An Examination on the Modularity of Grammars in Grammatical Evolutionary Design

John Mark Swafford and Michael O'Neill

Abstract— **This work furthers the understanding of modularity in grammar-based genetic programming approaches by analyzing how different grammars may be capable of producing the same phenotypes, but still display differences in performance on the same problems. This is done by creating four grammars with varying levels of modularity and using them with grammatical evolution to evolve floor plan designs. The results of this experimentation show how increases in modularity, brought about by simple modifications in the grammars, and increases in the quality of solutions go hand in hand. It also demonstrates how more modular grammars explore more individuals even while fitness remains the same or changes in only minor increments.**

I. Introduction

An evolutionary design system is any evolutionary algorithm used to solve or help solve a design problem; examples include, but are not exclusive to, aesthetic art, architecture, commercial products, and sculptures. When working with evolutionary design systems, having a representation that is able to take advantage of modularity is vital. While it has been shown that this characteristic may increase the search space of an evolutionary algorithm [4], it also allows for easier identication and exchange of useful information between individuals throughout evolution. A widely known fact among evolutionary algorithm researchers is that using representations capable of taking advantage of modularity greatly improve the scalability of their algorithm's problem solving capability.

The general idea behind modularity is that of being able to "break apart" a solution to any given problem and manipulate the pieces independently of each other. This could mean swapping pieces of information with other solutions, combining small pieces to create larger ones, or breaking down large pieces to be manipulated separately. This can be done in either a top-down or bottom-up manner. With the first approach, a complete solution is examined and useful modules are identied to be promoted in future generations of evolution. The bottom-up method takes small modules and attempts to combine them to create larger modules that represent a complete solution.

This preliminary work further extends the understanding of modularity in evolutionary design system in hopes of opening the door to more in-depth studies on the modularity of different representations for design in grammatical evolution (GE) [13], [3]. Using the grammar-based genetic programming algorithm, GE, as an approach to evolving designs, the importance of incorporating modular features in the problem representation can easily be seen. To illustrate how this approach can be used in real world problems, an analysis of this is given using GE to evolve floor plan designs. This problem was picked because it is particularly modular in nature and may be scaled up arbitrarily by increasing the size and number of rooms in the floor plan. This facilitates future studies on the scalability of approaches based on the lessons learned from this work. More specically, an analysis of how grammar shape/form impacts the modularity and tness of individuals in evolutionary design problems using GE is presented. More details on this are given in Section III.

The structure of this work is as follows: Section II gives prior work and background information on GE, modularity in genetic programming, and modularity in evolutionary design. Sections III and IV detail the experiments carried out for this work as well as the results and a discussion of their signicance. In Section V conclusions are drawn and possible avenues for future work is outlined.

John Mark Swafford, Complex and Adaptive Systems Laboratory, School of Computer Science and Informatics, University College Dublin, Clonskeagh, Dublin 4, Ireland (email: john-mark.swafford@ucdconnect.ie).

Michael O'Neill, Complex and Adaptive Systems Laboratory, School of Computer Science and Informatics, University College Dublin, Clonskeagh, Dublin 4, Ireland (email: m.oneill@ucd.ie).

II. Background Work

The idea of exploiting modularity in evolutionary algorithms has been examined in many ways. However, GE has yet to achieve the popularity of genetic algorithms (GAs) [6] and genetic programming (GP) [8], and the modular nature of GE has not been analyzed to the same extent. This section gives background information on GE (Section II-A), modularity in genetic programming (Section II-B), and modularity in evolutionary design (Section II-C).

