Grammatical Evolution

Michael O'Neill, *Student Member, IEEE*, and Conor Ryan

***Abstract*—We present grammatical evolution, an evolutionary algorithm that can evolve complete programs in an arbitrary language using a variable-length binary string. The binary genome determines which production rules in a Backus–Naur form grammar definition are used in a genotype-to-phenotype mapping process to a program. We demonstrate how expressions and programs of arbitrary complexity may be evolved and compare its performance to genetic programming.**

***Index Terms*—Automatic programming, Backus–Naur form, degenerate code, evolutionary algorithms, neutral networks.**

I. Introduction

EVOLUTIONARY algorithms have been used with much success for the automatic generation of programs. In particular, genetic programming (GP) has enjoyed considerable popularity and widespread use [14]–[16]. GP originally employed Lisp as its target language. However, many experimenters generate a home-grown language specific to their particular problem.

Unlike GP, grammatical evolution (GE) does not perform the evolutionary process on the actual programs, but rather on variable-length binary strings. A mapping process is employed to generate programs in any language by using the binary strings to select production rules in a Backus–Naur form (BNF) grammar definition. The result is the construction of a syntactically correct program from a binary string which can then be evaluated by a fitness function.

As noted in [1] and [12], a mapping process and its subsequent separation of search and solution spaces can result in benefits such as the unconstrained search of the genotype while still ensuring validity/legality of the program's output. For a discussion on various methods of generating legal phenotypes from a genotype, see [37]. Another potential benefit of such a morphogenic process is that genetic diversity may be enhanced based on the neutral theory of evolution [13], which states that most mutations driving the evolutionary process are neutral with respect to the phenotype; that is, a mutation may have no effect on the phenotypic fitness of an individual. This phenomenon is facilitated in this system by the use of a degenerate genetic code, which is observed in biological genetic systems. The degenerate code facilitates the occurrence of neutral mutations, a consequence of which is that various genotypes can represent the same phenotype, thus facilitating the maintenance of genetic diversity within a population.

Manuscript received August 30, 1999; revised March 9, 2000 and November 13, 2000.

The authors are with the Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland (e-mail: michael.oneill@ul.ie; conor.ryan@ul.ie).

Publisher Item Identifier S 1089-778X(01)04278-3.

This paper serves as an introduction to GE and the system's unique features, namely, the degenerate genetic code and wrapping.

II. Background

GE is not the first instance in which grammars have been used with evolutionary approaches to automatic programming. A number of other attempts using grammars with GP have been made [5], [7], [10], [12], [24], [33], [36] largely to overcome the so-called "closure" problem, the generation and preservation of valid programs. As well as examining the closure problem, Whigham [33] used grammars as a method to introduce bias into the evolutionary process [34], [35]. This is achieved by biasing the individuals in generation zero and by modifying the grammars during runs. As in Wong and Leung [36] and Horner [10], derivation trees are used as the genotype representation by Whigham. The derivation trees state exactly which production rules are to be used at any time during the mapping process onto the phenotype. In [10], a great deal of effort is put into generating complete derivation trees for the initial generation and genetic operators must be designed to maintain closure of the generated programs. Paterson [24] and Freeman [5] attempted to overcome the problem of generating the initial generation by introducing a repair mechanism that used default values in the case that a nonterminal was left without a terminal having been specified. Each uses fixed-length integer arrays as the genotype representation, where each integer represents a production rule from the BNF. When rules cannot be applied in these systems, they are ignored, which can result in a proliferation of introns. Keller and Banzhaf [12] use a repair mechanism of a different sort, whereby illegal terminal symbols in the generated code are replaced with a legal terminal according to an LALR grammar. Each terminal symbol is represented by a unique binary code, an illegal symbol is replaced by the legal symbol whose code is closest by hamming distance. This system also uses fixed-length genomes, in this case with binary coding.

GE presents a unique way of using grammars in the process of automatic programming. Variable-length binary string genomes are used with each codon representing an integer value where codons are consecutive groups of 8 bits. The integer values are used in a mapping function to select an appropriate production rule from the BNF definition, the numbers generated always representing one of the rules that can be used at that time. GE does not suffer from the problem of having to ignore codon integer values because it does not generate illegal values. The issue of ensuring a complete mapping of an individual onto a program comprised exclusively of terminals is partly resolved using a novel technique to evolutionary algorithms (EAs) called *wrapping*. This technique draws inspiration from the *overlapping genes* phenomenon exhibited by many bacteria, viruses,

and mitochondria that enables them to reuse the same genetic material in the expression of different genes [3].

GE, then, is a system that employs a robust new mapping process, the end result of which is the ability to produce code in any language from a simple binary string. At present, the search element of the system is carried out by an EA, although conceivably any search method with the ability to operate over variable-length binary strings could be employed. In particular, future advances in the field of EAs can be easily incorporated into this system due to the program representation.

A. Backus–Naur Form

BNF is a notation for expressing the grammar of a language in the form of production rules [20]. BNF grammars consist of `terminals`, which are items that can appear in the language, e.g., $+$, $-$, etc., and `nonterminals`, which can be expanded into one or more terminals and nonterminals. A grammar can be represented by the tuple $\{N, T, P, S\}$, where N is the set of nonterminals, 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 that is a member of N. When there are a number of productions that can be applied to one particular N, the choice is delimited with the '|' symbol.

