Introducing Grammar Based Extensions for Grammatical Evolution

Miguel Nicolau and Ian Dempsey

***Abstract*— This paper presents a series of extensions to standard Grammatical Evolution. These grammar-based extensions facilitate the exchange of knowledge between genotype and phenotype strings, thus establishing a better correlation between the search and solution spaces, typically separated in Grammatical Evolution. The results obtained illustrate the practical advantages of these extensions, both in terms of convenience and potential increase in performance.**

I. Introduction

The main principle behind Grammatical Evolution (GE) [1] is simple but effective: using a standard genetic algorithm [2], a population of binary strings is evolved, and a mapping process is then applied, which, through the use of a grammar, transforms those strings into functional programs adapted to the problem domain. This complete independence between the search algorithm and the mapping process presents some advantages, such as the unconstrained exploration of the search space, and even the possibility of using different search engines with GE [3], [4].

In this paper, the division line between the search engine and the mapping process is slightly blurred. The main idea is to allow the mapping process to use extra information from the underlying genetic algorithm, and even pass information back to the genetic algorithm. This is achieved through the introduction of special symbols in the grammars used by GE.

The first of these symbols allows the direct phenotypic use of genotypic values; the use of this symbol in the grammar, during the mapping process, inserts a gene value into the phenotype program. This allows for the design of more compact grammars, without any detrimental effects on the performance of GE.

The second symbol introduced uses information from the mapping process to setup a list of crossover locations in the genotype string; this allows the use of domain-knowledge to setup blocks of information to exchange between individuals, and even the evolution of the boundaries of these blocks.

These special symbols are tested in a series of different problems, ranging from constant discovery to a form of symbolic regression, chosen specifically to test the validity of these new approaches. The results obtained illustrate the advantages introduced by their usage, both in terms of simplicity and performance.

Miguel Nicolau is with the Biocomputing and Developmental Systems Group, Department of Computer Science and Information Systems, University of Limerick, Ireland (email: Miguel.Nicolau@ul.ie).

Ian Dempsey is with the Natural Computing Research & Applications Group, University College Dublin, Belfield, Dublin 4, Ireland (email: ian.dempsey@gmail.com).

This paper is structured as follows. Section II starts by presenting GE and its mapping process, while Sections III and IV present the extensions used in this work. Section V introduces the problem domains tackled, and Section VI analyses the results obtained. Finally, Section VII concludes this work and draws future work directions.

II. Grammatical Evolution

Grammatical Evolution [5], [6], [1] is an automatic programming system, similar to Genetic Programming (GP) [7], in that it uses an evolutionary process to automatically generate computer programs. Unlike GP, GE uses a population of linear genotypic binary strings, which are transformed into functional phenotypic programs, through a genotype-to-phenotype mapping process [8]. This transformation is governed through the use of a BNF (Backus-Naur Form) grammar, which specifies the language of the produced solutions.

The use of a mapping process creates a distinction between the search and solution spaces; genotype strings are evolved with no knowledge (or respect) of their phenotypic equivalent, apart from a fitness measure. This has been shown to create a "ripple effect" when standard genetic operators are applied: as the function of a gene depends on the genes that preceed it, a small genotypic change can lead to a big phenotypic change. However, evidence suggests that this effect can promote a useful exchange of derivation subsequences during crossover events [9].

A. Mapping Process

An example of the mapping process employed by GE is shown in Fig. 1. Starting with a binary string, an integer string is evolved, typically by using 8 bits per *codon*; these are then used to choose rules from a given BNF grammar, to generate a phenotypic program.

Starting with a given *start symbol*, a production associated with that symbol is chosen to replace it, by using the current codon from the integer string. In the example, the codon 6 is mapped to the number of available productions associated with `<E>`, which are 3, so $6 \ mod \ 3 = 0$, and `<E>` is replaced with the sequence `(<OP> <E> <E>)`.

The next step consists in choosing a production for the `<OP>` non-terminal symbol, which is now the leftmost symbol on the phenotype string under construction. This is done by using the next codon, 7, and mapping it to the number of productions associated with `<OP>`, giving $7 \ mod \ 4 = 3$, so `<OP>` is replaced with `/`.

