Rule-based Genetic Programming

Thomas Weise, Michael Zapf, and Kurt Geihs
University of Kassel
Wilhelmshöher Allee 73
34121 Kassel, Germany
weise|zapf|geihs@vs.uni-kassel.de

ABSTRACT

In this paper we introduce a new approach for Genetic Programming, called rule-based Genetic Programming, or RBGP in short. A program evolved in the RBGP syntax is a list of rules. Each rule consists of two conditions, combined with a logical operator, and an action part. Such rules are independent from each other in terms of position (mostly) and cardinality (always). This reduces the epistasis drastically and hence, the genetic reproduction operations are much more likely to produce good results than in other Genetic Programming methodologies. In order to verify the utility of our idea, we apply RBGP to a hard problem in distributed systems. With it, we are able to obtain emergent algorithms for mutual exclusion at a distributed critical section.

Keywords

Genetic Programming, Rule-Based Genetic Programming, RBGP, Critical Section, Distributed Algorithms, Epistasis

1. INTRODUCTION

In this paper we introduce a new Genetic Programming technique called *rule-based Genetic Programming* or RBGP for short which is especially robust in terms of reproduction operations.

It reduces the strong positional interdependencies of different parts of a program which are a common drawback of many other Genetic Programming approaches, as discussed in the related work Section 2.

We elaborate on our new rule-based approach in Section 3. RBGP does not suffer from this general problem because of the general structure of its phenotypes, which is similar to Learning Classifier Systems. A program in RBGP consists of multiple rules. A rule, in turn, consists of two conditions, combined with a logical operator, and an action part. The different rules in such a program are positionally and cardinally independent. This leads to a very low epistasis which increases the efficiency of the reproduction operators. Like many of the well-known GP methods, RBGP uses a genetic algorithm [13] for search space exploration and employs a genotype-phenotype mapping [22, 21].

In the past, we have applied Genetic Programming to different problems in distributed computing [43, 41]. It is only natural to test the utility of the new rule-based Genetic Programming in the same domain. In Section 4 we demonstrate first tests of the new approach, the evolution of algorithms for mutual exclusion at a distributed critical section. These experiments exhibit interesting properties like *emergence* and a prolonged improvements during the evolution.

In Section 5 we finally conclude with a short summary and outline our plans for future work.

2. RELATED WORK

The roots of Genetic Programming go back to Friedberg who used a learning algorithm to stepwise improve a fixed-size program in 1958 [11, 12]. In the mid-1980s, Cramer utilized genetic algorithms and tree-like structures to evolve programs [6]. The standard tree-based Genetic Programming, which is most often used in practical applications and as reference model, was invented by Koza a few years later [23]. Since then, many different approaches have branched off.

A compiler parses the source code of a program as sentence in a given formal language. Only source code that can be produced by the grammar of this language is valid. In the late 1990s, multiple researchers began to recognize independently from each other that similar restrictions are needed if we want to evolve more complex structures [35, 45, 44, 20]. Grammar-guided Genetic Programming was born, culminating in methods like Grammatical Evolution [31, 34], tree-adjoining grammar-guided Genetic Programming [15, 14], and the recently developed Christiansen Grammar Evolution [7]. These approaches further have in common that they employ a genotype-phenotype mapping [1, 22]. Instead of applying the genetic operators directly to the programs, they work on genotypic representations like integer strings or derivation trees which subsequently are translated to program trees.

Another important stream is the linear Genetic Programming, close to the original idea of Friedberg. Here, programs are viewed as sequence of machine-code instructions. They are processed as string chromosomes by the genetic operators. Highly developed approaches even temporarily construct control flow graphs in order to preserve jump instructions. Prominent examples of this research area are [29] and [4].

A more thorough discussion of the different approaches to Genetic Programming can be found in [40].

2.1 Epistasis in Genetic Programming

Epistasis is defined as a form of interaction between different genes in biology. It was coined by Bateson [3] in order to describe how one gene can suppress the phenotypical expression of another gene.

The aforementioned, sophisticated new branches of Genetic Programming have successfully solved many problems of Koza's standard method. However, they also commonly share one of its drawbacks: a high degree of positional interdependencies in the phenotypes, which is a very basic form of *epistasis*.

In order to clarify the role of positional epistasis in the context of Genetic Programming, we begin with some basic assumptions. Let us consider a program P as a form of function $P : I \mapsto O$ that connects the possible inputs I of a system to its possible outputs O. Two programs P_1 and P_2 can be considered as equivalent if $P_1(i) = P_2(i) \;\; \forall i \in I$.[1]

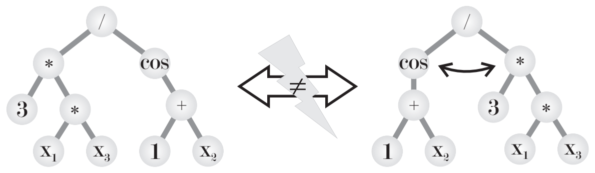
For the sake of simplicity, we further define a program as a sequence of n statements $P = (s_1, s_2, \ldots, s_n)$. For these n statements, there are $n!$ possible permutations. We argue that the fraction $\theta(P) = {}^{v}/_{n!}$ of permutations v that leads to programs equivalent to P is a measure of robustness for a given phenotypic representation in Genetic Programming. More precisely, a low value of θ indicates a high degree of epistasis, which means that the loci (the positions) of many different genes in a genome have influence on their functionality [28]. This reduces for example the efficiency of reproduction operations like crossover, since they often change the number and order of instructions in a program. A general rule in evolutionary algorithms is thus to reduce epistasis [33].

