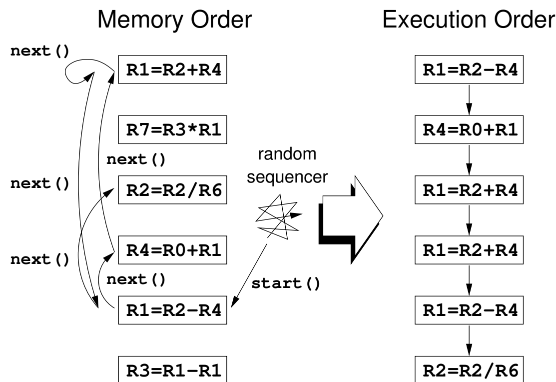


Figure 11-2. Execution in the AC system. The sequence generator returns a random order for execution.

[3](Conditional) jumps are a deviation from this behavior.
[4]Should crossover generate two offspring, the instructions not copied will be used for a second offspring.
[5]For technical reasons instructions are ordered in memory space, but access to an instruction (and subsequent execution) are done in random order.

It should be noted that there are registers with different features: Some registers are read-only. They can only be used as source registers. These registers contain constant values and are initialized for each fitness case at the start of program execution. All other registers can be read from and written into. These are the connection registers among which information flows in the course of the computation. Initially they are set to zero.

How a program behaves during execution will differ from instance to instance. There is no guarantee that an instruction is executed, nor is it guaranteed that this happens in a definite order or frequency. If, however, an instruction is more frequent in the multi-set, then its execution will be more probable. Similarly, if it should be advantageous to keep independence between data paths, the corresponding registers should be different in such a way that the instructions are not connecting to each other. Both features would be expected to be subject to evolutionary forces.

Evolution of an Algorithmic Chemistry

Genetic programming of this algorithmic chemistry (ACGP) is similar to other GP variants. The use of a sequence generator should help understand this similarity. We have seen already in Section 3.0 how an individual in ACGP is evaluated.

Initialization and mutation. Initialization and mutation of an individual are the same for both the ACGP and usual linear GP.

Mutation will change operator and register numbers according to a probability distribution. In the present implementation register values are changed using a Gaussian with mean at present value and standard deviation 1.

Crossover. Crossover makes use of the randomized sequences produced by the sequence generator. As shown in Figure 11-3 a random sequence of instructions is copied from the parents to the offspring. Though the instructions inherited from each of the parents are located in contiguous memory locations, the actual sequence of the execution is not dependent on that order. The probability that a particular instruction is copied into an offspring depends on the frequency of that instruction in the parent. Inheritance therefore is inheritance of frequencies of instructions, rather than of particular sequences of instructions.

Constant register values will be copied with equal probability from each parent, as is done for choice of the result register.

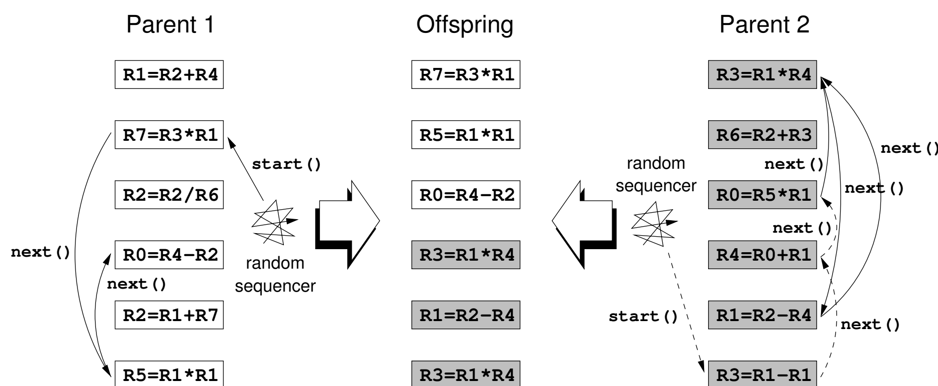


Figure 11-3. Crossover in an Artificial Chemistry.

Measures

Most measures, like the number of instructions of a program, can remain the same as in other GP systems, some are not observable at all, e.g. edit distance, or are completely new, as *connection entropy* described next.

If a register is written into from different instructions of a program which all might be equally frequent in the program, reproducibility of a result is strongly dependent on the sequence generator's seed. If, however, all registers are only written into from one instruction the result is more reproducible.

In order to increase reproducibility of results, the concentration of necessary instructions needs to be increased and that of other instructions needs to be decreased. One main influence on this is provided by crossover. At the same time, however, it is advantageous, to decouple flow of data interfering with the proper calculation. The connection entropy is designed to measure the progress along this line.

