Total Synthesis of Algorithmic Chemistries

Christian W. G. Lasarczyk
Department of Computer Science
University of Dortmund
D-44221 Dortmund, Germany
christian.lasarczyk@udo.edu

Wolfgang Banzhaf
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, A1B 3X5, Canada
banzhaf@cs.mun.ca

ABSTRACT

Algorithmic Chemistries are Artificial Chemistries that aim at algorithms. In this contribution we present a new algorithm to execute Algorithmic Chemistries during evolution. This algorithm ensures synthesizes of the whole program and cuts off execution of unneeded instructions without restricting the stochastic way of execution. We demonstrate benefits of the new algorithm for evolution of Algorithmic Chemistries and discuss the relation of Algorithmic Chemistries with Estimation of Distribution Algorithms.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms

Keywords

Algorithmic Chemistries, Genetic Programming, Total Synthesis

1. INTRODUCTION

In this contribution we discuss a variant of a recently introduced model of computing based on the metaphor of chemistry, called Algorithmic Chemistry[1] [1]. In this approach, the input - output relation of a computation is formulated in direct analogy to a chemical reaction with educt–product relation. The computation itself is considered in analogy to a chemical reaction.

While the smallest entities of chemistry are molecules, what are the smallest entities of computation? Our point of view is that instructions can be considered on that level, forcing us to think of instructions as interacting with each other in a more or less irregular order.

[1]This term is considered first by Fontana[9]. Here we use it as an umbrella term for artificial chemistries that aim at algorithms [8].

Would there be any hope for a coordinated behavior in such a system? The answer is yes, provided that there is some degree of coherence between what instructions compute. That is to say, it all depends on the data flow. In the same way as molecules react with each other based on the pattern of their 3D shape, which is, in the case of more complicated molecules, often referred to as key-lock interaction, instructions interact with each other based on the patterns of addresses for source and destination registers.

We are going to take this analogy further here, by stripping away all those reactions which do not end up contributing to the final result of our computation. Because the system is of a stochastic nature, and we must allow all possible "reactions" to really occur, this is equivalent to simulating infinite time.

2. ALGORITHMIC CHEMISTRIES

Algorithmic Chemistries (AC) are multisets of objects, here of instructions[2], which cannot be accessed in a specific order. As usual, instructions consist of one operation id and three register addresses (two sources and one target). ACs are executed in an environment called reactor. As Fig.1 shows, this reactor mainly provides the initial values of the register set accessed by the instructions. There are three kinds of registers: Connection registers, registers containing constant values and input registers, which also do not change over the course of the computation. While instructions are allowed to read all of these registers, writing is only allowed to connection registers. In order to execute a chemistry we repeatedly draw a random instruction and execute it using the specified registers.

Because there is no explicit order, an arbitrary number of instructions can be executed in parallel, similar to what would happen in molecular reactions, for instance, in an in-vitro experiment. The probability of an instruction being executed at a specific moment only depends on its frequency in the multiset. This behavior contrast with most other program representations, where order determines execution.

If instructions share a common (connection–)register, in such a way that one instruction uses this register as its target and the other instruction uses it as one of its source registers, there is interaction between the two instructions.

[2]See [2] for a stepwise transformation of Linear Genetic Programming into Algorithmic Chemistries. Here we just consider Algorithmic Chemistries in its purest form, selecting all instructions with identical probability at each point in time ($\zeta = \infty$).

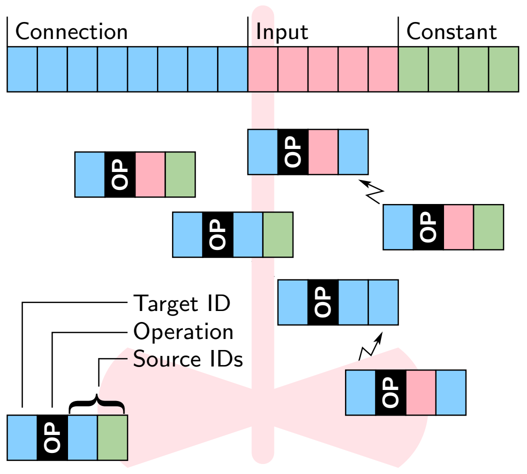


Figure 1: The reactor provides an environment for executing a multiset of instructions.The environment consists of empty connection registers, inputs from the fitness cases and registers containing constant values.