All phenotypic and most genotypic representations in Genetic Programming known to us are rather fragile in terms of insertion and crossover points. One of the causes is that their genomes have high positional epistasis (low θ-measures), as sketched in Figure 1.

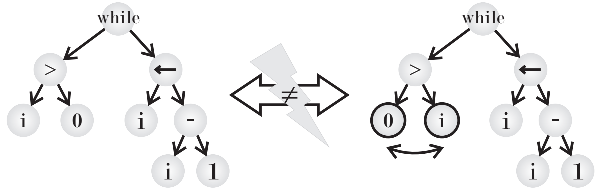
There exists one class of evolutionary algorithms that elegantly circumvents such problems: the learning classifier systems (LCS) which were developed in the 1970s by Holland [19, 18]. Here we focus on the Pittsburgh approach associated with Smith and De Jong [38, 39], where a genetic algorithm evolves a population of rule sets. Each individual in this population consists of multiple classifiers (the rules) that transform input signals into output signals. The evaluation order of the rules in such a classifier system C plays absolutely no role except for rules concerning the same output bits, i.e. $\theta(C) \approx 1$. Again, a more elaborate discussion of LCS can be found in [40].

The basic idea behind our approach is to use this knowledge to create a new representation for Genetic Programming that retains these high θ values in order to become more robust in terms of reproduction operations. This will probably lead to a smoother evolution with a higher probability of finding good solutions. On the other hand, we want to extend LCS by introducing some of the concepts from Genetic Programming like mathematical operations.

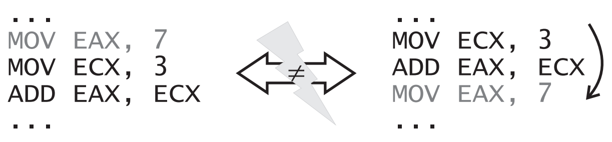
[1]In order to cover stateful programs, the input set may also comprise sequences of input data.



(a) In Symbolic Regression

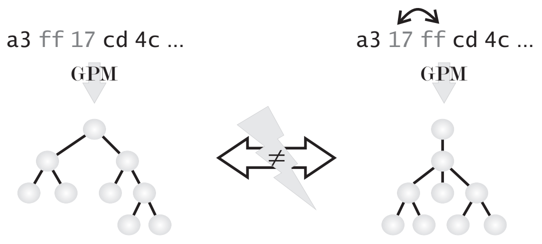


(b) In Standard Genetic Programming.



```
...                     ...
MOV EAX, 7              MOV ECX, 3
MOV ECX, 3              ADD EAX, ECX
ADD EAX, ECX            MOV EAX, 7
...                     ...
```

(c) In Linear Genetic Programming



(d) With Genotype-Phenotype Mapping, as in Grammatical Evolution-like approaches.

Figure 1: Positional epistasis in Genetic Programming.

3. RULE-BASED GENETIC PROGRAMMING

We illustrate our new approach to Genetic Programming [40] by the example shown in Figure 2. Like in Pitt-style learning classifier systems, our programs consist of arbitrary many rules. A rule evaluates the values of some symbols in its condition part (left of $\Rightarrow$) and, in its action part, assigns a new value to one symbol or performs any other procedure specified in its setup.

3.1 Genotype and Phenotype

Before the evolution begins, the number of symbol and their properties must be specified as well as the possible actions. Each symbol identifies an integer variable, which is either read-only or read-write. Generally we define the constants `0` and `1` and the input symbol `start` which will only be `1` during the first execution of the program and then becomes `0`. Additionally, a program can access some general-purpose variables `var1` and `var2`. If we want to evolve distributed algorithms, we could add an input symbol `receive` where incoming messages will occur and a variable `send` from which

Symbol	Encoding
0	0000, 1100
1	0001, 1101
start	0010, 1110
id	0011, 1111
netSize	0100
receive	0101
send	0110
enter	0111
leave	1000
var1	1001
var2	1010

Comp.	Enc.
$>$	000
$\geq$	001
$=$	010
$\leq$	011
$<$	100
$\neq$	101
true	110
false	111