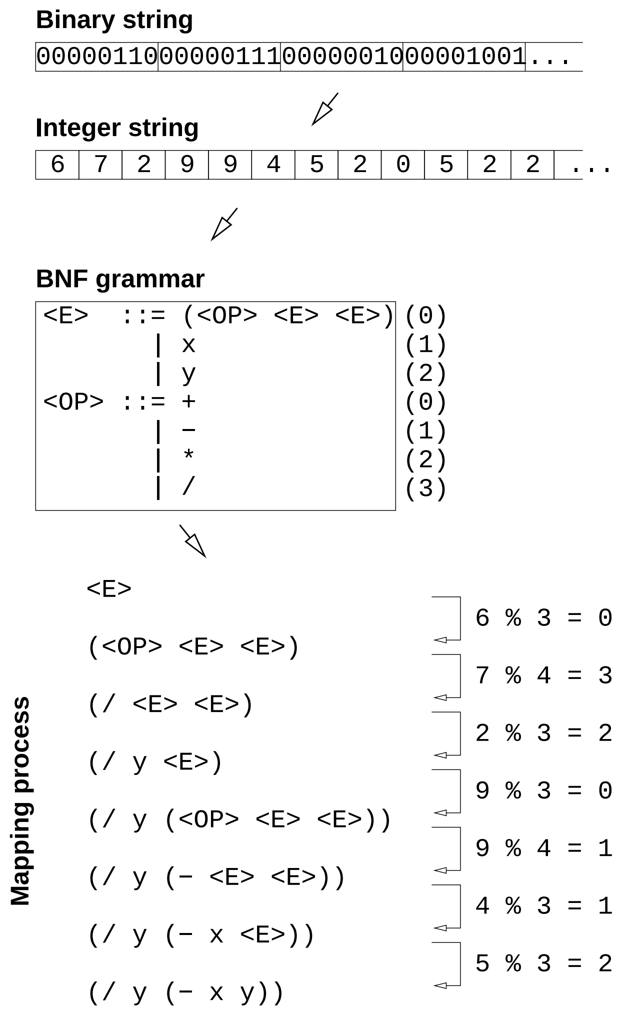

Fig. 1. The GE mapping process. A binary string is converted into an integer string, using 8 bits per codon; these integers are then used to choose productions from a BNF grammar, which map a given start symbol into a sequence of terminal symbols.

The mapping process continues in this manner, always replacing the leftmost non-terminal symbol by a production associated with it on the grammar, chosen by a codon; the mapping terminates and is successful when a sequence of terminal symbols is generated. If there are non-terminal symbols left in the phenotype string once all codons are used, then the string of codons is reused, in a process called "wrapping" [1]; this is done until the phenotype string is valid, or a fixed number of wrapping events occur without a successful mapping.

III. Inserting Genotype Codon Values into Phenotype Strings

The first extension presented in this paper allows the extraction of genotype values, and their direct use in the generated phenotype strings, as a means of constant creation. This section presents the motivation for this extension, details of its implementation, and some example grammars.

A. Motivation

Ephemeral random constants are the standard approach to constant creation in GP, having values created randomly within a pre-specified range at a run's initialisation [7]. These values are then fixed throughout a run, and new constants can only be created through combinations of these values and other items from the function and terminal set.

A number of variations on the ephemeral random constant concept have been applied in tree-based GP systems, all of which have the common aim of making small changes to the initial constant values.

Constant Perturbation [10] allows GP to fine-tune floating point constants by rescaling them by a factor between 0.9 and 1.1. This has the effect of modifying a constant's value by up to 10% of its original value.

Numerical Terminals and Numerical Terminal Mutation are another approach [11]. The numerical terminal mutation operator selects a real valued numerical terminal in an individual and adds a Gaussian distributed noise factor, such that small changes are made to the constant values.

The Numeric Mutation operator [12] replaces the numeric constants in an individual with new ones drawn at random from a uniform distribution with a pre-specified range. The selection range for each constant is specified as the old value of that constant plus or minus a temperature factor.

Linear Scaling [13], [14], [15] has been used to optimise values within a local neighbourhood. It uses linear regression on the values expressed, where a line is derived to fit the data and new values explored in the neighbourhood.

A recent study [16] used two forms of constant mutation, Creep and Uniform mutation, where values are altered by a small amount or mutated to a randomly different number. The study found greater benefits from uniform mutation, on the problems analysed, where the ability to introduce new constants into a population as evolution progresses and maintain a highly diverse array of constants is generally beneficial to the fitness of individuals.

With Ephemeral random constants as their base each of these methods focused on changing the original random values by small amounts to improve fitness, with one exception [16], which also examined wholesale transformation of constant values and found this feature to be more beneficial than slight changes. GE can borrow from the experience of GP by extending the established methodology, and introducing grammar based constant creation through digit concatenation, which addresses the issue of beginning an evolutionary run with a fixed range of constants, and also provides the feature of being able to create new values over the course of a run [17], [18], [19].

B. Implementation

This new approach consists in extracting codon values and integrating them directly into the generated phenotype strings. This is achieved through the use of the following special symbol in the BNF grammar:

<GECodonValue[-x_l][+x_u]>

During the mapping process, if this symbol is encountered, it is replaced with the numerical value of the current codon, and this codon is consumed. The -x_l and +x_u are optional specifications of a lower and upper limit for this value: if present, the codon value is mapped to the range specified, using the mod operator; if omitted, the exact value of the codon is used[1].

This creates the possibility of inserting genotypic values directly into phenotype strings, and should not affect the performance of GE, as its behaviour will be the same as if an exhaustive enumeration of the sequential constants was done in a grammar rule.