Let W be the set of connection registers participating in any data path. A connection register $i \in W$ might be written into by operations o_i^j of instructions j. Each of these instructions might be in multiple copies in the program, with $|o_i^j|$ the number of copies. We then have

$$O_i = \sum_{\forall j} |o_i^j|$$

the number of instructions that write into register i. Instruction o_i^j has probability

$$p_i^j = \frac{|o_i^j|}{O_i}$$

to have written into i. The entropy

$$e_i = -\sum_{\forall j}(p_i^j \cdot \log p_i^j)$$

of register i states how reproducible the value in a register is. The connection entropy finally reads

$$E = \frac{\sum_{i \in W} e_i}{|W|}.$$

The lower the connection entropy the more reliable the results from the execution of a program in ACGP are.

4. Description of Experiments

We take two sample problems to demonstrate that the idea works in principle. The first one is approximation of the sine function, the second problem is classification of thyroid function on real world data. Table 11-1 lists parameters chosen identically for both problems. Table 11-2 shows settings that differ for

Table 11-1. Common settings of both experiments.

Parameter	**Value**
Population	
Parents	100
Offsprings	500
Individual/Algorithmic Chemistry	
Connection registers	30
Evolved Constants	11
Operationset	add,sub,div,mult, pow,and,or,not
Init Length	100
Maximum Length	1000
Evaluation	
Nr. of randomly drawn instructions	$5\times$ length
Training set sampling	Stochastic subset sampling
Crossover rate	0.5
Mutation probability per entry	0.03
Evolution	
Generations	500

both problems. Additionally, both problems vary in their fitness function. In the following section we describe the applied fitness functions and give some more information about the two problems.

Table 11-2. Differences mainly concern the problem type and evaluation sets.

Parameter	**Value**	
	sine	**thyroid**
Problem type	regression	classification
Number of inputs	1 real value	21 values (6 real, 15 binary)
Training set size	1000	3772
subset (SSS)	100	400
Validation set size	400	1000
Testing set size	400	2428

Regression — Sine Function Approximation

Approximation of a sine function with non–trigonometric functions is a non–trivial but illustrative problem. The set of fitness cases $V = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ is created in the following way: In the interval $[-\pi, \pi]$ random values x_i are used to calculate values $y_i = sin(x_i)$, $i \in \{1, 2, \ldots, n\}$.

Given a subset V' of the training set V, the fitness function is the mean squared error of the individual I applied to all fitness cases of the subset:

$$f(I) = \left(\sum_{(x,y)\in V'} (I(x) - y)^2 \right) \Big/ |V'| \quad .$$

(x, y) denotes a fitness case in the subset V' of size $|V'|$, x the input and y the desired output.

Classification — Thyroid Problem

The thyroid–problem is a real world problem. The individual's task is to classify humans thyroid function. The dataset was obtained from the UCI–repository (Blake and Merz, 1998). It contains 3772 training and 3428 testing samples, each measured from one patient. A fitness case consists of a measurement vector containing 15 binary and 6 real valued entries of one human being and the appropriate thyroid function (class).

There are three different classes for the function of the thyroid gland, named *hyper function*, *hypo function* and *normal function.* As Gathercole (Gathercole, 1998) already showed, two out of these three classes, the hyper function and the hypo function, are linearly separable. Given the measurement vector as input, an individual of the ACGP system should decide whether the thyroid gland is normal functioning (class 1), or should be characterized as hyper or hypo function (class 2).

Because more than 92% of all patients contained in the dataset have a normal function, the classification error must be significantly lower than 8%. The

classification error is the percentage of misclassified dataset. We use the classification error as our fitness function.

The selection algorithm picks its subsets out of the 3772 training examples. From the set of testing examples we remove the first 1000 examples to form a validation set. The remaining examples form the testing set.

We assign the following meaning to the output of the individuals. A negative output (< 0) denotes normal function, otherwise hyper or hypo function.

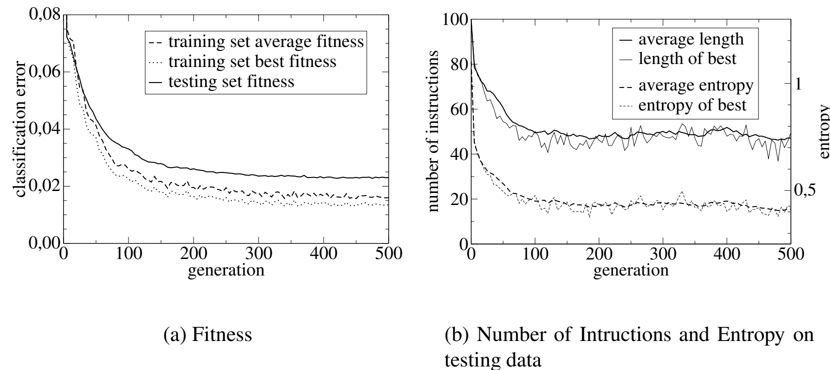
5. Performance Observation

Figure 11-4 shows the characteristics of fitness, length and entropy for both experiments described in section 4. All results are averaged over 100 runs.