Concat.	Enc.
$\wedge$	0
$\vee$	1

Action	Enc.
= x+y	00
= x−y	01
= x	10
= 1−x	11

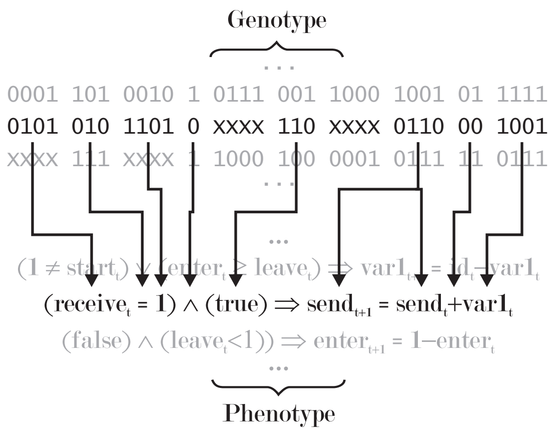

Genotype
...
0001 101 0010 1 0111 001 1000 1001 01 1111
0101 010 1101 0 xxxx 110 xxxx 0110 00 1001
xxxx 111 xxxx 1 1000 100 0001 0111 11 0111
...
...
$(1 \neq \text{start}_t) \vee (\text{enter}_t \geq \text{leave}_t) \Rightarrow \text{var1}_{t+1} = \text{id}_t - \text{var1}_t$
$(\text{receive}_t = 1) \wedge (\text{true}) \Rightarrow \text{send}_{t+1} = \text{send}_t + \text{var1}_t$
$(\text{false}) \wedge (\text{leave}_t < 1)) \Rightarrow \text{enter}_{t+1} = 1 - \text{enter}_t$
...
Phenotype

Figure 2: Genotype-Phenotype mapping in Rule-based Genetic Programming.

outgoing messages could be transmitted. An action set containing addition, subtraction, value assignment, and some sort of logical negation (`1-x`) is sufficient in most cases. Alternatively to the `send` and `receive` symbol, actions could be defined with the same semantics.

From these specifications, the system can determine how many bits are needed to encode a single rule. The genotypes are bit strings with a length which is a multiple of this bit count.

With this simple genotype, we can encode any possible nesting depth of condition statements, implicit branches and loops, and all possible logical operations. Furthermore, we could even construct a tree-like program structure from the rules, since each of them corresponds to a single `if` statement in a normal programming language.

There are similarities between our RBGP and some special types of LCS, like Browne's abstracted LCS [5] and S-expression-based LCS [26]. The two most fundamental differences lie in the semantics of both, the rules and the approach: In RBGP, a rule may directly manipulate symbols and invoke external procedures with (at most) two in/out-arguments. This includes mathematical operations like multiplication and division which do not exist a priori in LCS. They would have to evolve on basis of binary operations, which is, although possible, very unlikely.

Furthermore, the individuals in RBGP are not classifiers but programs. Classifiers are intended to be executed once for a given situation, judge it, and decide upon an optimal output. A program on the other hand runs independently and performs an asynchronous and interactive computation with its environment. Furthermore, the syntax of RBGP is very extensible. The nature of the symbols and actions is not bound to specific data types, our approach can for example easily be adapted to floating point computation.

3.2 New Dimensions of Independence

In the simple case, the evolved programs would be continuously executed in a cycle on a real machine. In a more efficient scenario, rules would explicitly be triggered whenever one of the concerning symbols changes. This is especially a useful technique for distributed algorithms.

3.2.1 Positional Independence

During the (simple) execution of a program coded in this scheme, all rules are applied consecutively and their assignments are buffered. Before the next execution cycle, these values are written back to the symbols. Therefore, the symbols in the condition part and in the computation parts of the actions are annotated with the index t and those in the assignment part of the actions are marked with $t+1$. This approach allows for a greater amount of disarray in the rules, since the only possible positional dependencies left are those of rules that write to the same variables. All other rules can be freely permutated without any influence on the behavior of the program. Therefore, epistasis in RBGP is very low.

3.2.2 Cardinality Independence

By excluding any "learning" features like the bucket brigade algorithm [17] from our evolution, we additionally gain some form of insensitivity in terms of rule cardinality. It is irrelevant whether a rule occurs once, twice, or even more often in a program because, if triggered, all occurrences of the rule will write the same values to the target symbol. An additional objective function which puts pressure into the direction of smaller programs will cause superfluous appearances of rules to be wiped out during the course of the evolution.

3.2.3 Neutrality

The existence of neutral reproduction operations can have a positive influence on the evolutionary progress [36, 37]. The positional and cardinality independence are a clear example of phenotypic redundancy and neutrality in RBGP. If the condition part of a rule is for example useful, it can simple be duplicated by copying the whole rule. This is likely to happen during crossover, without changing any functionality. Subsequent mutation operations may now modify the action part of the rule and lead to improved behavior.

3.2.4 Crossover and Mutation

As reproduction operators, we can apply all the standard operations known from genetic algorithms like single-point and multi-point crossover and mutation [13, 16]. The interesting fact is that crossover applied to two identical programs will usually yield a valid program with the exception of only the small fraction where remaining positional restrictions are violated. If two good programs are recombined, the result will most probably be a new good program. In standard Genetic Programming and most of the sophisticated approaches mentioned in the related work, the result will usually be an invalid program and only very rarely be reasonable [15, 30, 2]. As for the RBGP mutation operators, we cannot easily argue on a better performance yet, but there are no indications for a worse efficiency either.

