Recent Adventures in Grammatical Evolution

Michael O'Neill[1], **Anthony Brabazon**[2]

[1] University of Limerick, Ireland, michael.oneill@ul.ie
[2] University College Dublin, Ireland, anthony.brabazon@ucd.ie

Abstract. *We outline the Grammatical Evolution algorithm before an exposition of some of the more recent research and developments in this form of grammar-based Genetic Programming. Some potential avenues for further exploration are suggested.*

1. Introduction Grammatical Evolution (GE) [32, 26, 34, 38] is an evolutionary algorithm which can be used to evolve computer programs, rule-sets, or more generally sentences in any language. Rather than representing the programs as syntax trees (as in GP), a linear genome representation is used in conjunction with a grammar. Each individual (genome), a variable-length binary string, contains in its codons (groups of bits) the information to select production rules from a Backus Naur Form (BNF) grammar. BNF is a notation that represents the language in the form of production rules. It is comprised of a set of non-terminals that can be mapped to elements of a set of terminals, according to the production rules.

The GE system is inspired by the biological process of generating a protein from the genetic material of an organism. Proteins are fundamental in the proper development and operation of living organisms and are responsible for traits such as eye colour and height. The genetic material (usually DNA) contains the information required to produce specific proteins at different points along the molecule. For simplicity, consider DNA to be a string of building blocks called nucleotides, of which there are four, named A, T, G and C, for adenine, tyrosine, guanine, and cytosine respectively. Groups of three nucleotides, called codons, are used to specify the building blocks of proteins. These protein building blocks are known as amino acids, and the sequence of these amino acids in a protein is determined by the sequence of codons on the DNA strand. The sequence of amino acids is very important as it plays a large part in determining a protein's functional properties. In order to generate a protein from the sequence of nucleotides in the DNA, the nucleotide sequence is first transcribed into a slightly different format, a sequence of elements on a molecule known as RNA (ribonucleic acid). Codons within the RNA molecule are then translated to determine the sequence of amino acids that are contained within the protein molecule. The application of production rules to the non-terminals of the incomplete code being in GE is analogous to the role amino acids play when being combined together to transform the growing protein molecule into its final functional three-dimensional form.

The paper provides an introduction to the GE methodology, an overview of research to date in GE, and provides suggestions for future research avenues.

2. Grammatical Evolution When tackling a problem with GE, a suitable BNF (Backus Naur Form) grammar definition must initially be defined. The BNF can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared towards the problem at hand.

In GE, a BNF definition is used to describe the output language to be produced by the system. BNF is a notation for expressing the grammar of a language in the form of production rules. BNF grammars consist of *terminals*, which are items that can appear in the language, e.g. binary boolean operators `and`, `or`, `xor`, and `nand`, unary boolean operators `not`, constants, `true` and `false` etc. and *non-terminals*, which can be expanded into one or more terminals and non-terminals.

For example the grammar below can be used to generate boolean expressions, and `<expr>` can be transformed into one of three rules. It can become either `( <expr> <biop> <expr> )`, `<uop> <expr>`, or `<bool>`. A grammar can be represented by the tuple $\{N, T, P, S\}$, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. When there are a number of productions that can be applied to one element of N the choice is delimited with the '`|`' symbol. For example

```
N = { <expr>, <biop>, <uop>, <bool> }
T = { and, or, xor, nand, not,
      true, false, (, ) }
S = { <expr> }
```

And P can be represented as:

```
(A) <expr> ::= ( <expr> <biop> <expr> )
             | <uop> <expr>
             | <bool>

(B) <biop> ::= and
             | or
             | xor
             | nand

(C) <uop> ::= not

(D) <bool> ::= true
             | false
```

The code produced will consist of elements of the terminal set T. The grammar is used in a developmental approach whereby the evolutionary process evolves the production rules to be applied at each stage of a mapping process, starting from the start symbol, until a complete program is formed. A complete program is one that is comprised solely from elements of T.

As the BNF definition is a plug-in component of the system, it means that GE can produce code in any language thereby giving the system a unique flexibility. For the above BNF grammar, Table 1 summarises the production rules and the number of choices associated with each.

Table 1. The number of choices available from each production rule.

