An Evolution of a Complete Program using XML-based Grammar Definition

Nor Zainah Siau, Christopher J. Hinde and Roger G. Stone
Department of Computer Science, Loughborough University, Loughborough, U.K.

Keywords: Genetic Programming, XML-based Grammar Definition, Complete Program Evolution.

Abstract: XML technology is a technique to describe structured data that can be manipulated by different types of applications, especially to represent content on the Web. This paper presents a viable approach to automatically evolve a 'sorting program' by applying genetic programming and full syntax XML-based grammar definition to map the genotype to phenotype. The genotypes are composed of fixed-length blocks of genes that are made up of a series of integer values. The paper reports that our approach improves the structure of the grammar used in the mapping process, which guarantees that the generated program follows the correct syntax with no repair function, in comparison to earlier work. This allows more structured programs than earlier systems.

1 INTRODUCTION

Over the years, queries for specific information from online sources have increased significantly. However, not all information is well defined that could be automatically searched and extracted. This paper presents our initial work, contributing to a bigger research of producing a teachable web information extraction (TWIE) system. TWIE aims to capture specific pieces of information on the Web with some assistance from a human, by evolving regular expressions. Typically the human will point to a typical item and the system will evolve a suitable expression to locate it and similar items. The regular expression notations are defined as rules in XML form to match the DOM tree structure and the data pattern of the information.

Earlier GP, popularised by John Koza (1992), automatically generates computer programs and evaluates them to solve a user-defined problem. More recently, GP has been used to solve problems in various fields, such as: medical (Guo and Nandi 2006); (Hong and Cho, 2004), Railway platform allocation (Clarke et al., 2010), robotics (Konig and Schmeck, 2009), and symbolic regressions (Castillo et al., 2005).

In this paper, we build a programming language subset grammar defined in a XML format to guide the creation of a 'sorting' program. Specifically, we are focusing on the Genotype-Phenotype mapping, which is part of GP. This XML-based grammar presents the basic construct of programming syntax arranged in a hierarchical form of rules and elements, which ensure the translation of the genomes into valid program codes (phenome).

The remainder of this paper focuses on describing related research, followed by our approach to extend and improve the work of Withall, Hinde and Stone, (2009) and Xhemali et al., (2010). Details of the experiment and the result are described in section 5 and finally, conclusions are drawn.

2 RELATED WORK

A genotype refers to the search space, which represents a potential solution to a problem, whereas a phenotype refers to the solution space, where the instance is measured for fitness. The Genotype-Phenotype mapping has been applied in various fields of evolutionary computation such as Genetic Algorithms (Holland, 1975), Genetic Programming (Koza 1992) and Grammatical Evolution (Ryan et al., 1998).

The first work to introduce the separation of genome from phenome in the field of Genetic Programming was by Banzhaf (1994). With this separation, he emphasized the feasibility of the phenotype solution while the genomes may be

Zainah Siau N., J. Hinde C. and G. Stone R..
An Evolution of a Complete Program using XML-based Grammar Definition.
DOI: 10.5220/0004155502140219
In *Proceedings of the 4th International Joint Conference on Computational Intelligence* (ECTA-2012), pages 214-219
ISBN: 978-989-8565-33-4

modified by the genetic operators without constraints. However, it is important to establish a direct and consistent mapping (Withall et al., 2009). For these reasons, Xhemali introduced rules in XML format to define the mapping of genotype to phenotype.

The genotype in Banzhaf's method is fixed in length and represented as strings of 5-bit binary but the resulting phenomes could vary in length. Basically the mapping is done in two steps; raw translation of the bit-strings to a set of functions and terminals and applying a correction mechanism to ensure that they are syntactically valid and any errors are corrected. Unlike Koza's evolution system, which only works with the phenome, Banzhaf evolves the genomes, which allows for application of any kind of genetic operators. However, there was a certain amount of redundancy in the program.

Paterson and Livesey (1996) extend Banzhaf's work, introducing a different genotype representation, that is, a linear string of integers to map to the phenotype. Their method uses Backus Naur Form (BNF) grammar definition to represent rules of the programming subset and these integers are used to decide which rule to take to make up a complete program.

Another attempt to evolve program code was that of Ryan et al., (1998). Their method uses a linear, variable length genotype made up of a string of 8-bit binary numbers. Similar to Paterson and Livesey, the mapping is through a grammar defined in BNF format using a rule called *expr* as the starting point. The advantages of using BNF definition is that a system may be built independently of any programming language and a correcting mechanism is not necessary because the binary string maps directly onto the grammar definition. However, one important issue that arises during the mapping process is when the individuals run out of binary strings to produce a complete program. Although this is solved by binary reuse, it could result in a lack of inheritance of characteristics. This means a change early in the gene value of a genome (through crossover or mutation) can change the entire construct or type of statement following, which results in the child having little similarity to its parents.