C. Examples

A typical usage of this symbol is when a decimal value is required:

```
<expr>  ::= <var>=<value>
<var>   ::= x | y | z
<value> ::= <GECodonValue>
```

If a specific range is required for each of these variables, then range specifications can be used, such as:

```
<expr>  ::= x=<xvalue>
          | y=<yvalue>
          | z=<zvalue>
<xvalue> ::= <GECodonValue-0+100>
<yvalue> ::= <GECodonValue-101+200>
<zvalue> ::= <GECodonValue-201+300>
```

The use of this symbol can also replace the need of exhaustive enumeration of constants in grammars. For example, this grammar:

```
<expr>  ::= <expr> <op> <expr>
        |   <value>
<op>    ::= + | - | / | *
<value> ::= 0 | 1 | 2 | 3 | 4
          | 5 | 6 | 7 | 8 | 9
```

can be replaced by the following one:

```
<expr>  ::= <expr> <op> <expr>
        |   <GECodonValue-0+9>
<op>    ::= + | - | / | *
```

which is functionally equivalent. The use of this symbol clearly creates a more elegant and easier to read and write grammar.

IV. Grammar-Defined Crossover Markers

The second extension presented in this paper permits the specification of crossover locations through the grammar, which in turn allows the grouping of context-sensitive data together and their exchange as blocks, as well as the evolution of crossover boundaries. The motivations, implementation details and examples of this feature follow.

[1]As GE typically uses 8 bit codons, that value will therefore be in the range $0..255$.

A. Motivation

The evolution of possible crossover points in genetic algorithms is an idea which has been tackled by many previous studies. Rosenberg [20] proposed the evolution of the probability of crossover between genes; Schaffer and Morishima [21] proposed the Punctuated Crossover, where each individual carried an extra crossover mark string, of the same length of the individual, marking the possible locations where crossover could occur. Many further studies used similar techniques, with considerable success [22], [23].

A considerable amount of research has also been done in GP to allow the choice of crossover points between trees, but many of those studies require the periodic calculation of measures of performance.

One notable exception is the work of Angeline [11], who proposed two self-adaptive crossover operators, termed Selective Self-Adaptive Crossover (SSAC) and Self-Adaptive Multi-Crossover (SAMC). SSAC works by associating with each individual an equally shaped parameter tree, with functions and terminals replaced with probabilities of crossover at those points; once a point in each individual is chosen, crossover is applied to both the individual tree and its parameter tree, and random noise is added to the parameter tree. SAMC works in a similar way, but allows for multiple crossover points between the two individuals.

Edmonds [24] pushed this idea forward, and augmented standard GP with the co-evolution of genetic operators, which are represented as trees with their own set of functions and terminals, and which evolve throughout the run.

B. Implementation

As before, a special symbol is used in the BNF grammar:

```
<GEXOMarker>
```

Whenever this symbol is encountered, it marks where a crossover operation can take place at the genotypic level. This requires that an extra structure is kept with each genotype string, containing a sequence of possible crossover locations (which we will refer to as *XOSites*).

When a genotype string is mapped into a phenotype program, the *XOSites* structure is reset. Then, everytime the symbol `<GEXOMarker>` is encountered, it is removed from the phenotype program, and the location of the currently used codon is inserted into *XOSites*. Depending on the individual and the grammar, this will mean that each genotype string will have 0 or more possible crossover locations associated with it, contained in the *XOSites* structure.

Once two genotype strings are picked for a crossover event, rather than picking a random location for the crossover point in each[2], an element of their corresponding *XOSites* is chosen instead. If *XOSites* is empty on either of them, then the corresponding individual will not exchange information.

[2]GE typically uses a 1-point crossover operator

C. Examples

A typical usage of this symbol is when the problem is composed of subproblems or blocks, known *a priori*:

```
<prog>  ::= v1=<expr>;<GEXOMarker>
            v2=<expr>;<GEXOMarker>
            v3=<expr>;<GEXOMarker>
            v4=<expr>;
<expr>  ::= <expr> <op> <expr>
        |   <GECodonValue-0+9>
<op>    ::= + | - | / | *
```

If, by domain knowledge, it is known that the variables in the example grammar contribute independently to the fitness of the program, then the use of <GEXOMarker> can actively promote the exchange of variables as a whole.

Another interesting possibility is to let GE evolve the location of these markers, as shown in the following grammar:

```
<prog>  ::= v1=<expr>;<XO>
            v2=<expr>;<XO>
            v3=<expr>;<XO>
            v4=<expr>;
<expr>  ::= <expr> <op> <expr>
        |   <GECodonValue-0+9>
<op>    ::= + | - | / | *
<XO>    ::= <GEXOMarker> |
```

In this example, predefined locations for the crossover markers are specified, but their usage is left to evolution, as the symbol <XO> can be replaced by either <GEXOMarker>, or nothing at all.

The use of this approach opens up many possibilities. For example, the use of a two-point or even multi-point crossover operator would be an interesting approach. This approach could also be used to minimise the negative effects of the ripple effect [9], and it could also facilitate the implementation of many context-aware crossover operators for GE [25], without the need of analysing the derivation tree of each individual.

V. Problem Domains

In this study we illustrate the use of the new grammar extensions for GE. Initially we examine <GECodonValue> with a view to comparing it against the standard method for generating and evolving constants using grammars in GE. For this, two problems are examined: the evolution of a static high precision floating point number, and a dynamic benchmark problem where the targets to be evolved are changed every 100 generations.

The behaviour of the <GEXOMarker> extension is examined using three different grammars, one as a control without the new feature, one with the <GEXOMarker>, and a third one with the <GEXOMarker> as optional. A symbolic regression problem is examined in these experiments.