Below is an example BNF, where

$$N = \{\text{expr}, \text{op}, \text{pre\_op}\}$$
$$T = \{\text{Sin}, +, -, /, *, \text{X}, 1.0, (,)\}$$
$$S = \langle \text{expr} \rangle$$

and P can be represented as

```
(1)<expr> ::= <expr><op><expr>        (0)
            | (<expr><op><expr>)      (1)
            | <pre-op>(<expr>)        (2)
            | <var>                   (3)
(2)<op> ::= +    (0)
          | -    (1)
          | /    (2)
          | *    (3)
(3)<pre-op> ::= Sin
(4)<var> ::= X     (0)
           | 1.0   (1).
```

Unlike the approach in [14], there is no distinction made at this stage between functions (operators in this sense) and terminals (variables in this example). However, this distinction is more of an implementation detail than a design issue. Whigham [35] also noted the possible confusion with terminology and used the terms GPFunctions and GPTerminals for clarity. We use the term terminals with its usual meaning in grammars.

For the above BNF, Table I summarizes the production rules and the number of choices associated with each.

In GE, the BNF definition is used to describe the output language to be produced by the system, i.e., the compilable code produced will consist of elements of the terminal set T. As the BNF 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.

TABLE I
NUMBER OF CHOICES AVAILABLE FROM EACH PRODUCTION RULE

Rule no.	Choices
1	4
2	4
3	1
4	2

III. BIOLOGICAL APPROACH

The GE system is inspired largely by the biological process of generating a protein from the genetic material of an organizm. Proteins are fundamental in the proper development and operation of living organizms and are responsible for traits such as eye color and height [3].

The genetic material [usually deoxyribonucleic acid (DNA)] contains the information required to produce specific proteins at different points along the molecule. For simplicity, consider DNA to be a string of four building blocks called nucleotides 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 the final three-dimensional structure of the protein, which in turn has a role to play in determining its 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, that being a sequence of elements on a molecule known as ribonucleic acid (RNA). Codons within the RNA molecule are then translated to determine the sequence of amino acids that are contained within the protein molecule.

The result of the expression of the genetic material as proteins in conjunction with environmental factors is the phenotype. In GE, the phenotype is a computer program that is generated from the genetic material (the genotype) by a process termed a genotype-phenotype mapping. This is unlike the standard method of generating a solution (a program in the case of GE) directly from an individual in an EA by explicitly encoding the solution within the genetic material. Instead, a many-to-one mapping process is employed within which the robustness of the GE system lies.

Fig. 1 compares the mapping process employed in both GE and biological organizms.

IV. GRAMMATICAL EVOLUTION

When tackling a problem with GE, a suitable BNF definition must first be decided upon. The BNF can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared toward the problem at hand. Complete BNFs are freely available for languages such as C and these can easily be plugged in to GE.

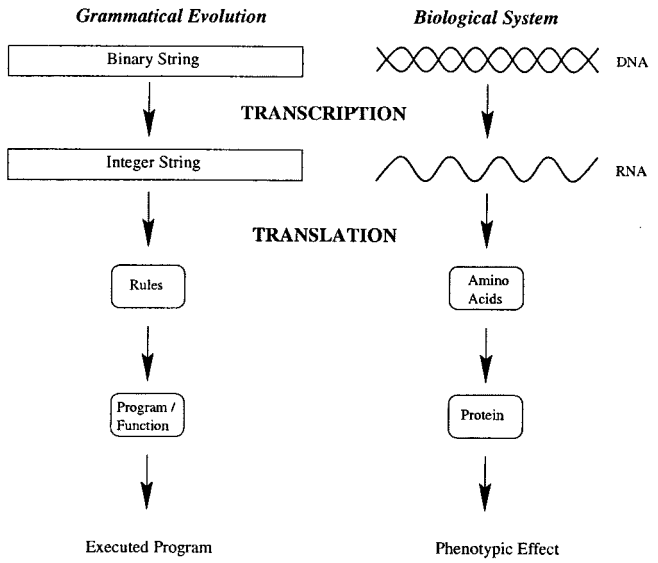


Fig. 1. Comparison between the GE system and a biological genetic system. Binary string of GE being equivalent to the double helix of DNA, each guiding the formation of the phenotype. In the case of GE, this occurs via the application of production rules to generate the terminals of the compilable program and in the biological case, directing the formation of the phenotypic protein by determining the order and type of protein subcomponents (amino acids) that are joined together.

A. Mapping Process

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:

$$\text{rule} = (\text{codon integer value}) \; MOD \; (\text{number of rules for the current nonterminal}).$$

Considering the following rule, i.e., given the nonterminal *op*, there are four production rules to select from:

```
(2)<op> ::= +   (0)
          | -   (1)
          | /   (2)
          | *   (3).
```

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

$$6 \; MOD \; 4 = 2 \tag{1}$$

would select rule (2) /. Each time a production rule has to be selected to map from a nonterminal, 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, we wrap the individual and reuse the codons. This is quite an unusual approach in EAs 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, which has been observed in many organizms [3].

In GE, each time the same codon is expressed, it will always generate the same integer value, but depending on the current nonterminal to which it is being applied, it may result in the selection of a different production rule. 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 results because the same choices are made each time. It is possible that an incomplete mapping could occur even after several wrapping events and in this case, 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 over and over. For example, consider an individual with three codons, all of which specify rule 0 from below

```
(1)<expr> ::= <expr><op><expr>      (0)
            | (<expr><op><expr>)    (1)
            | <pre-op>(<expr>)      (2)
            | <var>                 (3).
```

Even after wrapping, the mapping process would be incomplete and would carry on indefinitely unless stopped. This occurs because the nonterminal `<expr>` is being mapped indefinitely by production rule 0, i.e., it becomes `<expr><op><expr>`.Therefore, the leftmost `<expr>` after each application of a production would itself be mapped to a `<expr><op><expr>`, resulting in an expression continually growing as follows: `<expr><op><expr><op><expr><op><expr>`, etc. Such an individual is dubbed "invalid" as it will never undergo a complete mapping to a set of terminals.