4. THE DISTRIBUTED CRITICAL SECTION

In the past we have introduced new applications of Genetic Programming in the area of distributed computing. Among these are automated aggregation protocol synthesis [43] and the breeding of election algorithms [42, 41]. We now test our new approach on a hard problem in the same field, the distributed critical section.

Sharing resources in asynchronous systems always holds many dangers. If two or more processes for instance simultaneously access common variables and at least one of them writes, phenomena like "lost update" or other data inconsistencies may result. Therefore, software engineers must ensure *mutual exclusion*, which means that at most one process may access the resource at a time. This is an interesting problem which has been examined first in the 1960s. The first decentralized algorithms solving this problem were developed by Dekker [8] and Dijkstra [9]. Code that accesses a shared resource is called a *critical section* and the solutions of Dekker and Dijkstra used (shared) control variables that ensured that at most one process could execute such code at a time.

In a distributed system, solving this problem is more cumbersome since no common memory exists. Instead, the processes running concurrently on different nodes have to communicate by the means of message exchange. Based on the messages sent and received, a process has to decide whether it is allowed to access the critical section or whether it has to wait. The first algorithms for mutual exclusion at a distributed critical section were introduced by Lamport [25] and Ricard and Agrawala [32], followed by Maekawa's solution [27], which is optimal in the number of exchanged messages.

4.1 Prerequisites

In principle, all the primitives that we need in order to specify a valid solution for mutual exclusion at a distributed critical section are already illustrated in Figure 2. We assume a network of m nodes. In order to tackle the critical section problem, these nodes must be equipped with some basic abilities:

- A node must be able to conditionally perform primitive arithmetic operations. This is given by the four introduced actions.
- The nodes must be able to communicate. If we restrict the solutions to simple algorithms that exchange messages which contain only a single integer number, one `send` and one `receive` symbol will suffice. If a non-zero value is written to $\texttt{send}_t$, it will become the value of $\texttt{send}_{t+1}$ in the next program execution and be transmitted as broadcast to all nodes in the network. In the following execution, $\texttt{send}_{t+2}$ will be zero again. Messages sent may be delayed for an arbitrary amount of time. We provide each node with a message queue of limited size. Whenever a message arrives, it is queued. Before each program execution t, the top of queue value is made available in the read-only symbol $\texttt{receive}_t$.
- Whenever an algorithm decides that the node it is running on may access the critical section, it will set the symbol `enter` to a non-zero value. From the following execution on, the node will be in the critical section for a time defined by the underlying simulation system. When this time is elapsed and the critical section is left, the read-only symbol `leave` will be set to 1. Whatever the algorithm writes in the meantime to `enter` plays no role.
- Each node has an unique identifier, stored in the symbol `id`.
- Another read-only symbol called `start` will be 1 exactly in the first execution of an algorithm and 0 in all further executions so the nodes know when they are started.
- Finally, we add two multi-purpose variables, `var1` and `var2`.

We will execute the evolved programs in a simulation environment in order to evaluate their utility. This is easy since the structure of RBGP programs is very simple and especially suitable for simulations and stepwise interpretation. In the simulation we have m simulated nodes running asynchronously at approximately the same speed. This speed however differs slightly from node to node and cannot be considered as constant either. The communication between the nodes is also asynchronous and has a pseudo-random delay. A node in principle executes its program repetitively in an infinite loop. The critical section is monitored by the simulation core, which also decides on a pseudo-random basis how long a node may remain in it. The simulation core does not prevent multiple nodes from accessing the critical section, but always records the number of processes n_{cs} in it for each global time step t. The pseudo-random numbers used by the system are the same in each simulation so the results are reproducible and comparable.

4.2 Objective Functions

The first objective function f_1 imposed on the program evolution should of course be related to the number of violations of the mutual exclusion-criterion. Here we simply accumulate the time steps where more than one process accessed the critical section. To increase the pressure, we sum up the square of the number of nodes inside the CS.[2]

Solely using this function to drive the evolution would lead to programs that never access the critical section, since this is the easiest way to ensure that no errors can occur. Hence, we have to define a secondary objective function f_2 which forces the nodes to enter the CS. In principle, f_2 represents the number of times each process could enter the critical section *at least* in the fixed time span of the simulation. This not only furthers fairness but is simple needed, because otherwise, programs will evolve which allow only one single node to enter the critical section. Such programs will then have $f_2 = 0$ and will not survive the selection process. Yet it makes sense to add a value in $[0, 1)$ proportional to the total number of accesses of the CS so even among such programs, an efficiency hierarchy is formed.

In order to promote the eradication of doubled genes in the programs and for putting pressure into the direction of

[2]Another reason for this is that $n_{cs} > 1$ represents an error, and it is common to minimize square-error terms in many statistical applications in order to achieve maximum likelihood estimation [40].