A. <GECodonValue>

Two grammars are used for experiments examining the behaviour of <GECodonValue>. The benchmark grammar is the same as that identified in [17], [18], [19] as providing the best performance and adaptability in the evolution and generation of constants in GE, and is provided below.

```
<exp>::= <exp> <op> <exp>
       | <real>
<op>::= + | - | * | /
<real>::=<int>.<int>|<int>
<int>::= <int><digit>|<digit>
<digit>::=0|1|2|3|4|5|6|7|8|9
```

This grammar uses digit concatenation for the creation of its real numbers. The <digit> non-terminal defines the digits 0 to 9, while the <int> non-terminal allows integers to be formed by concatenating digits together. These integers can contain 1 to *n* digits where the size is limited only by the length of the individual.

The second grammar is similar, but makes use of the new <GECodonValue> extension:

```
<exp>::= <exp> <op> <exp>
       | <real>
<op>::= + | - | * | /
<real>::=<int>.<int>|<int>
<int>::= <int><GECodonValue-0+9>
          |<GECodonValue-0+9>
```

Here the non-terminal <digit> of the benchmark grammar is rendered obsolete. The use of <GECodonValue> allows us to take the literal value of the codon being consumed as the constant value in the phenotype. By specifying the range $-0+9$, this results in the calculation of the modulus of the codon value by 10, prior to inserting it on the phenotype string. In effect this allows the new grammar to produce the same phenotypic expressions as the benchmark grammar while using one less non-terminal.

1) Static Constant Creation: The aim of this problem is to evolve a single floating point constant. For these experiments, a high precision floating point number was chosen, 20021.11501. Fitness in these experiments is the absolute difference between the target and evolved values, the goal being to minimise this difference.

2) Moving Peaks Benchmark: This problem uses a dynamic function, the Moving Peaks Benchmark [26], [27], [28]. The implementation of this problem involves several peaks situated in a 5 dimensional landscape, where with the passage of every 100 generations, the height and location of the peaks is altered randomly. Fitness in this case is the Moving Peaks function value returned for each individual, which is to be minimised:

$$F(\vec{x},t) = \max(B(\vec{x}), \max_{i=1\ldots m} P(\vec{x}, h_i(t), w_i(t), \vec{p}_i(t)))$$

where $B(\vec{x})$ is a time-invariant “basis” landscape, and P is the function defining a peak shape, where each of the m peaks has its own time-varying parameters height (h), width (w), and location $(\vec{p})$. The objective of the system is to evolve the contents of the 5-dimensional $\vec{p}$ vector.

B. <GEXOMarker>

The experiments examining the behaviour of <GEXOMarker> used three different grammars. Again a benchmark grammar was used without the <GEXOMarker> extension as follows:

```
<s>::= a = <exp>; b = <exp>;
       c = <exp>; d = <exp>;
       e = <exp>; f = <exp>;
<exp>::= <exp><GECodonValue-0+9>
         |<GECodonValue-0+9>
```

In this grammar the variables **a** through **f** are being assigned integer values as a result of the concatenation of the digits generated by the non-terminal <exp>. As already described, these integers can contain multiple digits, and under such circumstances crossover can occur in the middle of these integers, yielding potentially destructive effects.

The second grammar was designed to address this problem, splitting each variable assignment into an atomic unit as far as crossover is concerned, through the use of the <GEXOMarker> symbol:

```
<s>::= a = <exp>;<GEXOMarker>
       b = <exp>;<GEXOMarker>
       c = <exp>;<GEXOMarker>
       d = <exp>;<GEXOMarker>
       e = <exp>;<GEXOMarker>
       f = <exp>;
<exp>::= <exp><GECodonValue-0+9>
         |<GECodonValue-0+9>
```

Due to the presence of <GEXOMarker>, crossover cannot occur in the middle of an integer, only at the points specified in the grammar. This does not affect the mutation operator, which may still be applied across the individual.

The final grammar tested presents a variation on the second, by making the use of <GEXOMarker> optional:

```
<s>::= a = <exp>;<XO>
       b = <exp>;<XO>
       c = <exp>;<XO>
       d = <exp>;<XO>
       e = <exp>;<XO>
       f = <exp>;
<exp>::= <exp><GECodonValue-0+9>
         |<GECodonValue-0+9>
<XO>::=<GEXOMarker>|
```

This grammar allows for the possibility of one or more variables to be treated as atomic units under crossover. The non-terminal symbol <XO> enables this process by being replaced with either <GEXOMarker> or a void, giving evolution the power to protect beneficial groups of vars from being destroyed in crossover.

1) Symbolic Regression: These three grammars are examined against a symbolic regression problem [29], in which a set of coefficients is evolved and plugged into an equation. We departed slightly from Wu and Garibay's approach by adding an extra coefficient for the purposes of these experiments, in order to create a more complex environment.