This corresponds to a data flow, possibly leading to the desired result of a computation in the target register. Due to the nature of probabilistic drawing of instructions, CPU cycle time needs to pass to ensure that the entire data flow encoded in the multiset actually executes ("synthesizes").

We use Genetic Programming to program Algorithmic Chemistries. In addition to the chemistry, each individual of the GP population consists of a set of evolved constants and a result register id for determination of the register interpreted as the chemistry's final output. Initially a population of p random individuals is generated. Constant values of an individual are randomly chosen from a predefined range, its chemistry is initialized by generating a set of i valid random instructions and its result register id is chosen randomly from the set of connection registers.

We use a (μ, λ)–strategy [5] and $\mu < \lambda$ best individuals are selected as parents of the next generation. A crossover rate determines the percentage of offspring generated by recombination of two randomly chosen parents. Recombination is done by drawing a random multiset of instructions from each of both parents. Size of these two multisets is uniformly distributed between 1 and the associated parents' chemistry size. The maximum size of a new chemistry is limited. The crossover operator copies each of the evolved constants as well as the id of the result register randomly from one of the two parents. The missing percentage of offspring are generated by reproducing randomly selected parents.

After recombination or reproduction, mutation takes place. The mutation rate determines the probability of each gene being mutated. Genes consist of register addresses and the operation id within instructions, the id of the output register and the individual's constants. While register addresses and operations are mutated by selecting another valid random value, constants are changed by multiplying them with a Gaussian variable $N(1, 0.1)$.

The process of evaluation of newly generated individuals is repeated until a termination criterion is reached. Figure 2 depicts these cyclic steps.

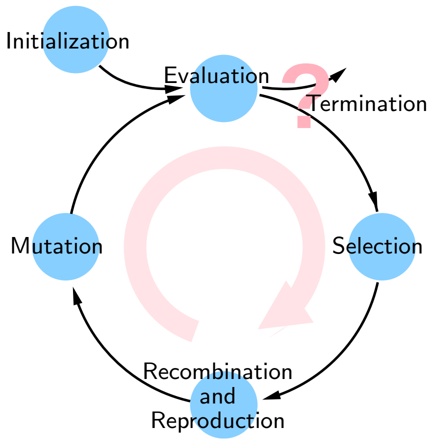


Figure 2: The evolution of Algorithmic Chemistries uses the same basic steps as classical evolutionary algorithms.

3. SYNTHESIS OF DATA FLOW

As mentioned above there is no inherent order of the instructions of an AC. Instead, instructions are executed in an arbitrary order in the reactor environment. Synthesis of the full data flow, however, is a matter of time, since data flow sequences are not encoded explicitly, but implicitly by shared connection registers in the multiset of instructions.

While time requirements could be compensated to some extent by parallelism, it is obvious that Algorithmic Chemistries require more computational power for execution than normal computer programs. A great many CPU cycles have to be spent on execution of instructions which do not participate in data flow towards the register interpreted as the individual's output. ACs share this problem with other representations of GP, where it is called non–effective code.

We discern two kinds of such dispensable instruction executions. The first kind is similar to introns/non–effective code in Genetic Programming. Instructions of this type participate under no circumstances in data flow, e.g. their target register is not read by another instruction. The second kind of instruction differs in that their execution is dispensable sometimes, and not at others. For example their source registers may not contain required values yet.

While spending computational power on executing dispensable instructions is intended for the benefit of parallelism during normal execution, it is problematic during the evolution of chemistries using GP, since many AC evaluations are needed.

Algorithm 1 outlines our original approach to evaluating AC–based individuals [1]. After preparing the reactor environment, for each fitness case the inner loop executes a number of randomly chosen (line 1.6) instructions. Subsequently, the value in the result register specified by the individual is used to calculate the contribution of this fitness case to fitness (line 1.14–1.15). In the end we compute the average contribution.