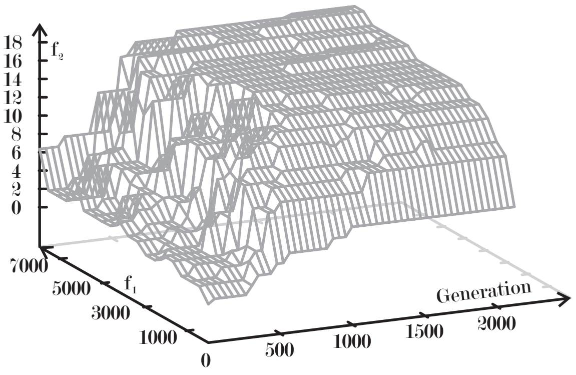


Figure 3: The relation of f_1 and f_2 during an example evolution (f_3 is ignored).

small algorithms, we define a third objective function f_3 whose value is the number of rules in a program. f_1 and f_3 are subject to minimization, f_2 is to be maximized.

$$f_1(P) = \sum_{t=1}^{T} \begin{cases} 0: & \textit{if } n_{cs}(t) \le 1 \\ (n_{cs}(t))^2: & \textit{otherwise} \end{cases} \quad (1)$$

$$f_2(P) = \min\{\textit{node in cs}\} + 1 - \frac{1}{\sum(\textit{node in cs})} \quad (2)$$

$$f_3(P) = |P| \quad (3)$$

4.3 Results

For our experiments we have chosen to run an elitist, steady-state genetic algorithm with population size 7177, archive size 89, and tournament selection. Nine nodes were running in the simulation for 1000 time steps in order to test every evolved program. Figure 3 illustrates the progress of the Pareto-optimal front in the dimensions f_1 and f_2 in generations. Surprisingly, the RBGP evolution does not converge. The experiment was terminated after more than 2100 generations, but even until then, improvements can still be observed.

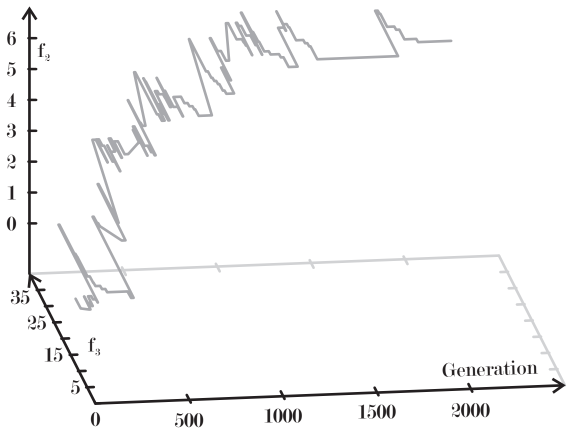


Figure 4: The relation of f_2 and f_3 ($f_1 = 0$) during an example evolution.

Only programs P with $f_1(P) = 0$ can be considered as valid solutions for the critical section problems. The third dimension (f_3) of the Pareto-front is depicted in Figure 4 in conjunction with f_2 for these valid solutions only ($f_1 = 0$). Here we can see that whenever an improvement in the number of accesses to the critical section (f_2) occurs, a deterioration in the number of rules in the program (f_3) can be observed. Successively, this degeneration is reduced until a program with the same functionality but much smaller size has evolved. Then the process starts all over. This is an indication for an efficient crossover operation since crossover is the only operation that changes the program size. A good crossover alone would probably not suffice to drive an evolutionary process for such a long time since it cannot create new rules. We assume that the ability to explore new regions of the search space of RBGP is due to a mutation operation, which is less destructive than in other GP approaches.

```
[useless]
(0≤id_t)∨(var1_t>enter_t) ⇒ var2_{t+1}=var2_t+id_t
(var1_t≤0)∧(send_t≠var2_t) ⇒ send_{t+1}=1-leave_t
[useless]
(enter_t<enter_t)∨(var1_t>receive_t) ⇒
    start_{t+1}=start_t-receive_t
[useless]
[useless]
[useless]
(start_t<0)∨(leave_t=0) ⇒ enter_{t+1}=start_t
(receive_t>enter_t)∧true ⇒ enter_{t+1}=enter_t-1
(netSize<enter_t)∨(var2_t>id_t) ⇒
    start_{t+1}=start_t-receive_t
[useless]
(1≤leave_t)∧(enter_t>leave_t) ⇒
    start_{t+1}=start_t+var2_t
(id_t>receive_t)∧(leave_t≠netSize) ⇒
    enter_{t+1}=0
(enter_t≤enter_t)∧(var2_t≠1) ⇒
    id_{t+1}=id_t-start_t
(netSize≤var2_t)∧true ⇒ id_{t+1}=id_t+var1_t
(1≥netSize)∨(receive_t=var1_t) ⇒
    id_{t+1}=id_t-enter_t
(netSize=var2_t)∧true ⇒ id_{t+1}=id_t+var1_t
[useless]
(var1_t≥netSize)∨(enter_t=var1_t) ⇒
    id_{t+1}=id_t-enter_t
[useless]
(start_t<id_t)∨(receive_t≠var1_t) ⇒
    var1_{t+1}=var1_t+receive_t
[useless]
(receive_t≤netSize)∧(id_t<var1_t) ⇒
    var2_{t+1}=var2_t-receive_t
(leave_t<id_t)∧(var2_t<send_t) ⇒
    enter_{t+1}=1-start_t
```

Listing 1: The best individual of the evolution.