Considering this, the purpose of these experiments is to evolve 6 coefficients for the equation

$$f(x) = \mathbf{a}x^3 + \mathbf{b}x^2 + \mathbf{c}x + \mathbf{d} + \mathbf{e}\cos(x) + \mathbf{f}\sin(x) \qquad (1)$$

The target coefficients are generated randomly at the beginning of each run. Rather than trying to match the target coefficients, the fitness score is calculated from the ratio of difference between the target data points and the function values generated using the evolved coefficients. This ratio is calculated for each target data point as follows:

$$ratio(i) = \begin{cases} \frac{d_i}{f(x)} & \text{if } d_i < f(x) \\ \frac{f(x)}{d_i} & \text{otherwise} \end{cases}$$

where d_i is the target data point and $f(x)$ the function value with the generated coefficients. The fitness is simply the average of all ratios, across all n data points (in our experiments, 100 were used, in the range $0..99$):

$$fitness = \frac{\sum_{i=1}^{n} ratio(i)}{n}.$$

Although not a standard GP symbolic regression problem, this problem is suitable to analyse the effect of specifying crossover locations, as it is composed of clear, separable blocks, which interact between them.

VI. Results and Analysis

For every problem instance, 30 runs were conducted using population sizes of 500, adopting one-point crossover at a probability of 0.9, and mutation at 0.1, along with roulette-wheel selection and a generational rank replacement strategy, where the worst performing 25% of the population is replaced each generation.

A. <GECodonValue>

The goal of these experiments was to examine if the benefits in grammar design and implementation introduced by the new extension had a detrimental effect on performance.

1) Static Constant: Each run for this problem was conducted over 50 generations. Fig. 2 displays the average best fitness for both grammars over 30 runs. By the final generation on average the benchmark grammar without <GECodonValue> produced an average error from target of 211.166 while the grammar with the new feature included produced an average error of 171.005 by the final generation. However, close examination of these results across the 30 runs, using a t-test and a bootstrap t-test, showed no statistical significance in the difference.

2) Moving Peaks Benchmark: This experiment was conducted over 500 generations with the peaks being moved every 100 generations leading to four different target values over the course of a run. Fig. 3 displays the average error of each grammar. In this case again a t-test and a bootstrap t-test yielded no statistical significance in the difference between the results, while using either grammar.

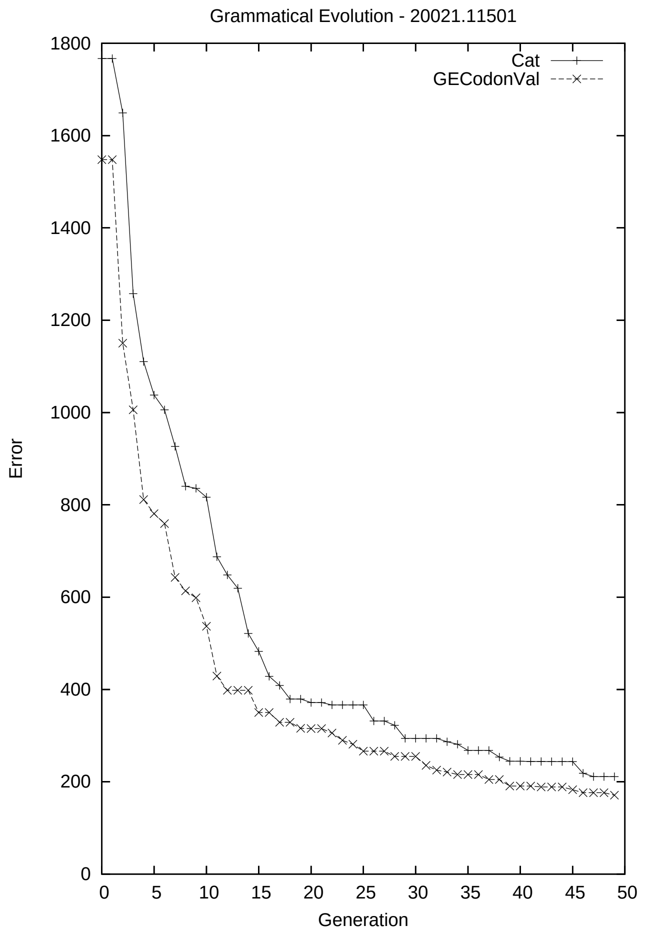

Fig. 2. Plot of the mean best fitness for the Static Constant problem, using a standard concatenation grammar and one including the `<GECodonValue>` extension (lower values are better).