Rule Number	Choices
A	3
B	4
C	1
D	2

The genotype is used to map the start symbol onto terminals by reading codons of 8 bits to generate a corresponding integer value, from which an appropriate production rule is selected by using the following mapping function:

$$Rule = c \quad mod \quad r$$

where `c` is the codon integer value, and `r` is the number of rule choices for the current non-terminal symbol.

Consider the following rule from the given grammar, i.e., given the non-terminal `<biop>`, which describes the set of binary operators that can be used, there are four production rules to select from. As can be seen, the choices are effectively labelled with integers counting from zero.

```
(B) <biop> ::= and        (0)
             | or         (1)
             | xor        (2)
             | nand       (3)
```

If we assume the codon being read produces the integer 6, then

$$6 \quad mod \quad 4 \; = \; 2$$

would select rule (2) `xor`. That is, `<biop>` is replaced with `xor`. Each time a production rule has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

During the genotype-to-phenotype mapping process, it is possible for individuals to run out of codons, and in this case the *wrap* operator is applied which results in returning the codon reading head back to the first codon in the individual. As such codons are reused when wrapping occurs. This is quite an unusual approach in Evolutionary Algorithms as it is entirely possible for certain codons to be used two or more times. This technique of wrapping the individual draws inspiration from the gene-overlapping phenomenon that has been observed in many organisms [22].

In GE each time the same codon is expressed it will always generate the same integer value, but depending on the current non-terminal to which it is being applied, it may result in the selection of a different production rule. This feature is referred to as *intrinsic polymorphism*. What is crucial however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is the case because the same choices are made each time. It is possible that an incomplete mapping could occur, even after several wrapping events, and typically in this case the mapping process is aborted and the individual in question is given the lowest possible fitness value. The selection and replacement mechanisms then operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules repeatedly. For example, consider an individual with three codons, all of which specify rule 0 from below.

```
(A) <expr> ::= (<expr> <biop> <expr>) (0)
             | <uop> <expr>           (1)
             | <bool>                 (2)
```

Even after wrapping, the mapping process would be incomplete and would carry on indefinitely unless terminated. This occurs because the nonterminal `<expr>` is being mapped recursively by production rule 0, i.e., it becomes `( <expr> <biop> <expr> )`. Therefore, the leftmost `<expr>` after each application of a production would itself be mapped to a `( <expr> <biop> <expr> )`, resulting in an expression continually growing as follows: `( ( <expr> <biop> <expr> ) <biop> <expr> )` followed by `( ( ( <expr> <biop> <expr> ) <biop> <expr> ) <biop> <expr> )` and so on.

Such an individual is dubbed invalid as it will never undergo a complete mapping to a set of terminals. For this reason an upper limit on the number of wrapping events that can occur is imposed. During the mapping process therefore, beginning from the left hand side of the genome codon integer values are generated and used to select rules from the BNF grammar, until one of the following situations arise:

1. A complete program is generated. This occurs when all the non-terminals in the expression being mapped are transformed into elements from the terminal set of the BNF grammar.
2. The end of the genome is reached, in which case the *wrapping* operator is invoked. This results in the return of the genome reading frame to the left hand side of the genome once again. The reading of codons will then continue, unless an upper threshold representing the maximum number of wrapping events has occurred during this individual's mapping process.
3. In the event that a threshold on the number of wrapping events has occurred and the individual is still incompletely mapped, the mapping process is halted, and the individual is assigned the lowest possible fitness value.

To reduce the number of invalid individuals being passed from generation to generation, a steady state replacement mechanism is commonly employed. One consequence of the use of a steady state method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals. Alternatively, a repair strategy can be adopted, which ensures that every individual results in a valid program. For example, in the case that there are non-terminals remaining after using all the genetic material of an individual (with or without the use of wrapping) default rules for each non-terminal can be prespecified that are used to complete the mapping in a deterministic fashion. Another strategy is to remove the recursive production rules that cause an individual's phenotype to grow, and then to reuse the genotype to select from the remaining non-recursive rules.