To reduce the number of invalid individuals being passed from generation to generation, a steady-state replacement mechanism is 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.

B. Example Individual

Consider the individual in Fig. 2. These numbers will be used to look up Table I, which describes the BNF grammar given in Section II-A.

Concentrating on the start symbol `<expr>`, we can see that there are four productions to choose from. To make this choice, we read the first codon from the chromosome and use it to generate a number. This number will then be used to decide which production rule to use according to (1) in Section IV-A. Thus, we have $220 \; MOD \; 4 = 0$, meaning we must take the zeroth production so that `<expr>` is now replaced with

`<expr><op><expr>.`

Notice that if this individual is subsequently wrapped, each codon will always result in the same integer value, but depending on the number of choices for the current nonterminal, a different rule number could be selected. In this way, although

220	240	220	203	101	53	202	203	102	55	220	202	241	130	37	202	203	140	39	202	203	102

Fig. 2. Example individual expressed as integers. Integer values are generated by converting the 8-bit binary number that is each codon into its corresponding integer value.

we have the same codon integer value, it could result in a different physical trait.

Continuing with the first <expr>, i.e., always starting from the leftmost nonterminal, a similar choice must be made by reading the next codon value (240) and again using the given formula we get $240\ MOD\ 4 = 0$, i.e., rule (0). The leftmost <expr> will now be replaced with <expr><op><expr> to give

<expr><op><expr><op><expr>.

Again, we have the same choice for the first <expr> by reading the next codon value 220, the result being the application of rule (0) to give

<expr><op><expr><op><expr><op><expr><op><expr>.

Now, the leftmost <expr> will be determined by the codon value 203 that gives us rule (3), which is <expr> becomes <var>. The next codon then determines what value <var>, which has two possible production rules, shall take. This is $101\ MOD\ 2 = 1$, i.e., rule (1), which turns out to be 1.0. We now have the following:

1.0 <op><expr><op><expr><op><expr><op><expr>.

The next codon will determine what <op> will become, so we have $53\ MOD\ 4 = 1$, which gives a -. The next <expr> has then to be expanded using the codon value 202, that is $202\ MOD\ 4 = 2$. So, we now have

1.0−<pre-op><expr><op><expr><op><expr><op><expr>.

There can only be one outcome for a <pre-op>, that being Sin. Therefore, no decision has to be made and so no codon is read. The next <expr> is then expanded by the value $203\ MOD\ 4 = 3$, which is rule (3) or <var>. Its value is then determined by $102\ MOD\ 2 = 0$, rule (0), and the resulting expression is

1.0 − Sin(x)<op><expr><op><expr><op><expr>.

The mapping continues until eventually we are left with the following expression:

1.0 − Sin(x) ∗ Sin(x) − Sin(x) ∗ Sin(x).

Notice how all of the codons were required in this case; had there been any extra codons, they would have been simply ignored.

C. Genetic Code Degeneracy

Given an 8-bit binary number, each codon in GE can represent 256 distinct integer values, although many of these integer values can represent the same production rule. Taking production rule 2 in Section II-A as an example, if the current codon value were 8, then $8\ MOD\ 4 = 0$ would select rule (0) +. The same rule would be chosen if the codon value were 4, 12, 16, etc.

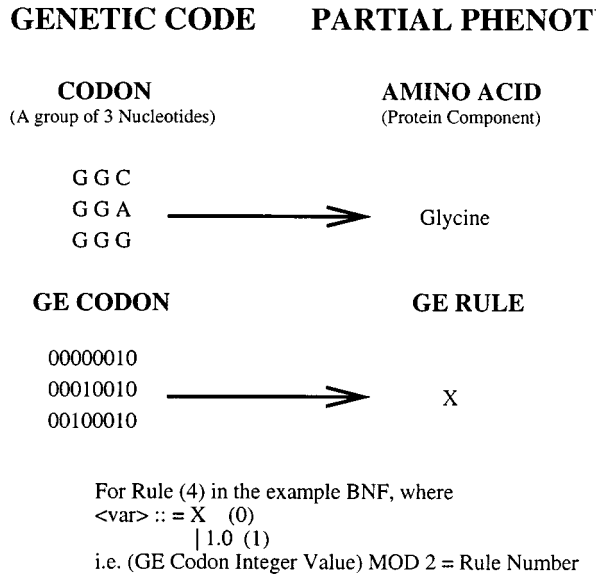


Fig. 3. Genetic code degeneracy.

A similar phenomenon can be observed in the genetic code of biological organizms, referred to as *degenerate genetic code* [3]. There are 4^3, i.e., 64, unique combinations of nucleotides in a codon, 61 of these coding for a specific amino acid, the other three being special codons that delimit the start and end of genes on the DNA. On average, there are three codons for every amino acid, i.e., more than one codon can represent the same amino acid, and it has been observed that the first two nucleotides in the codon are often sufficient to specify a particular amino acid. The value of the nucleotide at the third position is often irrelevant. Code degeneracy has interesting implications when it comes to mutation effects. A mutation at the third codon position can often produce what is called a *neutral mutation*, meaning that the amino acid specified will be the same as the one before the mutation event due to the flexibility at the third codon position. With respect to GE, this means that subtle changes in the search space (genotype) may have no effect on the solution space (phenotype), which could result in the maintenance of genotypic diversity throughout a run of the system as different genotypes can represent the same phenotype. It may also preserve valid individuals because the neutral mutations provide a buffering effect against destructive mutation events. Evidence to this effect has been presented in [23]. More recently, advantages of neutrality in the evolution of digital circuits and another GP variant has been discussed [19], [32]. Fig. 3 shows that in the genetic code of biological organizms the nucleotide at position three of the codon is independent of the amino acid produced (valine).