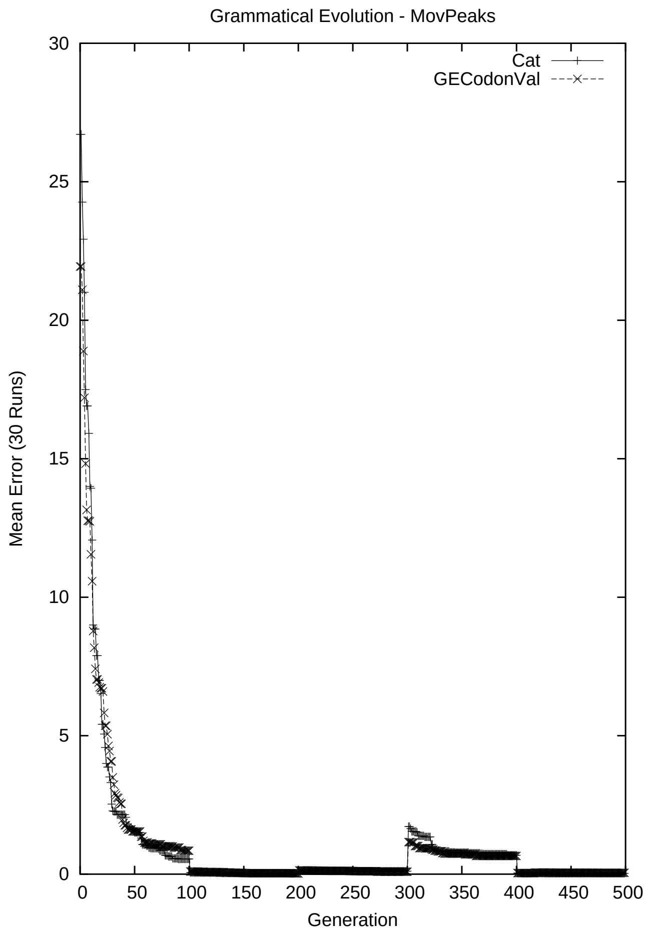

Fig. 3. Plot of the mean best fitness for the Moving Peaks problem, using a standard concatenation grammar and one including the `<GECodonValue>` extension (lower values are better).

B. *`<GEXOMarker>`*

The `<GEXOMarker>` potentially produces a significant difference in the function of the crossover operator. With this in mind these experiments were conducted in order to explore the behaviour of the new feature.

1) Symbolic Regression: Each run for this problem was conducted over 100 generations. Table I displays the average fitness by the final generation of both the best performing individual and the population mean along with the standard deviation, for the benchmark grammar, and the two grammars using crossover markers.

In this table it can be seen that the inclusion of the `<GEXOMarker>` symbol lead to an improvement in performance over the control grammar. When the `<GEXOMarker>` was made optional within the grammar a further improvement in results was again seen. After examining the results using a t-test and bootstrap t-test, a statistically significant improvement in performance was seen in the mean fitness for populations as a whole for the grammars which included `<GEXOMarker>`. This statistical significance was not carried through to the best performers

TABLE I

MEAN AND STANDARD DEVIATION ON THE SYMBOLIC REGRESSION PROBLEM. THE FIRST COLUMN DISPLAYS THE RESULTS OF THE DIGIT CONCATENATION GRAMMAR, THE SECOND THOSE OF THE COMPULSORY INCLUSION OF CROSSOVER MARKERS, AND THE FINAL ONE THOSE OF THE OPTIONAL INCLUSION OF THE CROSSOVER MARKERS

	Cat	**XOMarker**	**XOMaybe**
Best Mean	0.98831	0.99229	0.99264
Best Std	0.110992	0.0090787	0.0064935
Pop Mean	0.93218	0.95837	0.96474
Pop Std	0.0171193	0.0150013	0.0093356

from the population, although this is more a reflection of the complexity of the problem, as by the final generation each grammar in general produced a solution which very closely estimated the target coefficients.

It is also interesting to notice how the inclusion of `<GEXOMarker>` has minimised the standard deviation across all runs; this suggests that the (potentially) disruptive effects of crossover are minimised when using this feature. Allowing the locations of `<GEXOMarker>` to evolve sees a further decrease in standard deviation.

2) Further Experimentation: To investigate the implications of allowing the `<GEXOMarker>` to be optional within the grammar, a final experiment was conducted. Using the third grammar, 30 runs were conducted on the same symbolic regression problem, but this time for 500 generations.

On average by the final generation each run yielded a best performing individual with an average fitness of 0.99798 and mean fitness over the population of 0.984748. Fig. 4 displays the average number of `<GEXOMarker>`s per individual per generation across the 30 runs. By the final generation the system had reached an average of just 0.831667 crossover markers for each individual, suggesting that because the results at this stage closely estimated the target, GE attempted to evolve out the destructive capacity of crossover. Further analysis on the positioning of the `<GEXOMarker>`s reveals that the system over the course of evolution expressed a preference for placing the `<GEXOMarker>`s towards the end of the individuals. The significance of this is two-fold: firstly, because for the first two coefficients the *x* is raised to the power of 3 and 2 respectively, GE is seen to protect the individual from the potentially large changes in fitness by allowing crossover to disrupt this pair; secondly, because GE's mapping is left to right dependent, crossover points closer to the left have are more likely to inflict damage to a good performing individual, so over the course of evolution the crossover markers are seen to migrate to the right.

Table II displays the total number of `<GEXOMarker>`s that were present in each of the possible positions for the best performing individuals. From this we can see that GE had the least preference for conducting crossover after the first variable. The number of occurrences takes a step up after the second variable with a value of 7845, which correlates to the position after where *x* has been raised to powers in Eq. 1. The next three locations steadily rise after to the highest value at the final position which produces the least destructive impact for the equation under investigation and GE in general.