A. Grammatical Evolution

Grammatical evolution (GE) [13] is a grammar-based approach to GP. GE sets itself apart from typical GP by representing individuals as linear genotypes (in the implementation used here, these are integer arrays), instead of the usual GP trees. Elements in the genotype are referred to as "codons." A user-defined, context-free grammar is used to map these genotypes into phenotypes that are able to be interpreted by the algorithm's fitness function (this implementation uses context-free grammars in BNF form). GE allows these phenotypes to be strings, pieces of programs, or even complete programs. Following is an example of this mapping process. Given the integer array:

$$5 \quad 11 \quad 29 \quad 31 \quad 78 \quad 46 \quad 55$$

and a simplified version of the first grammar (Figure 1) used in the experiments in Section III:

```
<floor> ::= <rooms>
<rooms> ::= <room>;<rooms> | <room>
<room>  ::= <type>:<x>,<y>:<x>,<y>
<type>  ::= bed | bath | live | dine
<x>     ::= -3 | -2 | -1 | 0 | 1 | 2 | 3
<y>     ::= -3 | -2 | -1 | 0 | 1 | 2 | 3
```

an example of the genotype to phenotype mapping can be performed. GE's mapping process always begins with the grammar's start symbol and the left-most element in the integer array. For this example, the start symbol is `<floor>` and the starting element in the integer array is 5. Because the `<floor>` non-terminal has only one production, it is picked and replaces the previous non-terminal. Since there was only one choice for the next production rule, the current codon, 5, was not used. So, `<floor>` is replaced by `<rooms>` . The `<rooms>` non-terminal has two productions so the modulo operator is used to determine which production is picked. The equation for this is simple:

codon % # of productions = production picked.

For this example the equation would be: $5\%2 = 1$, so the first production will replace the current left-most non-terminal. This means `<rooms>` will be replaced by `<room>` . The `<room>` non-terminal also has only one production, so it is replaced by `<type>:<x>,<y>:<x>,<y>` and no codons are used. The left-most non-terminal in the current string is `<type>` , which has four possible productions, the current codon value is 11, and $11\%4 = 3$, so `<type>` is replaced with `dine` . This leaves the phenotype string as `dine:<x>,<y>:<x>,<y>` . The mapping process continues until there are no more non-terminals or the end of the integer array has been reached. For this specific example, the final phenotype is: `dine:-2,0:-2,1` . In the event that there are non-terminals left in the phenotype string, the individual that produced it will be designated as *invalid* and receives the worst possible fitness value. In hopes of repairing invalid individuals, wrapping may be used. With this feature, when there are no more unused codons in the integer array, mapping will resume with the first codon in the array and the entire array may be used again. Wrapping may only be allowed a limited amount of times in case an individual is infinitely recursive.

B. Modularity In Genetic Programming

SinceGAs [6] became popular as a problem solving method, the more detailed characteristics of these evolutionary algorithms have been intensely studied. One such aspect is modularity. Shortly after the rise of genetic programming GP [8] as an alternative to GAs, researchers began studying the modularity of different GP approaches and representations. It did not take long to realize that GP was especially well suited to solving problems with a modular nature and that representations should be capable of taking advantage of this modularity to navigate through the search space more efficiently and to increase the scalability of the algorithm.

Due to the wide variety of problems that have been tackled by GP and GP-based approaches and the large differences in the representations used, most, if not all, studies involving modularity use a different definition of this characteristic. Woodward [16], in his analysis of modularity in GP, defines modularity as: "a function that is defined in terms of a primitive set or previously defined modules." This is quite a general definition because it states that modules can be independent of each other or can be defined in terms of other modules.

Angeline and Pollack [1] explain their method for picking out modules of useful information and passing them through the evolutionary generations. They use a *compress* operation to randomly "lock" a piece of an individual so it will be passed on to the offspring of that individual without any modification. They also define an *expand* operation that "unlocks" all or part of a compressed piece of an individual. The *compress* operation was also extended such that compressed pieces of individuals could be compressed again with more information from an individual, creating a hierarchy of compressed modules that can be reused as a whole. They called this process *atomization*. Their results show how beneficial capturing these modules were during the course of an evolutionary run. This work shows the first step in basic module encapsulation and how advantageous this can be. Here, evolution was allowed to freely discover and use modules, but these results raise the question if there is a more "intelligent" way of identifying useful modules before "locking" them and passing them through future generations.