2.1. Mapping Example Consider the following genome, represented as a series of integer-valued codons.

```
21 6 104 70 31 13 4 25 9 3 86 44
```

We will demonstrate the application of this genome to the grammar presented below, which could be used to generate a simplified trading system. The trading system has ten possible input variables (var0 $\rightarrow$ var9), and three trading signals can be generated by the system, buy, sell, or do nothing. The start symbol (`<S>`) for the grammar from which the mapping process commences is the non-terminal `<tradingrule>`.

```
<S> ::= <tradingrule>

<tradingrule>  ::= if( <signal> ) {
                           <trade>;
                   }
                   else {
                           <trade>;
                   }

<signal> ::= <value> <relop> <var>
           | (<signal>) AND (<signal>)
           | (<signal>) OR (<signal>)

<value> ::= <int-const> | <real-const>

<relop> ::= <= | >=

<trade> ::= buy
          | sell
          | do-nothing

<int-const> ::= <int-const><int-const>
              | 1 | 2 | 3 | 4 | 5
              | 6 | 7 | 8 | 9

<real-const> ::= 0.<int-const>

<var> ::= var0 | var1 | var2 | var3 | var4
        | var5 | var6 | var7 | var8 | var9
```

As there is only one production rule for `<tradingrule>` it is automatically replaced with its right-hand side. Our developing trading rule becomes:

```
if( <signal> ) { <trade>; }
else { <trade>; }
```

Taking the left-most non-terminal `<signal>` there are three possible replacements. The codon reading starts at the leftmost codon on the genome.

21 6 104 70 31 13 4 25 9 3 86 44

Reading the next codon value determines that we use the first production rule ($21 \; mod \; 3 = 0$), which allows `<signal>` to be replaced with `<value><relop><var>`. This results in the following:

```
if( <value> <relop> <var> )
  { <trade>; }
else { <trade>; }
```

Again, taking the left-most non-terminal `<value>` there are two choices. The next codon value dictates that we replace this non-terminal with an `<int-const>` (i.e., $6 \; mod \; 2 = 0$) giving:

```
if( <int-const> <relop> <var> )
  { <trade>; }
else { <trade>; }
```

According to the next codon value ($104 \; mod \; 10 = 4$) the non-terminal `<int-const>` is replaced with an integer (4).

```
if( 4 <relop> <var> )
  { <trade>; }
else { <trade>; }
```

The development of `<relop>` proceeds as follows:

$70 \; mod \; 2 = 0$ results in `<relop>` being replaced with `<=`. Reading the next codon value `<var>` is expanded by $31 \; mod \; 10 = 1$ leaving:

```
if( 4 <= var1 )
  { <trade>; }
else { <trade>; }
```

The first `<trade>` becomes *sell* by $13 \; mod \; 3 = 1$ with the second one being replaced with *buy* by $4 \; mod \; 4 = 0$

The fully expanded trading rule now has the form:

```
if( 4 <= var1 )
  { sell; }
else { buy; }
```

The final position of the codon reading head is illustrated by the bold character below.

21 6 104 70 31 13 4 **25** 9 3 86 44

The five leftover codons are unused during the mapping process and are simply ignored and consequently are referred to as introns as they do not impact on the function of the phenotype.
The variables (var0 to var9) could represent a selection of elements of information drawn from fundamental analysis of an industry sector, for example, var5 could be a price/earnings ratio for a company, and var3 could represent a company's sales growth over the past 3 years. Of course, successful real-world filter-rules for trading would not be as simple as this, and would typically contain multiple conditions.

3. Recent Adventures Since the publication of the first book on GE there has been a great deal of research conducted both by the authors in Ireland and internationally, including three international workshops on the subject [35, 36, 37]. In this section we will briefly outline some of the research that has taken place.

3.1. Grammar There have been a number of investigations into the use of different grammars as distinct from a straightforward context-free grammar. Most recently, the use of meta-grammars, that is grammars that describe other grammars, have been developed in the context of GE. Two initial studies presented the Grammatical Evolution by Grammatical Evolution or $(GE)^2$[33] and the mGGA (meta-Grammar Genetic Algorithm) [27]. The idea behind the use of a meta-grammar is to allow further abstraction and enable the creation of hierarchies and modularity in a convenient grammatical manner that can be adopted with GE. An example of a typical grammar adopted in the mGGA is given below.

```
<g> ::=
      "<bitstring> ::=" <reps>
           "<bbk4> ::=" <bbk4>
           "<bbk2> ::=" <bbk2>
           "<bbk1> ::=" <bbk1>
            "<bit> ::=" <val>

<bbk4> ::= <bbk4t>
         | <bbk4t> "|" <bbk4>
<bbk2> ::= <bbk2t>
         | <bbk2t> "|" <bbk2>
<bbk1> ::= <bbk1t>
         | <bbk1t> "|" <bbk1>
<bbk4t> ::= <bit><bit><bit><bit>
<bbk2t> ::= <bit><bit>
<bbk1t> ::= <bit>
<reps> ::= <rept>
         | <rept>  "|" <reps>
<rept> ::= "<bbk4><bbk4>"
         | "<bbk2><bbk2><bbk2><bbk2>"
         | "<bbk1><bbk1><bbk1><bbk1>
            <bbk1><bbk1><bbk1><bbk1>"
<bit> ::= "<bit>"
        | 1
        | 0
<val> ::= <valt>
        | <valt> "|" <val>
<valt> ::= 1
         | 0
```