TABLE II
TOTAL NUMBER OF CROSSOVER MARKERS AFTER EACH VARIABLE OVER 30 RUNS FOR THE BEST PERFORMING INDIVIDUALS

After	**a**	**b**	**c**	**d**	**e**
Total	5969	7845	7540	7787	9250

VII. CONCLUSIONS

By using a mapping process, Grammatical Evolution creates a distinct separation between its search and solution spaces, allowing the evolution of genotypic strings independently of their phenotypic equivalents; this is in contrast with Genetic Programming, in which trees representing the phenotype structures are evolved.

Up to now, the only relationship between genotype and phenotype strings was a shared fitness measure. In this study, this genotype/phenotype separation is somewhat blurred, in that more information is shared between the search and solution spaces: the `<GECodonValue>` symbol allows phenotype strings to directly manipulate raw genotypic data, while the `<GEXOMarker>` symbol allows the crossover operator to function based on information gathered during the mapping process.

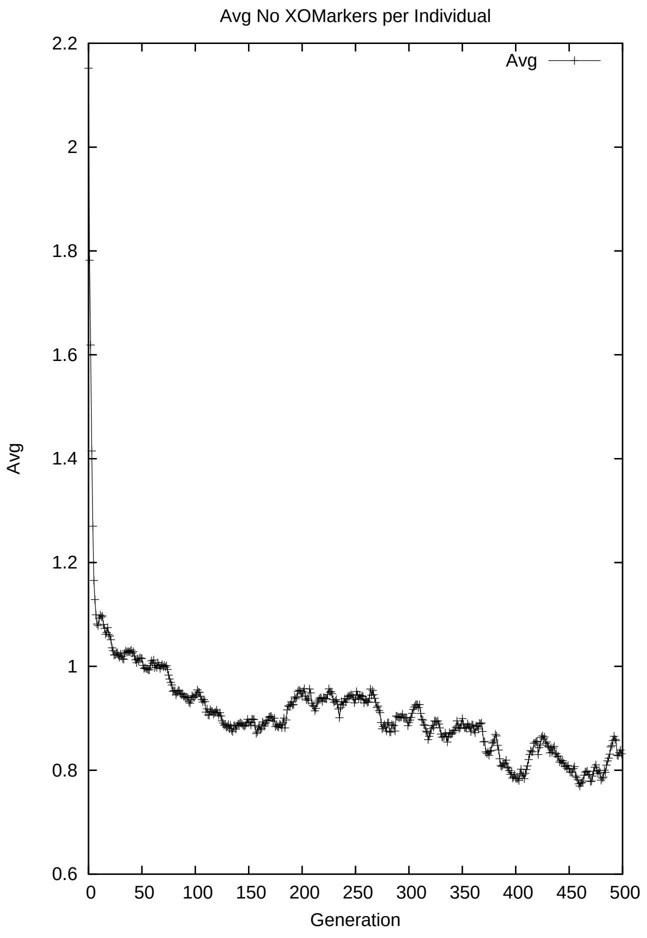

Fig. 4. Plot of the average number of crossover markers per individual over 500 generations, using a grammar which optionally includes these markers.

The experimental results obtained with the `<GECodonValue>` extension show that its implementation does not affect the performance of GE. Its use however presents non-negligible advantages in grammar design, simplifying the definition of ranges of constants.

The results obtained with the `<GEXOMarker>` extension are more substantial, in that they improve the quality of the solutions generated, in the experiment analysed. A deeper analysis of these results reveals that the crossover points each individual carries tend to evolve, in an effort to protect information already discovered.

Future work will involve a deeper analysis of these extensions; more experimentation is needed, across a wider set of problems, and a better understanding of the correlation between search and solution spaces is required. The use of `<GECodonValue>`, for example, allows the introduction of localised mutation rules for the manipulation of the constants generated; the effect of carrying these manipulated constants back to the genotype string should be analysed. Also, with the evolution of the location of `<GEXOMarker>` symbols, more forms of crossover can be examined, to explore the potential for discovery and evolution of derivation sequences within GE.

References

[1] O'Neill, M. and Ryan, C.: Grammatical Evolution - Evolving programs in an arbitrary language. Kluwer Academic Publishers. (2003)

[2] Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley. (1989)

[3] O'Sullivan, J.: An Investigation into the use of Different Search Engines with Grammatical Evolution. MSc Thesis, University of Limerick. (2001)

[4] O'Neill, M., and Brabazon, A.: Grammatical Swarm. In: Deb et al. (Eds.): Genetic and Evolutionary Computation - GECCO 2004. Lecture Notes in Computer Science, Vol. 3103. Springer. (2004) pp. 163–174

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

[6] O'Neill, M., and Ryan, C.: Grammatical Evolution. IEEE Transactions on Evolutionary Computation, Vol. **5**, No. 4. (2001) pp. 349–358

[7] Koza, J.: Genetic Programming. MIT Press. (1992)

[8] Banzhaf, W.: Genotype-Phenotype Mapping and Neutral Variation - A case study in Genetic Programming. In: Davidor et al. (Eds.): Proceedings of the third conference on Parallel Problem Solving from Nature. Lecture Notes in Computer Science, Vol. 866. Springer-Verlag. (1994) pp. 322–332