Withall looked at the problem of characteristics inheritance, which saw the introduction of fixed length, linear block of integers to represent the genotype. He studied software evolution and evolved several programs including a 'sorting program'. Each block contains four genes, which translates to a single statement in the resulting phenotype. The blocks are padded with unused genes to avoid the problem of insufficient genes as can be seen in the work of Banzhaf and Ryan. Withall argues that the padding is useful to preserve characteristics of parents that can be inherited by the offspring. Therefore, in a case where a particular program structure or statement requires fewer genes, these unused genes will be ignored. This should ensure that the next statement/structure translation would start from the first gene of the next block; its interpretation would also be unchanged.

Xhemali later extended Withall's work by manipulating variable-length genotypes and introducing XML rules, to specify the mapping of genotype to phenotype. However this method poses a disadvantage of inheritance of characteristics and insufficient genes. Both authors also noted that the generated program has the possibility of having syntactically incorrect code segments, thus, a 'repair' function in the GP is included.

We have modified Withall's system, incorporating an improved version of Xhemali's XML system, greatly improving the construction of the grammar and ensuring that syntactically valid programs are produced without any intervention function (the 'repair' function). Hereafter, this improved grammar is referred to as 'clean' grammar.

3 A REPAIR FUNCTION

It is possible that the phenotype produced from the raw mapping of the genotype contains errors or incomplete elements to make up a valid program statement. This happens because individuals run out of genes required by a particular rule definition.

The repair function in both Withall's and Xhemali's system was performed after all the genes have been decoded. In Withall's proposed work, the grammar is coded in PERL and if we are to represent this grammar in BNF, it would look like in Figure 1. Note that this is not a full description of the grammar. Withall's grammar was inspired by trying to keep the number of non-terminals minimal, and this was used together with blocking to minimise the 'damage' caused by a single mutation.

The minimised grammar has the possibility of not mapping to the *end* statement to close any open bracket used in *if*, *for* and *doublefor*. This will cause imbalance '{}' pair, thus, a repair function would append any missing '}' at the end of the program and to ensure that the *end* statement is discarded if there is no prior map to a terminal '{'.

Moreover, in the case of Xhemali's work, not only it is used to do the above, but also to add the necessary operators e.g. '+' symbol indicating an addition, to deal with insufficient genes to complete a particular rule and to add the required variable declaration and variable return.

This hard-wired constraints put into the program would stop it being able to handle general grammars. Our aim is to remove the reliance on the repairing function while achieving a valid phenome just as successfully with a 'clean' grammar.

```
statement ::=null | assignment | if | for | doublefor | end
if ::= "if" "(" rvar op rvar ")" "{"
end ::= "}"
```

Figure 1: Withall's grammar in BNF notation.

```
statement ::= null | assignment | if | for | doublefor
statementseq ::= statement | statement statementseq
if ::= "if" "(" rvar op rvar ")" "{" statementseq "}"
```

Figure 2: 'clean' grammar in BNF notation used in our work. By having a proper grammar description in place as well as the introduction of *statementseq* and taking away the *end* statement, the repair function is eliminated.

4 EVOLUTION USING GP

In this paper, the evolution technique follows the standard GP set up (defined in Figure 3) and then moves on to some specific requirements. These include using the 'clean' grammar definition in the XML format to guide the Genotype-Phenotype mapping, the phenotype is translated into an executable form for fitness evaluation, and blocks of 5 genes translates to single program statements, thus ensuring similarity when mapped to statements.

```
1: Randomly create an initial population of individuals
2: Genotype-Phenotype mapping and evaluate fitness.
3: Repeat
4:     Select individuals from the population and apply
       genetic operations
5:     genotype-phenotype mapping, evaluate fitness of the
       new individuals and insert the new individuals in the
       next generation
6:   until an acceptable solution is found or it reaches a
     maximum number of generations).
7: return the best-so-far individual.
```

Figure 3: Genetic programming algorithms.

The following subsections describe the GP stages and parameters applied in this research.

4.1 Individuals Representation

We are using greedy initial population by seeding the random number generator to bias the search towards good solutions. The initial population consists of 7 genomes; each made up of 50 genes. Genes are made up of randomly created integers between 1 and 255. A genome is constrained by fixed-length blocks of genes, with each contains five genes. A block translates to a single program statement and the first gene of the block always represents the type of statement to follow.