Similarly with GE, it can be seen in the given example that a single bit mutation has no effect on the rule used in this case, i.e., $2\ MOD\ 2 = 18\ MOD\ 2 = 34\ MOD\ 2 = 66\ MOD\ 2 = 0$, the zeroth rule *line*. Note, however, if the number of choices in the example was uneven, e.g., three, a single bit mutation would change the rule used.

Kimura's neutral theory of evolution [13] suggests that it is these neutral mutations that are responsible for the genetic diversity that has been observed in natural populations, a phenomenon that has been exhibited within GE.

D. Evolutionary Algorithm

As the population being evolved comprises simple binary strings, we do not have to employ any special crossover or mutation operators and an unconstrained search is performed on these strings due to the genotype-to-phenotype mapping process that will generate syntactically correct individuals.

The EA adopted in this case is a variable-length genetic algorithm. Individual initialization is achieved by randomly generating variable-length binary strings within a prespecified range of codons. For all experiments conducted in this paper, we use the initialization range of ten codons, where a codon is a group of 8 bits.

As well as the standard genetic operators of mutation (point) and crossover (one-point), we adopt a codon duplication operator. Duplication involves randomly selecting a number of codons to duplicate and the starting position of the first codon in this set. The duplicated codons are placed at the end of the chromosome.

The GE component of the system, i.e., the part that carries out the mapping from binary string to the output code, could conceivably be plugged in to the fitness function of any EA. The result of this is that GE can benefit from the latest advances in EA research. For example, we are currently investigating the use of "competent GAs," which have been shown to have superior scaling properties to the simple GA [6], [8], [9], [11], [25], [31].

V. Automatic Generation of Expressions and Functions

We now describe how GE was applied to three problem domains, namely, symbolic regression, the Santa Fe ant trail, and symbolic integration, to illustrate that GE can produce compilable code. These problems are deliberately diverse. In the symbolic regression problem, a simple one-line expression is evolved, whereas in the Santa Fe trail problem, a multiline function including branch statements is required. A brief description of each problem domain used now follows.

A. Symbolic Regression

Symbolic regression problems involve finding some mathematical expression in symbolic form that represents a given set of input and output pairs. The aim is to determine the function that maps the input pairs onto the output pairs. The particular function examined is

$$f(X) = X^4 + X^3 + X^2 + X$$

with the input values in the range $[-1 \cdots 1]$.

The grammar used in this problem is given below

$$\begin{aligned} N &= \{\text{expr}, \text{op}, \text{pre\_op}\} \\ T &= \{\text{Sin}, \text{Cos}, \text{Exp}, \text{Log}, +, -, /, *, \text{X}, 1.0, (,)\} \\ S &= \langle \text{expr} \rangle \end{aligned}$$

and P can be represented as

```
(1)<expr> ::= <expr><op><expr>      (0)
            | (<expr><op><expr>)    (1)
            | <pre-op>(<expr>)      (2)
            | <var>                 (3)
(2)<op> ::= +    (0)
          | -    (1)
          | /    (2)
          | *    (3)
(3)<pre-op> ::= Sin   (0)
              | Cos   (1)
              | Exp   (2)
              | Log   (3)
(4)<var> ::= X     (0)
           | 1.0   (1).
```

The production rules for <expr> are similar to those given earlier in Section II-A, with the terminal operator set also including Cos, Exp, and Log.

We adopt a style similar to [14] of summarizing information using a modified version of a tableau (see Table II). Notice how our terminal operands and terminal operators are analogous to GPTerminals and GPfunctions, respectively.

The fitness for this problem is given by the sum, taken over 20 fitness cases, of the error between the evolved and target functions.

B. Santa Fe Ant Trail

The Santa Fe ant trail is a standard problem in the area of GP and can be considered a deceptive planning problem with many local and global optima [18]. The objective is to find a computer program to control an artificial ant so that it can find all 89 pieces of food located on a discontinuous trail within a specified number of time steps, the trail being located on a 32 by 32 toroidal grid. The ant can only turn left, right, move one square forward, and may also look ahead one square in the direction it is facing to determine if that square contains a piece of food. All actions, with the exception of looking ahead for food, take one time step to execute. The ant starts in the top left-hand corner of the grid facing the first piece of food on the trail. The grammar used in this problem is different to the one given earlier, as with this problem we wish to produce a multiline function. The grammar used is given below

$$\begin{aligned} N &= \{\text{code}, \text{line}, \text{expr}, \text{if} - \text{statement}, \text{op}\} \\ T &= \{\text{left}(), \text{right}(), \text{move}(), \text{food\_ahead}(), \\ &\qquad \text{else}, \text{if}, \{, \}, (,), ;\} \\ S &= \langle \text{code} \rangle \end{aligned}$$

TABLE II
SYMBOLIC REGRESSION TABLEAU

Objective :	Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target function is the quartic polynomial $X^4 + X^3 + X^2 + X$
Terminal Operands:	X (the independent variable), 1.0
Terminal Operators	The binary operators $+, *, /$, and $-$ The unary operators Sin, Cos, Exp and Log
Fitness cases	The given sample of 20 data points in the interval $[-1, +1]$ i.e. { -1, -.9, -.8, -.76, -.72, -.68, -.64, -.4, -.2, 0, .2, .4, .63, .72, .81, .90, .93, .96, .99, 1 }
Raw Fitness	The sum, taken over the 20 fitness cases, of the error
Standardised Fitness	Same as raw fitness
Wrapper	Standard productions to generate C functions
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01