[9] O'Neill, M., Ryan, C., Keijzer, M., and Cattolico, M.: Crossover in Grammatical Evolution. Genetic Programming and Evolvable Machines, Vol. **4**, No. 1. (2003) pp. 67–93

[10] Spencer, G.: Automatic Generation of Programs for Crawling and Walking. In: Kinnear, K. E. Jr. (ed): Advances in Genetic Programming, Chapter 15. MIT Press. (1994) pp. 335–353

[11] Angeline, P. J.: Two Self-Adaptive Crossover Operators for Genetic Programming. In: Angeline, P. J. and Kinnear, K. E. Jr. (eds.): Advances in Genetic Programming 2, Chapter 5. MIT Press. (1996) pp. 89–110

[12] Evett, M., and Fernandez, T.: Numeric Mutation Improves the Discovery of Numeric Constants in Genetic Programming. In: Genetic Programming 1998: Proceedings of the Third Annual Conference. University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann. (1998) pp. 66–71

[13] Iba, H., and Nikolaev, N.: Genetic Programming Polynomial Models of Financial Data Series. In: Proceedings of the 2000 IEEE International Congress on Evolutionary Computation. IEEE Press. pp. 1459–1466

[14] Nikoaev, N., and Iba, H.: Regularization Approach to Inductive Genetic Programming. IEEE Transactions on Evolutionary Computation, Vol. **5**, No. 4. (2001) pp. 359–375

[15] Keijzer, M.: Improving Symbolic Regression with Interval Arithmetic and Linear Scaling. In: Ryan et al. (Eds.): Proceedings of EuroGP-2003. Lecture Notes in Computer Science, Vol. 2610. Springer-Verlag. (2003) pp. 70–82

[16] Ryan, C., and Keijzer, M.: An Analysis of Diversity of Constants of Genetic Programming. In: Ryan et al. (eds): Proceedings of EuroGP-2003. Lecture Notes in Computer Science, Vol. 2610. Springer-Verlag. (2003) pp. 404–413

[17] O'Neill, M., Dempsey, I., Brabazon, A., and Ryan, C.: Analysis of a Digit Concatenation Approach to Constant Creation. In: Ryan et al. (Eds.): Proceedings of EuroGP-2003. Lecture Notes in Computer Science, Vol. 2610. Springer-Verlag. (2003) pp. 173–182

[18] Dempsey, I., O'Neill, M., and Brabazon, A.: Grammatical Constant Creation. In: Deb et al. (Eds.): Genetic and Evolutionary Computation - GECCO 2004. Lecture Notes in Computer Science, Vol. 3103. Springer. (2004) pp. 447–458

[19] Dempsey, I., O'Neill, M., and Brabazon, A.: meta-Grammar Constant Creation with Grammatical Evolution by Grammatical Evolution. In: Beyer et al. (Eds.): Genetic and Evolutionary Computation Conference - GECCO 2005. ACM 1-59593-010-8/05/0006. (2005) pp. 1665-1671.

[20] Rosenberg, R. S.: Simulation of Genetic Populations With Biochemical Properties. PhD Thesis, University of Michigan. (1967)

[21] Schaffer, J. D., and Morishima, A.: An Adaptive Crossover Distribution Mechanism for Genetic Algorithms. In Grefenstette, J. J. (Ed.): Proceedings of the Second International Conference on Genetic Algorithms. Lawrence Erlbaum Associates. (1987) pp. 36–40

[22] Levenik, J. R.: Metabits: Generic Endogenous Crossover Control. In Eshelman, L. J. (Ed.): Proceedings of the Sixth International Conference on Genetic Algorithms. Morgan Kaufmann Publishers. (1995) pp. 88–95

[23] Smith, J., and Fogarty, T. C.: Recombination Strategy Adaptation via Evolution of Gene Linkage. In: Bäck, T. et al. (Eds.): Proceedings of 1996 IEEE International Conference on Evolutionary Computation. IEEE Press. (1996) pp. 826–831

[24] Edmonds, B.: Meta-Genetic Programming: Co-evolving the Operators of Variation. CPM Report 98-32, Centre for Policy Modelling, Manchester Metropolitan University, UK.

[25] Harper, R., and Blair, A.: A Structure Preserving Crossover In Grammatical Evolution. In: Proceedings of the 2005 IEEE International Conference on Evolutionary Computation, Vol. 3. (2005) pp. 2537–2544

[26] Branke, J.: Evolutionary algorithms for dynamic optimization problems - a survey. Technical Report 387, Institute AIFB, University of Karlsruhe. (1999)

[27] Branke, J.: Memory Enhanced Evolutionary Algorithms for Changing Optimization Problems. In: Proceedings of the 1999 IEEE International Conference on Evolutionary Computation, Vol. 3. IEEE Press. (1999) pp. 1875–1882

[28] Morrison, R. W., and DeJong, K. A.: A Test Problem Generator for Non-Stationary Environments. In: Congress on Evolutionary Computation, volume 3. IEEE press. (1999) pp. 2047–2053

[29] Wu. A. S., and Garibay, I.: The Proportional Genetic Algorithm: Gene Expression in a Genetic Algorithm. Genetic Programming and Evolvable Machines, Vol. **3**, No. 2. (2002) pp. 157–192