The work carried out by Krawiec and Weiloch [10] define a new way to exploit modularity, called *functional modularity*, that could be used to address some of future work mentioned by Angeline and Pollack [1]. This approach attempts to discover modules without using the context of the problem as a whole. They give the example of a battery and a bulb in a flashlight. If the battery is a module, there may be a way to analyze how good of a battery it is without using it in the flashlight. This is done by using the semantics (what the module means instead of the symbols that produced it) of the module. The results of their experimentation show that functional modularity can be valuable in determining a problem's composition and difficulty. They also discuss how their approach exploits these characteristics in a problem and also how this is a difficult task and requires further studies to fully understand.

When undertaking any study on modularity, it is important to acknowledge Koza's [9] Automatically Defined Functions (ADFs) and the benefits they bring in terms of taking advantage of the modularity of a representation. However, ADFs are beyond the scope of this work as it is more focused on the modularity that can be found in simple grammars, independent of that found in representations using ADFs. See Section V for more detail as to how ADFs are planned for future, related work.

C. Modularity In Evolutionary Design

As modularity has been shown to be important in standard GP implementations, it has been found to be crucial in evolutionary design problems. Lipson et al. [11] discuss the importance of modularity in evolutionary design. They mention how scaling is always a large challenge for evolutionary systems, but modularity helps to alleviate this problem. They define modularity as "the separability of a design into units that perform independently." They also claim that modularity and regularity are independent because modules that are completely non-regular may exist. They hypothesize that modularity may be promoted by changing the environment or fitness criteria over time, forcing useful modules to be discovered and passed on. Their approach to modularity may also operate in a top-down fashion where working designs may be decomposed to determine which modules are beneficial.

Hornby [7], who defines modularity to be, "an encapsulated group of genotypic elements that can be manipulated as a group," shows how evolutionary design systems greatly benefit from having representations that are modular in nature. His example is the design of a table where modules may form the a leg of the table and then be reused to complete the other three legs. The target table designs are symmetrical, making the problem especially modular in nature as there are many pieces of the table that can be formed and reused to complete the design. He also shows how beneficial hierarchy and regularity are in the representation, both of which are enabled by modularity.

Modularity in evolutionary design does not necessarily have to be in regards to a product or design's composition. Gershenson [5] not only discusses the importance of modularity in the typical sense of design's composition or form, but also the life-cycle of a product. The elements of a life-cycle of a design may include manufacture, assembly, service, and recycling. While the modularity of a product's life-cycle is outside the scope of this work; Gershenson's work [5] enforces that modularity may play a more crucial role in the real-world applications as opposed to benchmark problems used solely for research purposes.

III. Experimental Setup

The first goal of this work is to show how different context-free grammars, all of which are capable of producing the same phenotypes, have varying levels of facilitating modularity. The second goal of this study is to show how the varying levels of modularity enabled by the grammars affect the performance of GE on design problems of varying difficulty. To accomplish these goals the following definition of modularity is used: *A module is any sub-derivation tree or group of sub-derivation trees in an individual.*

To measure how well grammars with varying levels of promoting modularity impact evolutionary search, five target floor plans were given for evolution to match (Figure 6) and four grammars, each with different capabilities for representing modules and creating different derivation tree structures (see Figures 1 - 4 for the grammars used). Each of these grammars produce strings of the form *roomType:x-position,y-position:width,height;roomType:...* that are translated into floor plan images. It it crucial to note that although each grammar is capable of producing the same phenotype strings, but the derivation trees may look very different.

```
<floor> ::= <rooms>
<rooms> ::= <room>;<rooms> | <room>
<room> ::= <type>:<x>,<y>:<x>,<y>
<type> ::=  bed | bath | live | close
         | dine | kitch
<x> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
<y> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
```

Fig. 1. Grammar 1

```
<floor> ::= <rooms>
<rooms> ::= <room>;<rooms> | <room>
<room> ::= <type>:<pos>:<size>
<pos> ::= <x>,<y>
<size> ::= <x>,<y>
<type> ::=  bed | bath | live | close
         | dine | kitch
<x> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
<y> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
```