4.2 Fitness Function

The fitness evaluation of the solution is measured by comparing the actual output produced by the algorithm against the expected output, like Koza's (Koza, 1992). In order to ensure fair comparisons, the fitness function proposed by Withall is used here (the function code in Figure 4 is in PERL). It is important to note that evaluation is based on comparison of adjacent elements in the list rather than all element pairs.

In contrast to the traditional fitness function where sample inputs are selected with known output, the fitness evaluation used here is derived from the formal specification of the desired function (sorting). We have had successful experiments using the formal specifications to define the complete and concise fitness functions, outperformed a simple input/output pair.

```
$fitness++ if(bageq(\@L, \@N));
if($#N > 0){
 for my $x (0..$#N-1){
          $fitness++ if($N[$x] <=
          $N[($x+1)]);
 }
}
```

Figure 4: Fitness function.

An individual is considered useful if it achieves 100% fitness value. This measurement is to determine the quality of individuals in the population for reproduction or being discarded, in comparison to other individuals in the population. In this experiment, the seven 'most fit' individuals will survive to reproduction at each cycle.

4.3 Reproduction

Reproduction creates the next generation of solutions (genomes), which ideally share many of the useful characteristics of their parents. During the reproduction process, the new individuals are generated with the aid of two genetic operators: uniform crossover, followed by mutation. The crossover involves all the genes of both parents' genotypes being combined from swapping the genotypes with a probability of 50% (Jones and Hinde, 2007). Mutation replaces the selected gene with a random integer between 1 and 255. Prior to the operators, the selection of the genomes is by Roulette Wheel Selection method.

4.4 XML-based Grammar Definition

```
<main>
  <statement dtype = "selection">
    <stmOption id = "nullstatement" />
    <stmOption id = "assignstatement" />
    <stmOption id = "ifstatement" />
    <stmOption id = "forstatement" />
    <stmOption id = "nestedforstatement" />
  </statement>
  <assignstatement dtype = "sequence">
    <item id = "wvariable" />
    <item id = "tokens">=</item>
    <item id = "rvariable" />
    <item id = "tokens">;</item>
  </assignstatement>
  <ifstatement dtype = "sequence">
    <item id = "tokens">if (</item>
    <item id = "rvariable" />
    <item id = "operator" />
    <item id = "rvariable" />
    <item id = "tokens">)</item>
    <item id = "tokens">{</item>
    <item id = "statementseq" />
    <item id = "tokens">}</item>
  </ifstatement>
  <statementseq dtype = "selection">
    <item id = "statement" />
    <item id = "statements" />
  </statementseq >
  <statements dtype = "sequence">
    <item id = "statement" />
    <item id = "statementseq" />
  </statements >
...
</main>
<rvariable dtype = "selection">
  <item id ="tokens">$inlist[$tmp1%($#inlist+1)]</item>
  <item id ="tokens">$inlist[$tmp2%($#inlist+1)]</item>
  <item id ="tokens">$tmp1</item>
  <item id ="tokens">$tmp2</item>
  <item id ="tokens">$tmp3</item>
  <item id ="tokens">$tmp4</item>
</rvariable>
<operator dtype = "selection">
  <item id ="tokens"><![CDATA[==]]></item>
  <item id ="tokens"><![CDATA[!=]]></item>
  <item id ="tokens"><![CDATA[>]]></item>
  <item id ="tokens"><![CDATA[<]]></item>
  <item id ="tokens"><![CDATA[>=]]></item>
  <item id ="tokens"><![CDATA[<=]]></item>
</operator>
...
```

Figure 5: Sample of our rules in XML form.

The rules are made up of unique programming statements structures containing terminal and non-terminal symbols. Some of them are precisely constructed, such as, a *doublefor* is represented in a form of *for counter (0..length){ for counter (N1+1..length) { statementseq }}*. Notice that *N1* is included in the second *for*. *N1* means that the *counter* from the previous *for* is to be reused. This is important because of the way a valid *nested for statement* is constructed to compare list's elements. The decision for a restricted construct is to reduce the search space and to ensure the statement's validity, thus speed up the fitness evaluation.