Fitness

Fitness characteristics are shown for population average as well as populations best individual, based on a subset of the training set. All individuals are tested on a validation set and the best individual is then applied to the testing set. The third characteristics shows fitness on this set in average.

In Figure 11.4(a) one can see their variation in time for the thyroid problem.



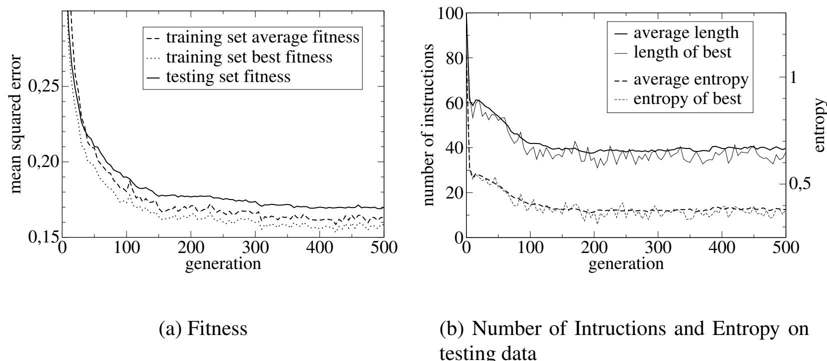
(a) Fitness

(b) Number of Intructions and Entropy on testing data

Figure 11-4. Observations on classification of thyroid function.

In this example fitness is equal to classification error. The average classification error (on the test set) after 500 generations is 2.29%. The lowest classification error ever reached is 1.36%, observed after 220 generations. Using different settings, Gathercole (Gathercole, 1998) reports classification errors between 1.6% and 0.73% as best result using his tree–based GP system. He also cites a classification error of 1.52% for neural networks from Schiffmann et. al. (Schiffmann et al., 1992).

Figure 11.5(a) shows mean squared error (MSE) as fitness value for the sinus approximation problem. The lowest MSE observed ever is 0.026. One



(a) Fitness

(b) Number of Intructions and Entropy on testing data

Figure 11-5. Observations on approximation of the sine function.

run achieved this value, but it got lost in subsequent generations. Runs better than average (0.17) show values next to 0.11. Certainly, there is room for improvement regarding these results.

The MSE might not be an adequate fitness function for Algorithmic Chemistries. No evaluation of a fitness case is like another, because a new random order of instructions is used for each fitness case. While an insufficient order leads to a low error on classification problems (one misclassification), it could lead to a large error on regression problems using MSE. Even if an Algorithmic Chemistry leads to good results on almost every fitness case, a single failure could have a large effect on the individual's fitness value. This complicates evolution. Limiting maximum error of a fitness case could be a possible way out.

Program Length and Connection Entropy

program length

Due to our definition of entropy, its variation in time is comparable to length of the individuals. For this reason they are shown in the same chart (Fig. 11.4(b) and 11.5(b)). We plotted the population means as well as the characteristics of population's best individual averaged over 100 runs. Values of the best individuals should give an impression on how selection pressure influences the mean value.

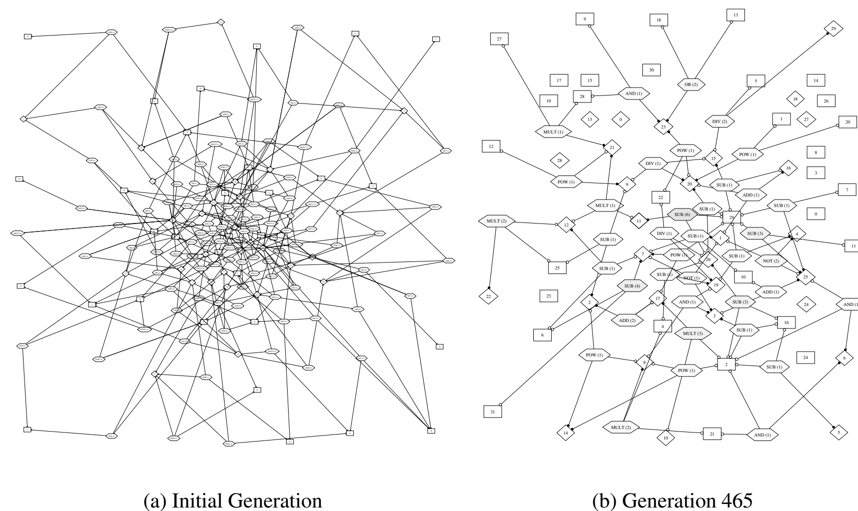
At the outset individuals loose between 20% and 40% of their initial length of 100 instructions immediately. Within the first 100 generations they reduce length even more and keep a nearly constant length afterwards. We cannot observe bloat by ineffective code as it is known in linear GP. For this behavior we take two reasons into account. First, bloat protects blocks of code belonging