Fig. 2. Grammar 2

```
<floor> ::= <rooms>
<rooms> ::= <combiRooms> <rooms>
          | <combiRooms> <room> | <room>
<combiRooms> ::= <one> | <two> | <three> | <four>
               | <five> | <six>
<one> ::= <room>;
<two> ::= <room>;<room>;
<three> ::= <room>;<room>;<room>;
<four> ::= <room>;<room>;<room>;<room>;
<five> ::= <room>;<room>;<room>;<room>;<room>;
<six> ::= <room>;<room>;<room>;<room>;<room>;<room>;
<room> ::= <type>:<x>,<y>:<x>,<y>
<type> ::=  bed | bath | live | close
         | dine | kitch
<x> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
<y> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
```

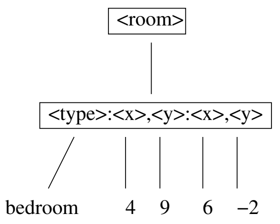
Fig. 3. Grammar 3

Each target image represents a floor plan where the different colored blocks are different rooms. Each of the targets has a different number of rooms, some with smaller rooms inside the larger ones. There are also different types of rooms e.g. bedrooms, closets, dining rooms, etc. As the number of rooms increase, some of the additional rooms are identical to existing rooms, but in different positions. This was purposely done to examine how each grammar performs in terms of capturing useful modules and reusing them in future generations. Using targets of increasing difficulty (targets larger numbers of rooms and more rooms of different types) also allows for the examination of how the different grammar scale to harder problems (targets with more rooms).

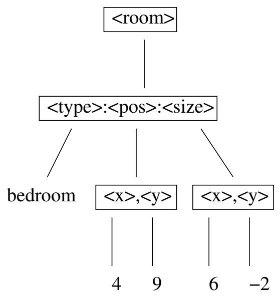
The problem was constructed in this manner because in order for an evolutionary algorithm to take advantage of the modularity in a representation, it must exist in the problem [2]. While this is not as prevalent in the easier problems, as there are fewer rooms in these problems, the targets may be broken down into individual rooms. Rooms can be further decomposed in the room type, location, and size. This allows for useful pieces of the problem to be discovered

```
<floor> ::= <rooms>
<rooms> ::= <combiRooms> <rooms>
          | <combiRooms> <room> | <room>
<combiRooms> ::= <one> | <two> | <three> | <four>
               | <five> | <six>
<one> ::= <room>;
<two> ::= <room>;<room>;
<three> ::= <room>;<room>;<room>;
<four> ::= <room>;<room>;<room>;<room>;
<five> ::= <room>;<room>;<room>;<room>;<room>;
<six> ::= <room>;<room>;<room>;<room>;<room>;<room>;
<room> ::= <type>:<pos>:<size>
<pos> ::= <x>,<y>
<size> ::= <x>,<y>
<type> ::=  bed | bath | live | close | dine
         | kitch
<x> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
<y> ::= -10 | -9 | -8 | ... | 8 | 9 | 10
```

Fig. 4. Grammar 4

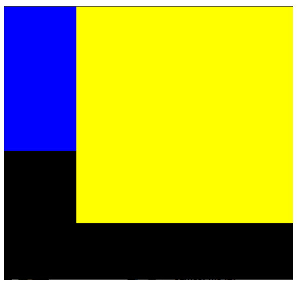


(a) Grammar 1

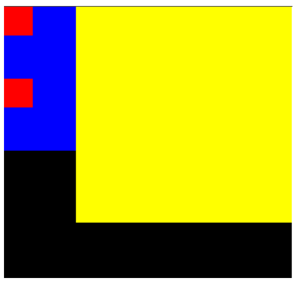


(b) Grammar 2

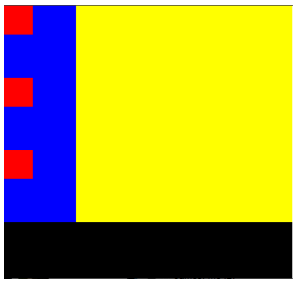
Fig. 5. Examples of possible sub-derivation trees from Grammars 1 and 2