TABLE III
GRAMMATICAL EVOLUTION TABLEAU FOR THE SANTA FE TRAIL

Objective :	Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail.
Terminal Operators:	left(), right(), move(), food_ahead()
Terminal Operands:	None
Fitness cases	One fitness case
Raw Fitness	Number of pieces of food before the ant times out with 600 operations.
Standardised Fitness	Total number of pieces of food less the raw fitness.
Wrapper	None
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01

and P can be represented as

```
(1)  <code> ::= <line>          (0)
              | <code><line>    (1)
(2)  <line> ::= <expr>
(3)  <expr> ::= <if-statement>  (0)
              | <op>            (1)
(4)  <if-statement> ::= if (food_ahead())
                        {<expr>}
                        else
                        {<expr>}
(5)  <op> ::= left ();   (0)
            |right ();   (1)
            |move ();    (2).
```

Note that it is the rules for the nonterminal <code> that are responsible for the production of multiline code. A tableau describing this problem can be seen in Table III.

C. Symbolic Integration

Symbolic integration involves finding a function that is the integral of the given curve. Similarly to symbolic regression, the system is given a set of input and output pairs and must determine the function that maps one onto the other. The particular function examined was

$$f(X) = \mathrm{Cos}(X) + 2X + 1$$

with the input values in the range $[0 \ldots 2\pi]$ and the target integral curve being

$$f(X) = \mathrm{Sin}(X) + X + X^2.$$

We reduce the problem to symbolic regression by integrating the function examined and performing symbolic regression on the target integral curve. The fitness for this problem is given by the sum, taken over 20 fitness cases, of the absolute value of the difference between the individual genetically produced function $f_j(x_i)$ at the domain point x_i and the value of the numerical integral $\mathbf{I}(x_i)$.

The grammar used for this problem is the same as for the symbolic regression problem given in Section V-A and a tableau is given in Table IV.

TABLE IV
SYMBOLIC INTEGRATION TABLEAU

Objective :	Find a function, in symbolic form, that is the integral of a curve presented either as a mathematical expression or as a given finite sample of points (x_i, y_i)
Terminal Operands:	X (the independent variable)
Terminal Operators	The binary operators $+, *, /, -$ and the unary operators Sin, Cos, Exp and Log
Fitness cases	A sample of 20 data points in the interval $[0, 2\pi]$
Raw Fitness	The sum, taken over the 20 fitness cases, of the absolute value of the difference between the individual genetically produced function $f_j(x_i)$ at the domain point x_i and the value of the numerical integral $\mathbf{I}(x_i)$
Standardised Fitness	Same as raw fitness
Wrapper	Standard productions to generate C functions
Parameters	Population Size = 500, Termination when Generations = 51 Prob. Mutation = 0.01, Prob. Crossover = 0.9 Prob. Duplication = 0.01

D. Results

GE was successful in finding correct solutions to all of the problems described here, some results of which have been reported in [21]. A brief overview of these results is given below.

1) Symbolic Integration: GE successfully found the target integral function $\mathrm{Sin}(X) + X + X^2$, illustrating that useful expressions could be generated by the system. A cumulative frequency measure of success over 100 runs can be seen in Fig. 4. The same problem was tackled using GP and a cumulative frequency of success for 100 runs can be seen in Fig. 4. As can be seen, GE outperforms GP on this problem from around the tenth generation.

2) Santa Fe Ant Trail: GE was successful at finding a solution to this case of the Santa Fe trail, demonstrating that GE can generate multiline code by using a simple modification to the grammar definition. An example solution produced by GE is

```
move();
left();
if(food_ahead())
  left();
else
  right();
right();
if(food_ahead())
  move();
else
  left();
```

This solution is executed in a loop until the number of time steps allowed is reached. A cumulative frequency measure of success over 100 runs of GE can be seen in Fig. 5 along with a cumulative frequency measure of success for GP. The left side of this figure shows the performance of the GP system, which incorporated solution length in the fitness measure as well as the number of pieces of food picked up. The performance of the two systems is comparable in this case with GP slightly outperforming GE over the first 30 generations.

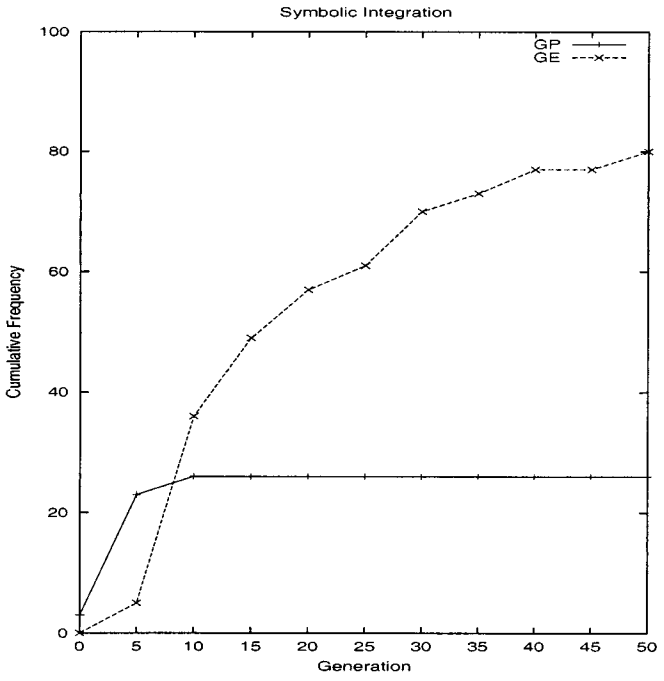
In order to see how the GP system would perform without the solution length as a measure of fitness, we ran 100 more runs of the GP system removing the solution length measure. We felt that using solution length as a measure of fitness required a prior knowledge of the solution's length and, therefore, gave an unfair advantage, as GE did not use such a measure. The right-hand side of Fig. 5 shows a comparison of the cumulative frequency measure of success for these results with the results produced by GE. As can be seen from the figure, the performance of the GP system was compromised and as a result GE outperformed GP.