Listing 1 illustrates the best solution $P^\star$ found in the evolutionary process. It grants full mutual exclusion, i.e. $f_1(P^\star) = 0$, under the simulated conditions. Out of its $f_3(P^\star) = 25$ rules, only 16 have a functional effect. This shows that even after more than 2000 generations, the RBGP population still has potential for improvement (at least in f_3). In the 1000 simulation steps where the single nodes approximately executes the program 500 times, each node enters the critical section at least 6 times ($f_2(P^\star) \approx 6.99$). The evolved distributed algorithm realizes a *Time Division Multiple Access* (TDMA) scheme [24] to the critical section as sketched in Figure 5. A schedule is derived with a complicated cooperative arithmetic computation determining the timeslots.

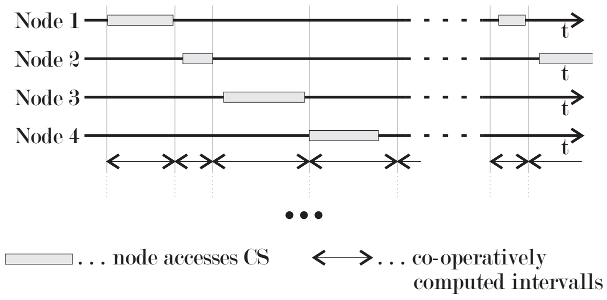


Figure 5: The TDMA scheme of Listing 1.

Further tests reveal, however, that some form of overfitting took place and the algorithm specializes on the settings of the simulation. It may not deliver full mutual exclusion in networks with different topologies. Thus, without modifications, the experiment can only be used for real-time systems where timing constraints are fixed.

An intermediate solution of the evolutionary process is displayed in Listing 2. It allows each process only to enter the critical section one time. This access occurs in the first rule, which sets the symbol `enter` to a negative value if the `start` symbol is equal to `enter` (which will only happen if both are zero) and the `id` reaches a value smaller than or equal to one. The symbol `id` initially contains the unique identifier of the node executing the algorithm. It is then modified by the rules 8 and 9. Together with the values of the two variables (`var1` and `var2`) and `start`, it oscillates through positive and negative integers due to overflows following no obvious schema.

4.4 Correctness and Discussion

The results shown in Listing 1 and Listing 2, although containing only few rules, are very hard to understand for a human being. Even the short algorithm of Listing 2 exhibits complex interrelations between the variables, the rules, and the message transfers. The communication between the nodes in Listing 2 for example plays an unclear role. In rule 3, a node sends a message unless it finds an incoming one in its receive buffer. The communication medium is thus permanently used. Due to parallelism, delays, and the inconstant execution speed of the nodes, only the senders alternate. Three other rules depend on this communication

```
1 (enter_t=start_t) ∧ (1≥id_t) ⇒
     enter_{t+1}=enter_t-netSize
2 true ∨ false ⇒ start_{t+1}=var1_t
3 true ∨ false ⇒ send_{t+1}=1-receive_t
4 true ∨ false ⇒ var1_{t+1}=var1_t+start_t
5 (start_t≥0) ∧ true ⇒ var1_{t+1}=netSize
6 (send_t<var2_t) ∨ (start_t≥receive_t) ⇒
     var1_{t+1}=var1_t+id_t
7 true ∨ false ⇒ var2_{t+1}=var2_t+start_t
8 (id_t<leave_t) ∨ false ⇒ id_{t+1}=send_t
9 (leave_t>send_t) ∨ (var2_t>netSize) ⇒
     id_{t+1}=id_t+var2_t
```

Listing 2: An intermediate solution.

and influence the values of `id` and `var1`.

Here we seemingly witness some form of *emergence* – our system works, but there is no simple way for finding out why by just using the semantics of the single rules. This is because of the fact that there is no kind of *intention* in the code and no structured design process in its creation. The programs are the result of a stochastic process, the artificial evolution. This is very interesting from the scientific point of view, but it poses the question of correctness.

Genetic programming only incorporates a fraction of the possible scenarios when evaluating a RBGP program. These are chosen in a way that the nodes run in parallel pseudo-randomly and messages have pseudo-random delays. There can be no guarantee for the full correctness of an evolved solution – it is just *very likely* to be correct. This is, of course, not only an issue in this example. The results of RBGP for other problems will probably be emergent too.