Here we call this approach "fixed", since a constant multiple c of the AC's size has to be specified in advance and fixes the number of executed instructions (line 1.5). Choosing c requires a balance between different considerations. On the one hand it is desirable to have a large c to ensure complete

Algorithm 1: Function to evaluate individual I using a set of fitness cases F and fixed number of executed instructions.

```
    Input: Set of fitness cases $F$, Individual
           $I$ = (chemistry, constants, id of result
           register $r$), number of cycles $c$,
           connection registers $C$
    Output: Fitness of $I$ in respect to $F$
1.1 $f = 0$;
1.2 foreach fitness case $\in F$ do
        /* prepare reactor environment */
1.3     $C_i = 0, \forall i$;
1.4     $R \leftarrow (C, \text{fitness case}, \text{constants})$;
1.5     for 1 to $c \times$ sizeOf(chemistry) do
1.6         instruction $\leftarrow$ getRandomInst(chemistry);
1.7         $j \leftarrow$ getTargetID();
1.8         $(k, l) \leftarrow$ getSourceIDs(instruction);
1.9         $m \leftarrow$ getRegister($k$,$R$);
1.10        $n \leftarrow$ getRegister($l$,$R$);
1.11        value $\leftarrow$ execute(instruction, $m$, $n$);
1.12        setRegister($j$,value,$R$);
1.13    end
1.14    result $\leftarrow$ getRegister($r$,$R$);
1.15    $f \leftarrow f +$ objective(fitness case,result);
1.16 end
1.17 return $f/|F|$;
```

data flow synthesis, on the other hand a small c is desirable for speeding up evolution.

By introducing total synthesis[3], we aim at synthesizing the entire data flow, something that is just ensured at infinite execution time otherwise. In addition, this should happen at the fastest speed possible, under the constraint that a reduction of randomness in the choice of instructions participating in the data flow is to be avoided.

Algorithm 2: Frame to totally synthesize an Algorithmic Chemistry.

```
    Input: Set of fitness cases $F$, Individual
           $I$ = (chemistry, constants, id of result
           register $r$)
    Output: Fitness of $I$ in respect to $F$
2.1 $f = 0$;
2.2 $S = (S_1, \ldots) \leftarrow$ GroupByTargetID(chemistry);
2.3 foreach fitness case $\in F$ do
2.4     $T \leftarrow$ (fitness case, constants);
2.5     result $\leftarrow$ recEval($r, S, T, \emptyset$);
2.6     $f \leftarrow f +$ objective(fitness case,result);
2.7 end
2.8 return $f/|F|$;
```

Total synthesis happens by recursively synthesizing one of the possible data flows each time an individual is executed. This is shown in Algo.2. First, the reactor environment does not access connection registers directly any more (line 1.4 vs. line 2.4). Instead, each access to a connection register evokes a recursive call of function `recEval`. T contains register values, which cannot be modified during evaluation. In order to be consistent with GP, we call T terminal set, since data flow branches terminate in those registers.

In addition a preparation step is advisable to execute an individual AC: First we have to group the chemistry (multiset of instructions) into sub–multisets S_i by the id i of the connection register they use as their target. This has to be done only once before executing an AC on a set of fitness cases (line 2.2), and is analog to removing structural introns before executing a linear GP individual (see [7]).

Algorithm 3: Pseudocode of the recursive evaluation function `recEval(`r,S,T,V`)` used in Algo.2.