Building blocks of size 1, 2, 4 and 8 bits are specified to be components of the solution grammar output as the result of mapping the above meta-grammar. For each building block size there can be many different building block instances represented as choices for that building block size in the solution grammar. An example bitstring solution grammar that could be produced by the above meta-grammar is provided below.

```
<bitstring> ::= <bit>11<bit>00<bit><bit>
              | <bbk2><bbk2><bbk2><bbk2>
              | 11011101
              | <bbk4><bbk4>
              | <bbk4><bbk4>

<bbk4> ::= <bit>11<bit>
         | 000<bit>

<bbk2> ::= 11
         | 00
         | <bit>1

<bbk1> ::= 0
         | 0

<bit> ::= 1 | 0 | 0 | 1
```

We can see in the above solution grammar that there are five possible building blocks of size eight (`<bitstring>`), two possible building block types of size four (`<bbk4>`) and three possible building blocks of size two (`<bbk2>`). Modularity exists above in the ability to specify the size and content (or partial content) of a building block through its incorporation into the solution grammar. This building block can then be repeatedly reused in the generation of the phenotype.

In addition to meta-Grammars there have also been studies on the use of context-sensitive grammars such as Attribute Grammars (indeed the ALP system was noted in the GE book, which adopts Logic Grammars), which have been explored in the context of formulating solutions to Knapsack problems [12, 13, 30]. Recently grammars have been used as a mechanism to represent constants in a Genetic Programming environment through Digit Concatenation and an alternative to Ephemeral Random Constants with Grammatical Persistent Random Constants [15, 16, 17].

3.2. Search Engine The authors of this paper have proposed and developed the use of a Particle Swarm Algorithm (PSA) in place of the standard Genetic Algorithm search engine for GE. The new algorithm represents a form of Social or Swarm Programming and is called Grammatical Swarm [28, 29]. Traditionally, PSAs have an underlying representation that is fixed in length. In the spirit of the variable-length structures that can be generated through Genetic Programming a variable-length PSA has been developed for the GS algorithm [21].

3.3. Genotype-Phenotype Map An alternative genotype-phenotype mapping algorithm has been proposed [31], which replaces the deterministic depth-first, left-right order of the mapping during the translation step. Instead the genome is consulted in order to decide the order in which the non-terminals will be expanded, the resulting system is called πGE (for position independent GE or piGE). It has been demonstrated that this alternative mapping clearly outperforms the standard GE genotype-phenotype map on a number of benchmark problem domains.

Using an XML implementation of GE there has been a study on exploiting the genotype-phenotype map to enable the identification and exploitation of derivation sequence building blocks to improve the efficiency of the evolutionary search [1].

3.4. Applications There have been a considerable number of applications of GE with some of the most notable being in the Financial Modelling domain [5], including the development of Trading Systems (on Index [14], Intra-day [4] and Foreign Exchange [6, 8] markets), Corporate Failure [7, 9, 10] and the Credit Rating of Bonds [11]. Other notable examples include solving Knapsack problems [12, 13], Complex Systems [3], Surface Generation [18, 19], digital circuit design with Verilog [20], Image Processing [25] Systems Biology and Bioinformatics [23, 24, 29].

4. Conclusions & Future Work In conclusion, Grammatical Evolution is a powerful grammar-based form of Genetic Programming that is increasing in popularity through its application to a large variety of problem domains and through continuous development of the underlying components of the algorithm. For further and more up-to-date information including code releases the reader is referred to the personal websites of the authors and http://www.grammatical-evolution.org.