Figure 5 shows our rules coded in XML. A new rule introduced here is the *statementseq* rule, which leads to one of the two options; a single statement (*statement* rule) or two or more statements (*statements* rule). The *dtype* attribute of a rule is either a *selection* or *sequence*. The *selection* indicates the requirement of a gene to decide the rule candidate to take whereas "*sequence*" indicates that all rule components must be taken. Note that some of the rule's components are identified as '*tokens'*, which is a terminal symbol. This means that the symbol such as '*if ('* in the *ifstatement* rule, is to be taken as it is and no gene is required.

4.5 Genotype-phenotype

The algorithm in Figure 6 describes the steps of translating the genome to the phenome.

1. *Divide the genome into fixed blocks of genes. The block size is determined by the rule with the most information.*
2. *Define the number of rule candidates for a 'statement' from the grammar as x.*
3. *Take the first integer y of a block and calculate the remainder z = y % x. Note that '%' symbol indicates a modulo operator.*
4. *Select the corresponding zth rule from the rule candidates.*
5. *If a component of zth rule is a non-terminal, apply a reminder operator to the next available gene to determine the next rule to follow. Otherwise, if it is a terminal, take the value to the solution. Any unused genes in the block are treated as padding and they are skipped.*
6. *If the block completes, repeat step 3. If this is the last block, the translation ends.*

Figure 6: Genotype-phenotype mapping algorithm.

This process is best explained with an example. Table 1 shows a two-block genome with each containing 5 genes. Table 2 shows the genotype-phenotype mapping process using the rules in Figure 5 and the phenome produced from the mapping.

The first integer of the first block always represents a rule called statement. In this case, the first gene (7) is an *if* statement. Considering there are five rule candidates in the *statement* rule, so 7%5 selects the third option (*ifstatement*). Note that the index of each component begins from 0 and '%' is a symbol for a modulo operator. The *ifstatement*, which *dtype* is a *sequence*, has six mandatory components. The first component is a terminal called *'tokens'* and does not require any gene. The second component maps to a *rvariable* rule, which is a *selection*. *rvariable* has five components, therefore 23%5 is 3, selecting the fourth component. The third *ifstatement's* component is an *operator* rule, which takes the next available gene. So 11%6 is 5, which is equivalent to *'<='* symbol. Next is a *rvariable* rule, thus 34%5 is 4, which selects *$tmp3*. Following is a '*tokens*' and a *statementseq* rule. The last gene (2) in this block maps to *statement* rule from the *statementseq* rule. This completes the translation of the first block.

The first gene (6) of the next block is translated as the *assignstatement* rule with respect to 6%5 = 1. The *assignstatement* rule has three components; *wvariable* , = and *rvariable* which is translated to *'$tmp3 = $tmp1;'* . The extra 2 integers (21, 9) are referred to as the padding and they are ignored.

Table 1: 2 blocks genome.

7	23	11	34	2	6	10	12	21	9

Table 2: Genotype-phenotype translation.

Blk	Gene	%	Mapped to	Translation
1	7	2	statement	ifstatement
	-		tokens	if (
	23	3	rvariable	$tmp2
	11	5	operator	<=
	34	4	rvariable	$tmp3
	-		tokens	)
	-		tokens	{
	2	0	statementseq	statement
	-		tokens	}
2	6	1	statement	assignstatement
	10	2	wvariable	$tmp3
	-		tokens	=
	12	2	rvariable	$tmp1
	-		tokens	;
	21		-	padding
	9		-	padding
Phenome :	if (tmp2 <= $tmp3) { $tmp3 = $tmp1; }			

5 EXPERIMENT SETTING & RESULT

The experiment used a seeded initial population, using parameters setting as in Table 3 below.

Table 3: Parameter setting for 'sorting program' evolution.

Parameter	Specification
Population size	7
Selection	Roulette Wheel
Runs	100
Maximum Generations in each run	50,000
Fitness score target	40
Uniform crossover probability	50%
Mutation probability	10%
Machine	Intel 3.00GHz PC with 4GB of RAM, running Windows7
Input lists	[4 ,3 ,2 ,1] , [1,2,55,3] , [1,999,2,3] , [71,1,2,3] , [1,2,33] , [100,88,211] , [100,1,2] , [13,7] ,[5,55] , [10]

The input lists are made up of various lengths and orders. The termination of each run is either when the maximum generation is reached or earlier, if a solution is found. Table 4 shows the result of the first 10 runs with the initial population seeded with the first 10 prime numbers.

Table 4: Experiment result.