The simple structure of RBGP programs suggests that additional, automated methods could be used to fully prove their correctness. An automated tool will not work better or worse just because an issue is formulated in a complicated way, as long as this formulation complies with its input grammar. There are many tools available for automated, formal proofs [10], and we consider it to be a topic of future work to apply such a tool to the output of our algorithm.

However, the application of an automated prover during the evolutionary cycle is not advisable since we then would only have two fitness cases: right and wrong. If we use simulations though, the evolution can give a program preference to another one if it produces fewer errors, i.e. violations of the critical section. The output of the RBGP system nevertheless should be verified.

5. CONCLUSIONS AND FURTHER WORK

In this paper, we have presented a new method of Genetic Programming, called rule-based Genetic Programming, RBGP in short. RBGP extends the idea of learning classifier systems with more distinct semantics in terms of symbols, logical and mathematical operations, as well as primitive actions. By making the sets of actions and symbols an input parameter of the evolutionary process, arbitrary new commands can be introduced. Thus, RBGP can easily be adapted to new problem domains.

The basic purpose of RBGP is to address the inefficiency of reproduction operators in Genetic Programming. It does so by eliminating positional and cardinality interdependencies in its geno- and phenotypes, thus reducing epistasis. In future, a detailed performance comparison with other approaches like those named in related work must be issued in order to determine the utility of RBGP more precisely.

We still need to support our assumptions about the genetic operators with in-depth analysis and targeted experiments. In this paper, we concentrated on the introduction, justification, and discussion of its basic properties instead. The solutions of RBGP exhibit very strong emergence and non-trivial interactions between the positional independent rules. We need to incorporate tools that are able to prove the correctness of such evolved programs in order to make the RBGP approach viable for real-world applications.

The utility of the approach could already be shown here by solving a hard problem in distributed computing. A valid algorithm for mutual exclusion at the distributed critical section could be evolved. In the experiment, a very smooth and prolonged evolutionary process could be observed. The solution finally found depends however on the network configuration in the simulation. In our future experiments, we will prevent such overfitting by changing the test cases after each generation. By doing so, an overspecialized solution yielding good objective values in one iteration will probably be exterminated in the next generation and only "real" solutions can prevail.

6. REFERENCES

[1] W. Banzhaf. Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In *Parallel Problem Solving from Nature III*, volume 866.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction – On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, first edition, Nov. 1997.

[3] W. Bateson. *Mendel's Principles of Heredity*. Cambridge University Press, Cambridge, 1909. 1930: fourth impression of the 1909 edition.

[4] 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, 2001.

[5] W. N. Browne and C. Ioannides. Investigating scaling of an abstracted lcs utilising ternary and s-expression alphabets. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2759–2764, New York, NY, USA, 2007. ACM Press.

[6] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, July 1985. Lawrence Erlbaum Associates, Inc.

[7] M. de la Cruz Echeandía, A. O. de la Puente, and M. Alfonseca. *Attribute Grammar Evolution*, volume 3562/2005 of *Lecture Notes in Computer Science*, pages 182–191. Springer Berlin / Heidelberg, June 2005.

[8] E. W. Dijkstra. Cooperating sequential processes. Technical report, Techniche Hogeschool Eindhoven, Techniche Hogeschool Eindhoven, 1965. About Dekker's Algorithm. Reprinted in: F. Genuys (ed.), Programming Languages, Academic Press, 1968, 43–112.

[9] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[10] M. L. et al., editor. *Proceedings of Automated Formal Methods 2006 (AFM 2006)*, Aug. 2006. Workshop on the tools PVM, SAL, and Yices.

[11] R. M. Friedberg. A learning machine: Part i. *IBM Journal of Research and Development*, 2:2–13, Nov. 1958.

[12] R. M. Friedberg, B. Dunham, and J. H. North. A learning machine: Part ii. *IBM Journal of Research and Development*, 3(3):282–287, Mar. 1959.

[13] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, first edition, 1989.

[14] N. X. Hoai. Solving trigonometric identities with tree adjunct grammar guided genetic programming. In A. Abraham and M. Köppen, editors, *Hybrid Information Systems, Proceedings of First International Workshop on Hybrid Intelligent Systems*, pages 339–351, Dec. 2001.

[15] N. X. Hoai. *A Flexible Representation for Genetic Programming: Lessons from Natural Language Processing*. PhD thesis, School of Information Technology and Electrical Engineering University College, University of New South Wales, Australian Defence Force Academy, Dec. 2004.

[16] J. Holland. Genetic algorithms. *Scientific American*, 267(1):44–50, July 1992.

[17] J. H. Holland. Properties of the bucket brigade algorithm. In *Proceedings of the International Conference on Genetic Algorithms and Their Applications*, pages 1–7, Pittsburgh, PA, 1985.

[18] J. H. Holland and A. W. Burks. Adaptive computing system capable of learning and discovery. US Patent Issued on September 29, 1987, Current US Class 706/13, Genetic algorithm and genetic programming system 382/155, LEARNING SYSTEMS 706/62 MISCELLANEOUS, Foreign Patent References 8501601 WO Apr., 1985.