3) Symbolic Regression: GE successfully found the $X + X^2 + X^3 + X^4$ target function. A cumulative frequency measure of success over 100 runs can be seen in Fig. 4 along with a similar measure for GP and in this case, GP outperforms GE. When comparing GE to GP, it is important to note how the initial generation is formed in each system. In the case of the GP system, a ramped half-and-half generation mechanism is used, creating a range of individuals of varying depths, whereas with GE the generation is totally random. Also, every individual in the initial population is engineered to be unique in the GP system, whereas with GE this is not the case. In the case of this problem, we feel that the generation strategy for the initial generation is providing GP with an advantage due to the regular nature of the solution. This is supported by the fact that on the symbolic integration problem, the solution does not have the same regularity as is the case here. It can be seen from the results that over each of the problem domains compared, GE is more consistent at finding solutions throughout a run and this, we feel, provides GE with an advantage in terms of generalizing to the different problem domains.

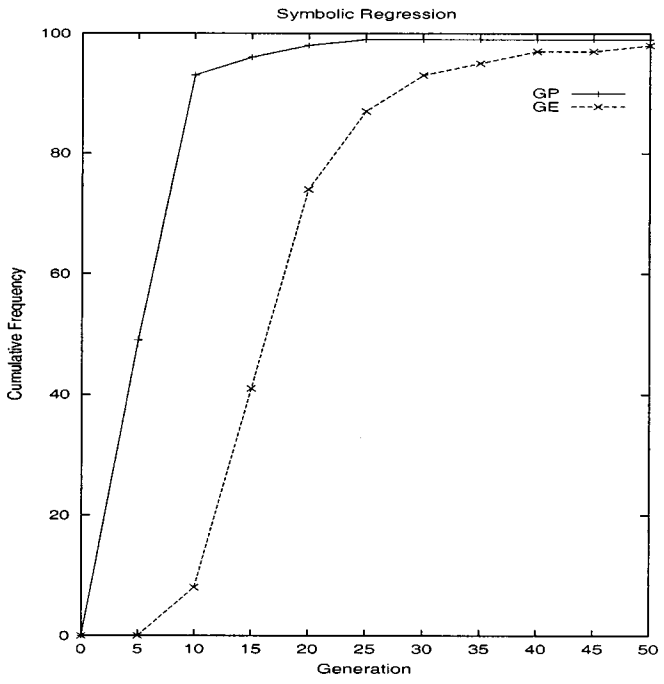
VI. CONCLUSION

GE is a system that can produce code in any language with arbitrary complexity. The only inputs are a BNF definition for the genotype-to-phenotype mapping process and a fitness function.

GE has been shown to be successful across a range of problem domains, including symbolic integration, symbolic regression, and the Santa Fe trail, and of being capable of producing consis-

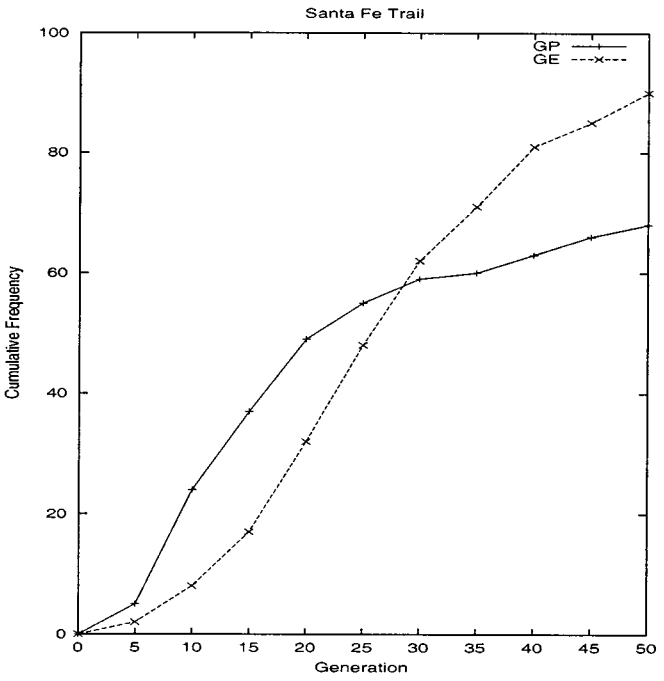


(a)

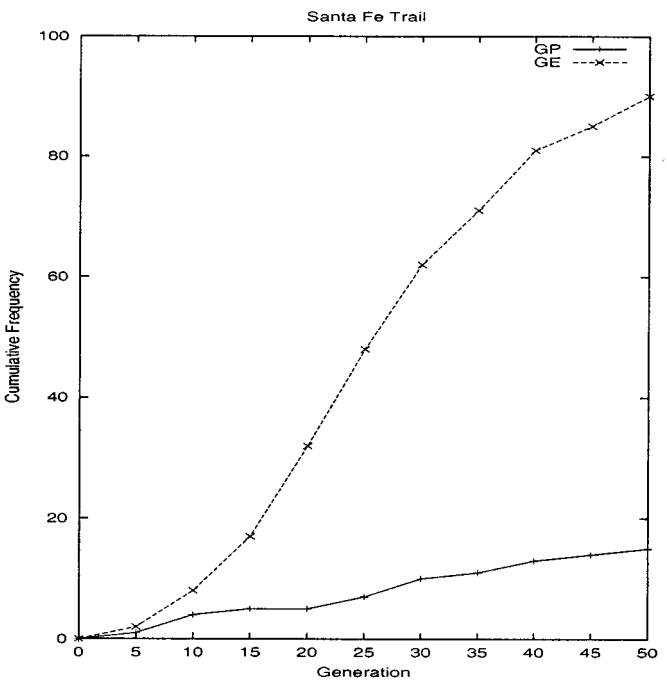


(b)

Fig. 4. Cumulative frequency of success measures of GE versus GP on the (a) symbolic integration and (b) symbolic regression problems.