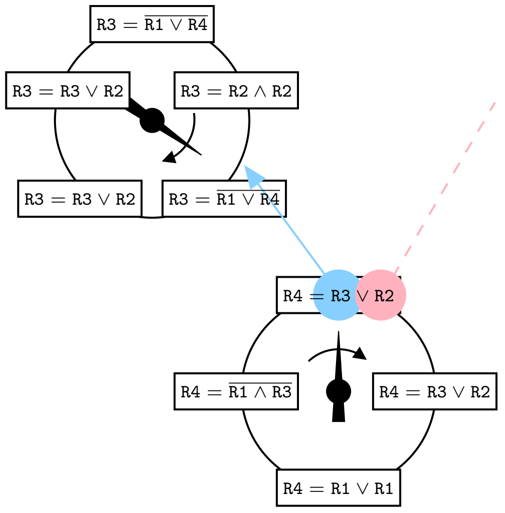
```
    Input: id $i$ of connection register to compute
           value for, set $S$ of instruction multisets,
           terminal registers $T$, set $V$ of connection
           register ids visited on recursive path
    Output: possible value of register $i$ at $t \to \infty$
3.1 if $i$ is connection register then
3.2     if $i \in V$ then
3.3         value $\leftarrow 0$ ;
3.4     else
3.5         instruction $\leftarrow$ getRandomInst($S_i$);
3.6         $(k, l) \leftarrow$ getSourceIDs(instruction);
3.7         $m \leftarrow$ recEval($k, S, T, V \cup \{i\}$);
3.8         $n \leftarrow$ recEval($l, S, T, V \cup \{i\}$);
3.9         value $\leftarrow$ execute(instruction, $m$, $n$);
3.10    end
3.11 else
3.12    value $\leftarrow$ getRegister($i, T$);
3.13 end
3.14 return value
```

The (connection–)register r used by an individual to compute output is known in advance, since its id r is an evolved component of the individual. Because r is the last register in the data flow the program writes to, a recursion is called by requesting its value (line 2.5). Function `recEval` presented in Algo.3 knows from its first attribute which connection register i is requested. Thus the sub–multiset S_i of relevant instructions targeting register i is known. An instruction is selected randomly from this multiset (line 3.5). Before the result of this instruction can be calculated, however, its source registers need to contain appropriate values. If sources are non–connection registers they are read from the terminal set T. If the current instruction reads any source k or l from connection registers, it needs to be ensured recursively that random instructions $s_k \in S_k$ or $s_l \in S_l$ have already been executed before.

Infinite loops are avoided by terminating recursion if a connection register id i is written to twice on a recursive path (line 3.2). Instead of drawing an instruction $s_i \in S_i$, `recEval` returns zero (line 3.3). Because all AC loops run the risk of being infinite, loops are terminated after the first cycle. If loops are required, bounded loops could be evolved by replacing V by a set of counters, counting the number of write accesses to a register an return zero (line 3.2–3.3) if a predefined number of writes accesses is reached.

Figure 3 depicts the basic idea of total synthesis. Starting at the result register chosen by evolution, instructions are executed recursively. As in earlier work multiple instances of the same instruction increase its probability of being executed.

[3]In chemistry this term is used for the complete synthesis of complex molecules from simple precursors.



R3 = $\overline{\text{R1} \vee \text{R4}}$
R3 = R3 ∨ R2
R3 = R2 ∧ R2
R3 = R3 ∨ R2
R3 = $\overline{\text{R1} \vee \text{R4}}$
R4 = R3 ∨ R2
R4 = $\overline{\text{R1} \wedge \text{R3}}$
R4 = R3 ∨ R2
R4 = R1 ∨ R1

Figure 3: With total synthesis of Algorithmic Chemistries, we build up data flow from the ground. Thereby a "wheel of Fortune" selects the instruction writing a connection register every time it is read.

4. RESULTS

Here we investigate the influence of total synthesis in the evolution of an Algorithmic Chemistries by considering the odd–parity problem.

The odd–parity problem has been used earlier in GP [10]. Individuals can use four logic operations {AND, OR, NAND, NOR} for a proper combination to detect parity. The cost for a random search on this problem has been discussed in [11].

Solving this problem does not require constant values, so the number of constant values is chosen to be zero. The fitness function (really an error function) corresponds to the fraction of fitness cases an individual cannot generate odd parity for. The solution hoped for is to have a fitness of zero.

As discussed in [12] noise evoked by the stochastic nature of executing instructions in ACs is a difficulty. This difficulty is amplified by problems with a small training set, such as is the case on the 3 bit odd–parity problem, which affords just $2^3 = 8$ fitness cases.