Seed	Generation	Time (hr:mm:ss)
1	9114	00:06:33
2	4407	00:03:12
3	27830	00:20:37
5	36028	00:26:01
7	24400	00:17:48
11	31384	00:22:57
13	31190	00:22:56
17	11928	00:09:00
19	35391	00:26:25
23	28154	00:20:44

The effect of moving the rules to an XML file and modification to the grammar definition is shown in Table 5, in comparison with the previous work of Withall et. al. (2009).

Table 5: Comparison of result against Withall.

	Generation		Time	
	Withall	Ours	Withall	Ours
N	100	100	100	100
Mean	15514.56	22906.53	10.50	16.55
Std. Error- Mean	1081.169	1546.987	.752	1.132
Median	12491.00	20527.50	9.00	15.50
Std. Deviation	10811.688	15469.870	7.524	11.316

Although the result shows some increase in the number of generations and time, our approach provides several benefits:

- The main contribution from this research was to remove the translation process from a hard coded system to a table driven approach.

- The 'repair' function that ensures the validity of the generated program is no longer required.
- The rules resemble the full program subset syntax, without any hidden terminal symbols.
- If the requirement is to generate blocks of a e.g. *{aaa}{aaaaaa}*, our method could easily produce this pattern as we specified the start and the stop of a block statement within the grammar. Withall's method would not be possible because of the repair function, which insert all the remaining *}* in the end of the program to match the *{* produced earlier in the program. Xhemali's similarly fails in this respect.

The above experiment is set to evolve a 'sorting program', however, the fitness evaluation function needs to be changed for other computer program problems. In addition, a domain specific grammar definition is needed to fit other areas such as regular expressions, Medical (e.g. DNA matching), linguistics (Natural Language) etc. However, further experiments are required to evaluate these applications.

6 CONCLUSIONS

This paper presents an investigation into the effect of full syntax XML-based grammar definitions to the resultant program and the fitness evaluations. Specifically, we have presented a novel approach to effectively map the genotype to phenotype with XML rules, demonstrated by evolving a sorting program. The results are compared to the former work and provide evidence of significant improvements in terms of the construction of a syntactically correct solution without a repair function and without significantly compromising performance.

In future, we will continue this investigation to include a function declaration e.g. a swap function in the grammar, which would speed up a sorting program evolution, and applying a similar technique to other domain such as regular expression, to identify a data pattern from a HTML web page for information extraction. This will enable our GP system to be extended by an external process, which can add to the XML rules without requiring a modification to the main GP system.

REFERENCES

Banzhaf, W., 1994. Genotype-Phenotype-Mapping and Neutral Variation: A case study in Genetic Programming. *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature, pp. 322-332.*

Castillo, F., Kordon, A., Sweeney, J., Zirk, W., 2005. Using genetic programming in industrial statistical model building. *Genetic programming theory and practice II, pp. 31-48.*

Clarke, M., Hinde, C. J., Withall, M. S., Jackson, T., Phillips, I. W., Brown, S., Watson, R., 2010. Allocating railway platforms using a genetic algorithm. *Research and Development in Intelligent Systems XXVI, pp. 421-434.*

Guo, H., Nandi, A. K., 2006. Breast cancer diagnosis using genetic programming generated feature. *Pattern Recognition, 39(5), pp. 980-987.*

Holland, J. H., 1975. Adaptation in natural and artificial systems. Ann Arbor MI: University of Michigan Press.

Hong, J. H., Cho, S. B., 2004. Lymphoma cancer classification using genetic programming with SNR features. *Genetic Programming, pp. 78-88.*

Jones, S., Hinde, C., 2007. Uniform Random Crossover. In *Proceedings of the 2007 workshop on Computational Intelligence.*

Konig, L., Schmeck, H., 2009. A Completely Evolvable Genotype-Phenotype Mapping for Evolutionary Robotics, *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO '09, pp. 175-185.*

Koza, J. R., 1992. *Genetic Programming*. Cambridge: MA: MIT Press.

Paterson, N. R., Livesey, M., 1996. Distinguishing genotype and phenotype in genetic programming. *Late Breaking Papers at the Genetic Programming, pp. 141-150.*

Ryan, C., Collins, J., O'Neill, M., 1998. Grammatical evolution: Evolving programs for an arbitrary language. *Genetic Programming, pp. 83-96.*

Withall, M. S., Hinde, C. J., Stone, R. G., 2009. An improved representation for evolving programs. *Genetic Programming and Evolvable Machines, 10(1), pp. 37-70.*

Xhemali, D., Hinde, C. J., Stone, R. G., 2010. Genetic evolution of sorting programs through a novel genotype-phenotype mapping. *Proceedings of the International Conference on Evolutionary Computation*, Valencia, Spain.