[19] J. H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern directed inference systems*, pages 313–329. Academic Press, New York, NY, 1978. Reprinted in: Evolutionary Computation. The Fossil Record. David B. Fogel (Ed.) IEEE Press, 1998. ISBN: 0-7803-3481-7.

[20] H. Hörner. A c++ class library for gp: Vienna university of economics genetic programming kernel (release 1.0, operating instructions). Technical report, Vienna University of Economics, May 29 1996.

[21] R. E. Keller and W. Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116–122, Stanford University, CA, USA, 1996. MIT Press.

[22] R. E. Keller and W. Banzhaf. Genetic programming using mutation, reproduction and genotype-phenotype mapping from linear binary genomes into linear lalr(1) phenotypes. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 116–122, Stanford University, CA, USA, Jan. 1996. MIT Press.

[23] J. R. Koza. *Genetic Programming, On the Programming of Computers by Means of Natural Selection*. A Bradford Book, The MIT Press, Cambridge, Massachusetts, 1992 first edition, 1993 second edition, 1992.

[24] S. S. Lam. Delay analysis of a time division multiple

access (tdma) channel. *IEEE Transactions on Communications (legacy, pre - 1988)*, 25:1489–1494, Dec. 1977. NASA STI/Recon Technical Report A, volume 78, December 1977, pp. 19522.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[26] P. L. Lanzi and A. Perrucci. Extending the representation of classifier conditions part ii: From messy coding to s-expressions. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 99)*, pages 345–352. Morgan Kaufmann, July 1999.

[27] M. Maekawa. An algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[28] B. Naudts. *Measuring GA-hardness*. PhD thesis, Antwerpen, Netherlands, 1998.

[29] P. Nordin. A compiling genetic programming system that directly manipulates the machine code. pages 311–331, 1994.

[30] P. Nordin and W. Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, July 1995. Morgan Kaufmann.

[31] M. O'Neill. Grammatical evolution. In *Proceedings of the Fifth Research Conference of the Deptartment of Computer Science and Information Systems, University of Limerick*, Sept. 1998.

[32] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, Jan. 1981.

[33] S. Ronald. Robust encodings in genetic algorithms: A survey of encoding issues. pages 43–48, Apr. 1997.

[34] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, 1998. Springer-Verlag.

[35] C. Ryan, M. O'Neill, and J. J. Collins. Grammatical evolution: Solving trigonometric identities. In *Proceedings of Mendel 1998: 4th International Mendel Conference on Genetic Algorithms, Optimisation Problems, Fuzzy Logic, Neural Networks, Rough Sets*, pages 111–119, Brno, Czech Republic, 1998. Technical University of Brno, Faculty of Mechanical Engineering.

[36] R. Shipman. Genetic redundancy: Desirable or problematic for evolutionary adaptation? In *Proceedings of 4th International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA)*, pages 1–11. Springer-Verlag, 1999.

[37] R. Shipman, M. Shackleton, and I. Harvey. The use of neutral genotype-phenotype mappings for improved evolutionary search. *BT Technology Journal*, 18(4):103–111, 2000.

[38] S. F. Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, University of Pittsburgh, 1980.

[39] W. M. Spears and K. A. De Jong. Using genetic algorithms for supervised concept learning. In *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence*, number IEEE Cat. No. 90CH2915-7, pages 335–341, Herndon, VA, 6-9 1990. IEEE Computer Society Press, Los Alamitos, CA.

[40] T. Weise. *Global Optimization Algorithms – Theory and Application*. July 2007 edition, July 2007. This e-book is online available at `http://www.it-weise.de/`.

[41] T. Weise and K. Geihs. Dgpf – an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In B. Filipič and J. Šilc, editors, *Proceedings of the Second International Conference on Bioinspired Optimization Methods and their Application, BIOMA 2006*, International Conference on Bioinspired Optimization Methods and their Application (BIOMA), pages 157–166. Jožef Stefan Institute, Ljubljana, Slovenia, Oct. 2006.

[42] T. Weise and K. Geihs. Genetic programming techniques for sensor networks. In *Proceedings of 5. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, pages 21–25, University of Kassel, July 2006.

[43] T. Weise, K. Geihs, and P. A. Baer. Genetic programming for proactive aggregation protocols. In B. Beliczyński, A. Dzieliński, M. Iwanowski, and B. Ribeiro, editors, *Proceedings of the 8th International Conference on Adaptive and Natural Computing Algorithms ICANNGA'07, Part 1*, volume 4431 of *Lecture Notes in Computer Science (LNCS)*, pages 167–173. Springer Berlin Heidelberg New York, Apr. 2007.

[44] P. A. Whigham. Inductive bias and genetic programming. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 461–466, Sheffield, UK, 12–14 1995. IEE.

[45] M. L. Wong and K. S. Leung. Combining genetic programming and inductive logic programming using logic grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 733–736, Perth, Australia, Nov. 1995. IEEE Press.