For example, the reader might think of an individual that contains the following two instructions targeting the individual's result register:

$$R_{\mathsf{res}} = \overline{R_i \vee R_i}; R_{\mathsf{res}} = R_i \vee R_i;$$

Suppose that the value of R_i is fixed, perhaps fixed by a unique data flow or by employing input or constant values. Every time this individual is executed, the output has an equal chance to be true or false. In other words, there is a small but finite probability of $1/2^8$ for this individual to achieve a perfect fitness. Such behavior misleads evolution during the selection step, much in contrast to what happens in linear or tree–based GP, where the same instruction enters into the result for each fitness case and therefore fitness is always 0.5.

Because fitness calculations become more accurate with an increased number of samples, the number of fitness cases is an important parameter for the evolution of Algorithmic Chemistries. A remedy in applications with a small number of fitness cases is to allow multiple consideration of fitness cases, whereas we ensure that frequencies of fitness cases do not differ by more than one. Training sets are resampled in each generation. For evaluating system performance, one selects the best individual on the basis of a validation set containing 128 fitness cases and test its fitness on a test set of 128 fitness cases[4]. Then the probability for the individual to achieve a fitness of zero by chance is $2^{-128} < 10^{-39}$.

We perform parameter optimization using [3, 4]. In short, this is a sequential iterative approach for stochastic model building, prediction and verification. First, the success rate is estimated after 10^8 instructions have been executed based on 10 runs at 200 points within our parameter space, chosen by an initial Latin Hypercube Sampling. The *success rate* corresponds to the proportion of runs that evolve an individual which computes the correct parity for all fitness cases of the test set. Success rates are used to model the response of the system. A quadratic regression model and a model for its prediction error using kriging interpolation can be employed. Using this composed model success rates of 2000 points from an Latin Hypercube Sampling can be predicted and the most promising settings by running real experiments. The sample size is doubled at the best (two) points known so far and for new settings we run the same number of experiments. After this verification step a new model is created, including the newly evaluated points. Again we predict success rates at points of a new Latin hypercube design and verify the most promising settings.

Table 1: Optimization ranges and optimal settings AC_c^* for executing a fixed number of instructions and AC_t^* for totally synthesized ACs.

	range		setting	
parameter	**min**	**max**	**AC_c^***	**AC_t^***
offspring λ	500	8000	3250	1323
crossover rate	0	1	11.78%	1.98%
mutation rate	0.01	0.15	7.32%	2.4%
initial length i	1	30	8	11
connection registers	5	30	8	11
training set size	8	128	67	15
cycles c	1	16	8.49	—
fixed values				
parents μ		100		
evolved constants		0		
max. instructions		2000		

Table 1 shows the parameter ranges for optimization. The column denoted AC_t^* shows the best settings found for total synthesis.

With AC_t^*, 400 runs were performed over a period of 10^9 instruction executions each. Every 2×10^7 performance of the best individual so far has been logged, using validation and test set as described above.

The solid line in Fig.4 shows the success rate using AC_t^*. After the period of time available for optimization (10^8) nearly 60% of all runs find a solution and correctly detect odd–parity. After 10^9 instruction executions more than 90% of all runs find a perfect solution. For comparison with the approach executing a fixed number of instructions, the

[4]Results achieved for the validation or test set do not find their way back into evolution

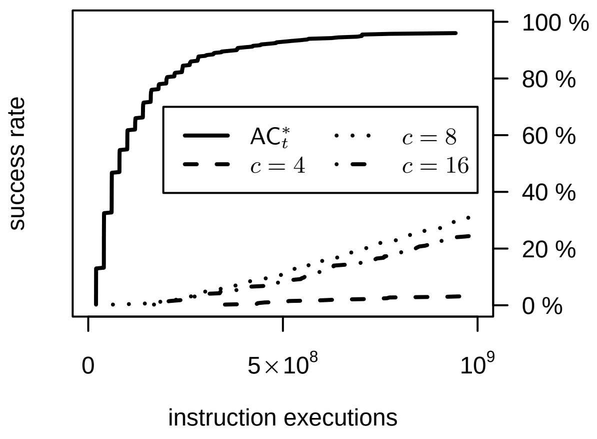


Figure 4: The solid line presents success rate on 3 bit odd–parity problem using optimized setting AC_t^* for total synthesis of Algorithmic Chemistries. The non solid lines present success rate for the same setting spending different amounts of executed instructions.