together in a specific order from being separated by the crossover operation. As there is no order in ACs, there is no need for such kind of protection. Second, each nonessential instruction reduces the probability of calling an essential instruction in the right period of time. This cannot reduce bloat in linear GP, because there it is assured that every instruction is called once in sequential order.

With decrease in average length, average connection entropy declines, too. This increases the uniqueness of the value assigned to a register.

Visualization of an Algorithmic Chemistry

Figure 11-6 represents an Algorithmic Chemistry of the population at two different time steps. Each register is represented by a node. Read-only registers



(a) Initial Generation

(b) Generation 465

Figure 11-6. Graph of the best Algorithmic Chemistry for Thyroid problem at different generations.

are drawn as boxes. Other registers are symbolized by a diamond. Thus the output register is a diamond, and it is also drawn bold. Instructions are presented as hexagons. They are labeled by the name of the operation that belongs to the instruction they represent with number of identical instructions in parenthesis.

Every instruction–node is connected by an edge to the register it uses. The circle at the end of an edge symbolizes the kind of access. A filled circle shows a write access, an empty circle shows an read access. Flow of information between instructions happens when one instruction writes and the other reads a particular register.

Figure 11.6(a) shows a graph of an initial individual. It consists of 100 randomly assembled instructions. Nearly all instruction are single instances, and all registers are in use. Figure 11.6(b) shows a later time step in evolution. One can clearly see the reduced number of different instructions. Some instructions show a higher concentration than the others. Many registers that only allow read access, proved to be useless during evolution and are not accessed any more.

In Figure 11-7 one can see a part of the last graph with a different layout.

Figure 11-7. Most important part of the Chemistry shown in Figure 11.6(b). It shows all operations and registers responsible for the chemistries result. The number in parenthesis indicates how often this instruction is in the multiset. Brightness should indicate an instruction's frequency, with darker nodes having higher frequency.

The result register of this AC is register 11 shown at top left. Just one instruction is doing a write access on this register. It is a subtraction available six times in this AC. One of its source registers is register 29 of the register set that just allows read access. While the first 21 Registers in this set contain the inputs of the fitness case, the others contain evolved constants. The second input is a writable register (numbered 26). At this point of time in evolution there is also just one write access to this register. Here it is another subtraction available three times in this chemistry. It subtracts two values of the fitness case.

This illustrates how evolution achieves an evaluation with a high reproducibility within Artificial Chemistries. The two main mechanisms are reduc-

tion of competitive write-access to registers and an increase in the concentration of important instructions.

6. Summary and Outlook

In this contribution it was not our intention to introduce simply a new representation for Genetic Programming. Instead, we wanted to show that goal-oriented behaviour is possible with a seemingly uncoordinated structure of program elements. This way we wanted to draw attention to the fact that an algorithmic chemistry could be a helpful concept for novel computer architecture considerations.

In fact a lot can be said about the similarity of this approach to dataflow architectures (Arwind and Kathail, 1981). Traditional restrictions of that architecture, however, can be loosened with the present model of non-deterministic computation, ”programmed” by evolution. Recent work in the dataflow community (Swanson et al., 2003) might therefore find support in such an approach.

Spinning the analogy of a genome further, we can now see that the instructions used in ACGP are equivalent to genes, with each gene being "expressed" into a form that is executed. Execution of an instruction can, however, happen uncoordinated with execution of another instruction. So we are much nearer to a regulatory network here than to a sequential program (Banzhaf, 2003).

The strength of this approach will only appear if distributedness is taken into account. The reasoning would be the following: Systems of this kind should consist of a large number of processing elements which would share program storage and register content. Elements would asynchroneously access storage and register. The program's genome wouldn't specify an order for the execution of instructions. Instead, each element would randomly pick instructions and execute them. Communication with the external world would be performed via a simple control unit.