References.

[1] Amarteifio S. (2005). Interpreting a Genotype-Phenotype Map with Rich Representations in XMLGE. MSc Thesis, University of Limerick.

[2] Amarteifio S., O'Neill M. (2005). Coevolving Antibodies with a Rich Representation of Grammatical Evolution. In Proceedings of CEC 2005, Edinburgh, Scotland. IEEE Press.

[3] Amarteifio S., O'Neill M. (2004). An Evolutionary Approach to Complex System Regulation. In Proceedings of Artificial Life IX. MIT Press.

[4] Brabazon A., Meagher K., Carty E., O'Neill M. and Keenan P. (2005). Grammar-mediated time-series prediction. Journal of Intelligent Systems 14(2-3).

[5] Brabazon A., O'Neill M. (2005). *Biologically Inspired Algorithms for Financial Modelling.* Springer.

[6] Brabazon A. and O'Neill M. (2004). Evolving Technical Trading Rules for Spot Foreign-Exchange Markets Using Grammatical Evolution, *Computational Management Science*, 1(3-4):311-327.

[7] Brabazon A. and O'Neill M. (2004). Diagnosing Corporate Stability using Grammatical Evo-

lution, *International Journal of Applied Mathematics and Computer Science*, 14(3):363-374.

[8] Brabazon, A. and O'Neill, M. (2002). Trading foreign exchange markets using evolutionary automatic programming. In Barry, A. M., editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pp. 133-136, New York. AAAI.

[9] Brabazon T. and O'Neill M. (2003). Anticipating Bankruptcy Reorganisation from Raw Financial Data using Grammatical Evolution, Proceedings of EvoIASP 2003, *Lecture Notes in Computer Science (2611): Applications of Evolutionary Computing*, edited by Raidl, G., Meyer, J.A., Middendorf, M., Cagnoni, S., Cardalda, J. J. R., Corne, D. W., Gottlieb, J., Guillot, A., Hart, E., Johnson, C. G., Marchiori, E., pp. 368-378, Berlin: Springer-Verlag.

[10] Brabazon T., O'Neill M., Matthews R., and Ryan C. (2002). Grammatical Evolution and Corporate Failure Prediction, In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, Spector et. al. Eds., July 9-13, 2002, pp. 1011-1019, New York: Morgan Kaufmann.

[11] Brabazon T., O'Neill M. (2004). Bond Issuer Credit Rating with Grammatical Evolution. In LNCS 3005 Proceedings of Applications of Evolutionary Computation EvoIASP 2004, pp.270-279, Coimbra, Portugal. Springer.

[12] Cleary R. (2005). Extending Grammatical Evolution with Attribute Grammars: An Application to Knapsack Problems. MSc Thesis, University of Limerick.

[13] Cleary R. O'Neill M. (2005). An Attribute Grammar Decoder for the 01 MultiConstrained Knapsack Problem, in LNCS 3488 *Proceedings of the European Conference on Evolutionary Combinatorial Optimisation - EvoCOP 2005*. Lausanne, Switzerland, pp. 34-45. Springer.

[14] Dempsey I., O'Neill M., Brabazon A. (2002). Investigations into Market Index Trading Models Using Evolutionary Automatic Programming. In LNAI 2464, Proceedings of the 13th Irish Conference on Artificial Intelligence and Cognitive Science AICS 2002, University of Limerick, Ireland, pp.165-170. Springer.

[15] Dempsey I., O'Neill M., Brabazon A. (2004). Grammatical Constant Creation, in LNCS 3103 *Proceedings of the Genetic and Evolutionary Computation Conference - GECCO 2004*, Part 2, pp. 447-458, Seattle WA, USA, Springer-Verlag.

[16] Dempsey I., O'Neill M. and Brabazon A. (2004). Live Trading with Grammatical Evolution, in *Proceedings of the Grammatical Evolution Workshop 2004*, a Workshop of the Genetic and Evolutionary Computation Conference, GECCO 2004.

[17] Dempsey I., O'Neill M., Brabazon A. (2005). meta-grammar Constant Creation, in *Proceedings of the Genetic and Evolutionary Computation Conference - GECCO 2005*, pp. 1665-1672, Washington DC, USA, ACM Press.