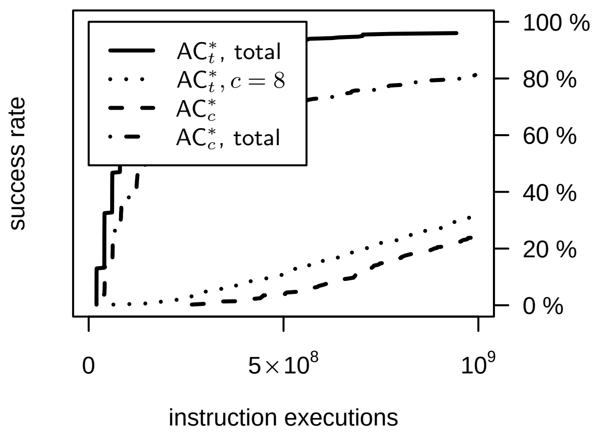


Figure 5: Comparison of success rate using optimized settings AC_t^* for total synthesis and settings AC_c^* optimized for randomly executing c times the number of instructions contained in chemistry.

number of executed instructions is limited in the latter case to a multiple c of the instructions contained in a chemistry. 400 runs are performed using $c \in \{4, 8, 16\}$, dashed lines in Fig.4 show performance of these runs. At first, performance increases with an increase in c. Chemistries benefit from an increased number of instruction executions, since this increases the chance to synthesize the data flow within execution time. If c increases beyond the value where data flow can be synthesized completely, no further gains for evolution can be registered, as less generations are accomplished in the same amount of time.

To ensure, that performance of the "fixed" approach is inferred on an equitable basis, another optimization is performed, this time including parameter c. Fig. 4 shows that only a low success rate can be expected after 10^8 instruction executions. The number is therefore increased to 5×10^8 instruction executions. Because the success rate is still low there, mean fitness instead of success rate is optimized. Column AC_c^* in Tab.1 shows the best settings found. While most parameter settings differ from the optimal setting for total synthesis, a number of $c = 8.49$ matches our expectation from Fig.4. Training set size is increased to 67. While total synthesis presented here is still a source of stochastic noise, it is much less so than in the "fixed" approach of execution.

Figure 5 shows the performance for AC_c^*. To enable cross validation we also depict AC_t^* and AC_c^* using total synthesis and AC_t^* executing a fixed number of instructions ($c = 8$). One can observe that optimization for fixed number of executed instructions AC_c^* does not result in a performance as good as AC_t^* with $c = 8$. Obviously one can simply adjust c a posteriori to get good results without total synthesis. This decreases computational power required for optimization, since one can optimize for an earlier moment (e.g. 10^8 instead of 5×10^8 here).

5. DISCUSSION AND OUTLOOK

The new execution algorithm groups instructions by their target register address into submultisets. It then randomly executes an instruction from this subset.

We can think of this procedure as a probabilistic instruction execution[5], where the probability $P(i)$ of executing instruction i depends on its frequency $f(i)$ in the chemistry. Assuming target register $t(i)$ of i is requested, we can formulate $P(i)$ in relation to frequencies of other instructions targeting the same register $t(i)$:

$$P(i) = \frac{f(i)}{\sum_{j|t(i)=t(j)} f(j)}.$$

The previous equation assumes, that $t(i)$ is requested. This is the case for $t(i) = r$ where r is the individual's result register. Otherwise, it depends on next instruction in data flow, which is the parent in the expanded tree. The probability of executing instruction i on a specific path p with distance d from final instruction (root) also depends on the next instruction's j source $s^p(j)$ on this path. So a multivariate distribution $P_d^p(i)$ describes the probability of instruction i being executed on a specific path p at distance d from the final instruction:

$$\begin{aligned} P_d^p(i) &= P(i) \sum_{j|s^p(j)=t(i)} P_{d-1}^p(j), \quad d > 0 \text{ and} \\ P_0^p(i) &= \begin{cases} P(i),\ t(i) = r \\ 0, \text{ else} \end{cases}. \end{aligned}$$

There are further approaches in Genetic Programming, to represent individuals in a probabilistic way, e.g.:

- Probabilistic Incremental Program Evolution(PIPE) [15] utilizes a single probabilistic prototype tree, which stores a random constant and a vector in each node, describing the probabilities of executing a particular instruction or using this constant.