It goes without saying that such a system would be well suited for parallel processing. Each additional processing element would accelerate the evaluation of programs. There would be no need for massive communication and for synchronization between processing elements. The system would be scalable at run-time: New elements could be added or removed without administrative overhead. The system as a whole would be fault-tolerant, failure of processing elements would appear merely as a slowed-down execution. Loss of information would not be a problem, and new processes need not be started instead of lost ones. Reducing the number of processors (and thus slowing down computation) could be allowed even for power management.

Explicit scheduling of tasks would not be necessary. Two algorithmic chemistries executing different tasks could be unified into one even, provided they used different connection registers. Would it be necessary that one task should be

prioritized a higher concentration of instructions would be sufficient to achieve that.

Finally (though we haven't demonstrated that here) programs which are never sequentially executed don't need to reside in contiguous memory space. A good deal of memory management would therefore also become superfluous.

According to (Silberschatz and Galvin, 1994) "[a] computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, file storage space, I/O devices, and so on. The operating system acts as a manager of these resources and allocates them to specific programs and users as necessary for their tasks". Architectural designs as the ones considered here would greatly simplify operating systems.

It is clear that non-deterministic programs resulting from runs of an ACGP system would not be suitable for all applications of computers. Already today, however, a number of complex systems (like the embedded systems in a car) have to process a large amount of noisy sensor data about the environment. It is frequently necessary to measure the same quantity repeatedly in order to arrive at safe observations. In such cases one would simply extend the repetition of tasks into computing. Adding processing would therefore simultaneously lead to more reliable conclusions from these observations.

Our real world is messy and non-deterministic. Would not a GP approach driving a messy and non-determinstic computational system be well suited for taking up these challenges?

Acknowledgements

The authors gratefully acknowledge support from a grant of the Deutsche Forschungsgemeinschaft DFG to W.B. under Ba 1042/7–3.

References

Arwind and Kathail, V. (1981). A multiple processor data flow machine that supports generalized procedures. In *International Conference on Computer Architecture (Minneapolis 1981)*, Los Alamitos, CA. IEEE Computer Society.

Banzhaf, W. (1993). Self-replicating sequences of binary numbers. *Comput. Math. Appl.*, 26:1–8.

Banzhaf, W., Nordin, P., Keller, R., and Francone, F. (1998). *Genetic Programming - An Introduction.* Morgan Kaufmann, San Francisco, CA.

Banzhaf, Wolfgang (1995). Self-organizing Algorithms Derived from RNA Interactions. In Banzhaf, W. and Eeckman, F.H., editors, *Evolution and Biocomputing*, volume 899 of *LNCS*, pages 69–103. Springer, Berlin.

Banzhaf, Wolfgang (2003). Artificial Regulatory Networks and Genetic Programming. In Riolo, R. and Worzel, B., editors, *Genetic Programming — Theory and Practice*, GP Series, pages 43–62. Kluwer, Norwell, MA.

Blake, C. L. and Merz, C. J. (1998). UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

di Fenizio, P. Speroni, Dittrich, P., Banzhaf, W., and Ziegler, J. (2000). Towards a Theory of Organizations. In Hauhs, M. and Lange, H., editors, *Proceedings of the German 5th Workshop on Artificial Life*, Bayreuth, Germany. Bayreuth University Press.

Dittrich, P. and Banzhaf, W. (1998). Self-Evolution in a Constructive Binary String System. *Artificial Life*, 4(2):203–220.

Dittrich, P., Ziegler, J., and Banzhaf, W. (2001). Artificial Chemistries - A Review. *Artificial Life*, 7:225–275.

Eiben, G. and Smith, J. (2003). *Introduction to Evolutionary Computing*. Springer, Berlin, Germany.

Fontana, W. (1992). Algorithmic chemistry. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 159–210, Redwood City, CA. Addison-Wesley.

Gathercole, Chris (1998). *An Investigation of Supervised Learning in Genetic Programming*. PhD thesis, University of Edinburgh.

Koza, John R. (1992). A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In *Proceedings of IJCNN International Joint Conference on Neural Networks*, volume IV, pages 310–318. IEEE Press.

Schiffmann, W., M.Joost, and Werner, R. (1992). Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical Report 15, University of Koblenz, Institute of Physics.

Silberschatz, A. and Galvin, P. B. (1994). *Operating System Concepts*. Addison-Wesley, Reading, MA, 4 edition.

Swanson, S., Michelson, K., and Oskin, M. (2003). Wavescalar. Technical Report UW-CSE-03-01-01, University of Washington, Dept. of Computer Science and Engineering.

Ziegler, J. and Banzhaf, W. (2001). Evolving Control Metabolisms for a Robot. *Artificial Life*, 7:171–190.