(a)



(b)

Fig. 5. Cumulative frequency of success measures of GE versus GP on the Santa Fe trail problem. (a) Result when GP uses a solution length constraint in the fitness function (GE has no such measure). (b) Result when the solution length constraint is removed from GP.

tent results across these domains. Results reported here have shown that GE outperforms GP on two of the three problem domains compared.

The problems tackled in this paper were used as proof of concept, the concept being the evolution of compilable computer programs using GE. In the event of tackling more complex problem domains, the choice of grammars may not be as obvious. Typically, one could strive to use the grammar as a means of incorporating domain knowledge into the system by using bias in the form the output programs could take or in the terminals that will be used. In many cases, one would probably know what constructs might be useful for a particular problem. In the event that the user had no idea of the format a solution may take, one could easily use a larger subset of a languages BNF or even the BNF for the entire language. Future work will include an investigation into the time to evolve solutions given such rich terminal sets.

Overall, GE is a biologically plausible EA that has been shown to be a robust approach to the automatic generation of computer programs. GE benefits from a unique mapping process that separates the search and solution spaces, allowing an unconstrained evolutionary search to be performed on simple variable-length binary strings, as well the use of a degenerate genetic code. This separation permits one to use the GE mapping process in conjunction with any search algorithm that operates on binary strings. Specifically, this allows one to take advantage of the advances being made with genetic algorithms, for example, the advent of competent genetic algorithms that have demonstrated superior scaling properties to the simple GA.

The source code for GE is freely available.[1]

ACKNOWLEDGMENT

The authors would like to thank D. Fogel and the anonymous referees for their helpful comments.

REFERENCES

[1] W. Banzhaf, "Genotype-Phenotype Mapping and Neutral Variation—A case study in Genetic Programming," in *Parallel Problem Solving from Nature—PPSN III*, Y. Davidor, H.-P. Schwefel, and R. Männer, Eds. Berlin, Germany: Springer-Verlag, 1994, vol. 866, Lecture Notes in Computer Science, pp. 322–332.

[2] K. A. De Jong and S. Jayshree, "Generation Gaps Revisited," in *Foundations of Genetic Algorithms 2*, D. Whitley, Ed. San Mateo, CA: Morgan Kaufmann, 1992, pp. 19–28.

[3] G. D. Elseth and K. D. Baumgardner, *Principles of Modern Genetics*. St. Paul, MN: West, 1995.

[4] A. Fraser and T. Weinbrenner, *The Genetic Programming Kernel Version 0.5.2*, 1992.

[5] J. J. Freeman, "A Linear Representation for GP using Context Free Grammars," in *Genetic Programming 1998: Proceedings of the 3rd Annual Conference* , J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds. Cambridge, MA: MIT Press, 1998, pp. 72–77.

[6] D. E. Goldberg, B. Korb, and K. Deb, "Messy genetic algorithms: motivation, analysis, and first results," *Complex Syst.*, vol. 3, no. 5, pp. 493–530, Sept. 1989.

[7] F. Gruau, "Neural Network synthesis using cellular encoding and the genetic algorithm," Ph.D. dissertation, Centre d'etude nucleaire de Grenoble, Grenoble, France, 1994.

[8] G. R. Harik and D. E. Goldberg, "Learning Linkage," in *Foundations of Genetic Algorithms 4*, R. Belew and M. D. Vose, Eds. San Mateo, CA: Morgan Kaufmann, 1997, pp. 247–262.

[9] G. R. Harik, "Linkage learning via probabilistic modeling in the ECGA," Univ. Illinois, Urbana-Champaign, Urbana, IL, Tech. Rep. 99010, 1999.

[10] H. Horner, "A C++ Class Library for Genetic Programming: The Vienna University of Economics Genetic Programming Kernel Release 1.0, Operating Instruction," Vienna University of Economics, Vienna, Austria, 1996.

[11] H. Kargupta, "Revisiting the GEMGA: Scalable evolutionary optimization through linkage learning," in *Proceedings of the IEEE International Conference on Evolutionary Computation*. Piscataway, NJ: IEEE Press, 1998, pp. 603–608.

[12] R. Keller and W. Banzhaf, "GP using mutation, reproduction and genotype-phenotype mapping from linear binary genomes into linear LALR phenotypes," in *Genetic Programming 1996: Proceedings of the 1st Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Cambridge, MA: MIT Press, 1996, pp. 116–122.

[13] M. Kimura, *The Neutral Theory of Molecular Evolution*. Cambridge, U.K.: Cambridge Univ. Press, 1983.

[14] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[15] ——, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press, 1994.

[16] ——, *Genetic Programming III: Darwinian Invention and Problem Solving*. San Mateo, CA: Morgan Kaufmann, 1999.

[17] W. Langdon, *Genetic Programming and Data Structures*. Norwell, MA: Kluwer, 1998.