- Extended Compact Genetic Programming (eCGP) [16] additionally considers multivariate interactions between nodes.

- Grammar based Estimation–of–Distribution for Genetic Programming [6] use simple grammars, limited

[5]Or even grammar expansion, e.g. replace instruction $R_k = R_i \vee R_j$ by production rule $S_k \rightarrow S_i \vee S_j$.

to one symbol. A probability for each production rule and (limited) expansion depth is adjusted.

The approaches all have in common that they explicitly model the probability distribution of instructions or production rules. They do not use crossover, but resample the next generation's population from the adjusted distribution. Therefore they are related to Estimation–of–Distribution algorithms (EDA) introduced in [13].

In the future, we would like to examine, whether Algorithmic Chemistries could benefit from this population genetic view of recombination in which we want to form offspring from the population distribution. Without making the distribution explicit, we are going to achieve this by sampling offspring chemistries from the common chemistry of all selected parents. In this way recombination will be similar to Gene Pool Recombination[14], a direct ancestor of EDA.

6. ACKNOWLEDGMENTS

The authors gratefully acknowledge support from a grant of the Deutsche Forschungsgemeinschaft DFG (German Research Foundation) to W.B. under Ba 1042/7–3 and from an NSERC Discovery Grant RGPIN 283304-04.

The authors thank Thomas Bartz–Beielstein for fruitful discussions and practical help on parameter optimization.

7. REFERENCES

[1] W. Banzhaf and C. W. G. Lasarczyk. Genetic programming of an algorithmic chemistry. In U.-M. O'Reilly, T. Yu, R. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice II*, volume 8 of *Genetic Programming*, pages 175–190. Kluwer/Springer, Boston MA, 2005.

[2] W. Banzhaf and C. W. G. Lasarczyk. A new programming paradigm inspired by artificial chemistries. In J.-P. Banâtre, J.-L. Giavitto, P. Fradet, and O. Michel, editors, *Unconventional Programming Paradigms (UPP–04)*, LNCS, Berlin, 2005. Springer.

[3] T. Bartz-Beielstein, K. E. Parsopoulos, and M. N. Vrahatis. Analysis of particle swarm optimization using computational statistics. In T.-E. Simos and C. Tsitouras, editors, *Proc. Int'l Conf. Numerical Analysis and Applied Mathematics (ICNAAM)*, pages 34–37, Weinheim, 2004. Wiley-VCH.

[4] T. Bartz-Beielstein, K. E. Parsopoulos, and M. N. Vrahatis. Design and analysis of optimization algorithms using computational statistics. *Applied Numerical Analysis & Computational Mathematics (ANACM)*, 1(2):413–433, 2004.

[5] H.-G. Beyer and H.-P. Schwefel. Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[6] P. A. N. Bosman and E. D. de Jong. Grammar transformations in an EDA for genetic programming. In *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, 26-30 June 2004. Distributed on CD-ROM at GECCO-2004.

[7] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, Feb. 2001.

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

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

[10] J. Koza. *Genetic Programming.* MIT Press, Cambridge, MA, 1992.

[11] W. B. Langdon and R. Poli. Boolean functions fitness spaces. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming: Second European Workshop EuroGP'99*, pages 1–14, Berlin, 1999. Springer.

[12] C. W. G. Lasarczyk and W. Banzhaf. An algorithmic chemistry for genetic programming. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *LNCS*, pages 1–12, Berlin Heidelberg, 2005. Springer.

[13] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions: I. Binary parameters. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature – PPSN IV*, pages 178–187, Berlin, 1996. Springer.

[14] H. Mühlenbein and H.-M. Voigt. Gene pool recombination in genetic algorithms. In L. Eshelman, editor, *Proc. Sixth Int. Conf. Genetic Algorithms*, pages 104–113, San Mateo, 1995. Morgan Kaufmann.

[15] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[16] K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practice*, chapter 13, pages 205–220. Kluwer, 2003.