[18] Hemberg M. (2001). GENR8 - A Design Tool for Surface Generation. MSc Thesis, Chalmers University of Technology, Gothenburg.

[19] Hemberg M., O'Reilly U-M. (2002). Using Grammatical Evolution in a Surface Design Tool. In Proceedings of the Grammatical Evolution Workshop at GECCO 2002, New York, New York.

[20] Karpuzca U.R. (2005). Automatic Verilog Code Generation Through Grammatical Evolution. In Proceedings of the Undergraduate Workshop at GECCO 2005, Washington, D.C., USA.

[21] Leahy F. (2005). Social Programming: Investigations in Grammatical Swarm. MSc Thesis, University of Limerick.

[22] Lewin B. (2000). Genes VII. Oxford University Press.

[23] Moore J.H., Hahn L.W. (2003). Petri net modelling of high-order genetic systems using Grammatical Evolution. BioSystems 72, pp.177-86.

[24] Moore J.H., Hahn L.W. (2004). Systems Biology Modelling in Human Genetics Using Petri Nets and Grammatical Evolution. In LNCS 3102 Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2004, Seattle, WA, USA, Vol.1, pp.392-401. Springer.

[25] O'Driscoll M., McKenna S., Collins J.J. (2002). Synthesising Edge Detectors with Grammatical Evolution. In Proceedings of the Grammatical Evolution Workshop at GECCO 2002, New York, New York.

[26] O'Neill M. (2001). Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution. PhD thesis, University of Limerick.

[27] O'Neill M., Brabazon A. (2005). mGGA: The meta-Grammar Genetic Algorithm. in LNCS 3447 *Proceedings of the European Conference on Genetic Programming - EuroGP 2005*, pp. 311-320, Lausanne, Switzerland. Springer.

[28] O'Neill M., Brabazon A. (2004). Grammatical Swarm, in LNCS 312 0 *Proceedings of the Genetic and Evolutionary Computation Conference - GECCO 2004*, Part 1, pp. 163-174, Seattle WA, USA. Springer-Verlag.

[29] O'Neill M., Brabazon A., Adley C. (2004). The Automatic Generation of Programs for Classification Problems with Grammatical Swarm, in *Proceedings of the Congress on Evolutionary Computation - CEC 2004*, Vol.1, pp. 104-110, Portland OR, USA. IEEE.

[30] O'Neill M., Cleary R. Nikolov N. (2004). Solving Knapsack Problems with Attribute Grammars, in Poli, R. et al. (eds.) *Grammatical Evolution Workshop 2004, Proceedings of the Workshops, Genetic and Evolutionary Computation Conference GECCO 2004*. Seattle, WA, USA, June 2004.

[31] O'Neill M., Brabazon A., Nicolau M., Mc Garraghy S., Keenan P. (2004). πGrammatical Evolution, in LNCS 3103 *Proceedings of the Genetic and Evolutionary Computation Conference - GECCO 2004*, Part 2, pp. 617-629, Seattle WA, USA. Springer-Verlag.

[32] O'Neill M., Ryan C. (2003) *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*, Kluwer Academic Publishers.

[33] O'Neill M., Ryan C. (2004). Grammatical Evolution by Grammatical Evolution. The Evolution of Grammar and Genetic Code, *LNCS 3003. Proc. of the European Conference on Genetic Programming 2004*, pp. 138-149, Coimbra, Portugal. Springer.

[34] O'Neill M., Ryan C. (2001) Grammatical Evolution, *IEEE Trans. Evolutionary Computation* 5(4):349-358.

[35] O'Neill M., Ryan C. (2002) Proceedings of the First Grammatical Evolution Workshop, In the Workshop Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2002, New York, New York, USA. AAAI.

[36] O'Neill M., Ryan C. (2003) Proceedings of the Second Grammatical Evolution Workshop, In the Workshop Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2003, Chicago, Illinois, USA.

[37] O'Neill M., Ryan C. (2004) Proceedings of the Third Grammatical Evolution Workshop, In the Workshop Proceedings of the Genetic and Evolutionary Computation Conference GECCO 2004, Seattle, Washington, USA.

[38] Ryan C., Collins J.J., O'Neill M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Lecture Notes in Computer Science 1391, Proceedings of the First European Workshop on Genetic Programming*, pp. 83-95, Springer-Verlag.