[18] W. Langdon and R. Poli, "Why Ants Are Hard," in *Genetic Programming 1998: Proceedings of the 3rd Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds. Cambridge, MA: MIT Press, 1998, pp. 193–201.

[19] J. F. Miller and P. Thomson, "Cartesian Genetic Programming," in *EuroGP 2000: Proceedings of the Third European Conference on Genetic Programming*, R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, Eds. Berlin, Germany: Springer-Verlag, 2000, pp. 121–132.

[20] P. Naur, "Revised report on the algorithmic language ALGOL 60," *Commun. ACM*, vol. 6, no. 1, pp. 1–17, Jan. 1963.

[21] M. O'Neill and C. Ryan, "Evolving Multi-line Compilable C Programs," in *Proceedings of the Second European Workshop on Genetic 1999*. Berlin, Germany: Springer-Verlag, 1999, vol. 1598, Lecture Notes in Computer Science, pp. 83–92.

[22] ——, "Under the Hood of Grammatical Evolution," in *GECCO '99: Proceedings of the Genetic and Evolutionary Computation Conference 1999*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds. San Mateo, CA: Morgan Kaufmann, 1999, vol. 2, pp. 1143–1148.

[23] ——, "Genetic code degeneracy: implications for grammatical evolution and beyond," in *ECAL'99: Proceedings of the Fifth European Conference on Artificial Life*, Lausanne, Switzerland, Sept. 1999, pp. 149–153.

[24] N. Paterson and M. Livesey, "Evolving caching algorithms in C by GP," in *Genetic Programming 1997: Proceedings of the 2nd Annual Conference*. Cambridge, MA: MIT Press, 1997, pp. 262–267.

[25] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian optimization algorithm," in *GECCO '99: Proceedings of the Genetic and Evolutionary Computation Conference 1999*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds. San Mateo, CA: Morgan Kaufmann, 1999, vol. 1, pp. 525–532.

[26] C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language," in *EuroGP'98: Proceedings of the First European Workshop on Genetic Programming*. Berlin, Germany: Springer-Verlag, 1998, vol. 1391, Lecture Notes in Computer Science, pp. 83–95.

[27] C. Ryan, M. O'Neill, and J. J. Collins, "Grammatical Evolution: Solving Trigonometric Identities," in *Mendel'98: Proceedings of the 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks, and Rough Sets*. Brno, Czech Republic: Tech. Univ. Brno, 1998, pp. 111–119.

[28] C. Ryan and M. O'Neill, "Grammatical Evolution: A Steady State Approach," in *Genetic Programming 1998: Proceedings of the 3rd Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds. Cambridge, MA, 1998, pp. 180–185.

[29] ——, "Grammatical Evolution: A Steady State Approach," in *Proc. Joint Conf. Information Sciences*, Research Triangle Park, NC, 1998, pp. 419–423.

[30] M. Schütz, "Gene Duplication and Deletion," in *The Handbook of Evolutionary Computation*, T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds. Bristol, U.K.: Inst. of Physics, 1997, sec. C3.4.3.

[31] D. Thierens, "Scalability Problems of Simple Genetic Algorithms," *Evol. Comput.*, vol. 7, no. 4, pp. 331–352, 1999.

[32] V. K. Vassilev and J. F. Miller, "The Advantages of Landscape Neutrality in Digital Circuit Evolution," in *ICES 2000: Proceedings of the Third International Conference on Evolvable Systems*, J. Miller, A. Thompson, P. Thomson, and T. C. Fogarty, Eds. Berlin, Germany, 2000, pp. 252–263.

[33] P. Whigham, "Grammatically-based Genetic Programming," in *Proceedings of the Workshop on GP: From Theory to Real-World Applications*. San Mateo, CA: Morgan Kaufmann, 1995, pp. 33–41.

[34] ——, "Inductive bias and genetic programming," in *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*. Stevenage, U.K.: IEE, 1995, pp. 461–466.

[35] ——, "Search Bias, Language Bias and Genetic Programming," in *Genetic Programming 1996: Proceedings of the 1st Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Cambridge, MA: MIT Press, 1996, pp. 230–237.

[36] M. Wong and K. Leung, "Applying logic grammars to induce subfunctions in genetic programming," in *Proceedings of the IEEE Conference on Evolutionary Computation*. Piscataway, NJ: IEEE Press, 1995, pp. 737–740.

[37] T. Yu and P. Bentley, "Methods to evolve legal phenotypes," in *Parallel Problem Solving from Nature—PPSN V*, A. E. Eiben, M. Schoenauer, and T. Bäck, Eds. Berlin, Germany: Springer-Verlag, 1998, vol. 1498, Lecture Notes in Computer Science, pp. 280–291.

[1] www.grammatical-evolution.org

Michael O'Neill (S'98) received the B.Sc. degree in biochemistry and the Graduate Diploma in computer science from the University College Dublin, Dublin, Ireland, in 1996 and 1997, respectively. He is working toward the Ph.D. degree in grammatical evolution at the University of Limerick, Limerick, Ireland.

He is currently a Lecturer with the Department of Computer Science and Information Systems, University of Limerick. His current research interests include automatic programming, evolutionary computation, artificial life, and robotics.

Conor Ryan received the B.A. degree in computer science and economics and the Ph.D. degree in computer science from the University College Cork, Cork, Ireland.

He is currently a Lecturer at the University of Limerick, Limerick, Ireland, where he is also the director of the Soft Computing and Re-Engineerring Group. His current research interests include genetic programming, evolutionary algorithms, automatic programming, and auto-parallelization and is particularly interested in the use of modern heuristic techniques in industrial scale problems.