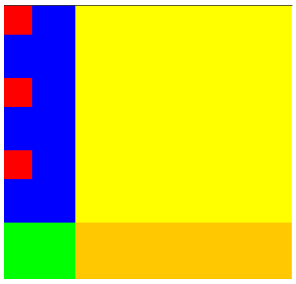
(a) Easy Target



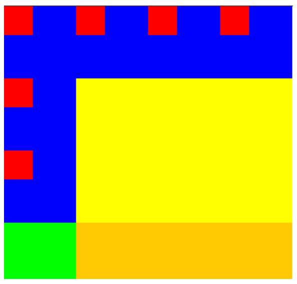
(b) Mid-Easy Target



(c) Medium Target



(d) Mid-Hard Target



(e) Hard Target

Fig. 6. Target floor plans (The black areas denote spaces where there are no rooms)

as modules and reused elsewhere and/or preserved through future generations. In a randomly constructed problem, this may not be the case. When the problem itself has little-to-no notion of modularity, attempts at decomposing the problem into smaller and more manageable pieces would be fruitless.

The fitness function used in this work is a weighted aggregate of four different values:

1) The difference in the number of rooms in the target design and the generated design;
2) If the generated designs have the same number of each type of room as the target design;
3) The distance from the starting point (top-left coordinate) of each room in the generated individual to their correct starting point in the target individual;
4) The distance from the ending point (bottom-right coordinate) of each room in the generated individual to their correct ending point in the target individual.

Each of these measures was scored on a per-room basis, meaning that there was a best and worst score for every design, depending on how many rooms existed in the target. If there were too few or too many rooms in the generated design, they would be penalized per room for this. Each of the measures listed above was given an equal proportion (25%) of the total fitness. Fitness values ranged from 0 to 100; 0 being the best and 100 being the worst. The rest of the experiment parameters used can be seen in Table I.

By the definition of modularity given above, the modularity of the population is calculated and measured at every generation. At each generation, the modularity of the population was measured in terms of occurrences of the following non-terminals in the derivation trees of the individuals:`<room>`, `<pos>`, and `<size>` . These non-terminals were picked because `<pos>` and `<size>` were deliberately omitted from some grammars while they were included in others and `<room>` was included in all grammars. This enables a comparison of the different approaches to determine if the changes manifested in the derivation trees by the alterations in the grammars yield any significant differences on any aspect of the population and its performance.

The reasoning behind the differences in the production rules used in the grammars is simple. The grammars which produce larger derivation trees, both in depth and width, will be more modular because they are capable of producing more and/or larger sub-derivation trees. For example, Grammar 1 (Figure 1) is the least modular of all the grammars because it creates the most minimal derivation trees during the genotype to phenotype mapping process. Grammar 2 (Figure 2) is much more modular, because is has `<pos>` and `<size>` non-

TABLE I
EXPERIMENTAL PARAMETERS

Parameter	Value
Number of Runs	50
Initialization	Ramped Half And Half
Pop. Size	1000
Generations	500
Elites	2
Selection	Tournament (Size 2)
Replacement	Generational
Crossover	Single Point (70%)
Mutation	Integer Flip (02%)
Wrapping	None
Max Derivation Tree Depth	10

terminals which encapsulate the position coordinates and width and height respectively. The third grammar, Grammar 3 (Figure 3), is modular in a different fashion than the previous two. It introduces a new type of modularity where more than one room may be created at once using the `<combiRooms>` non-terminal. The fourth and final grammar, Grammar 4 (Figure 4), is the most modular combining the position and size encapsulation from Grammar 2 with the creation of multiple rooms at once with Grammar 3.

In Figure 5, an example of the differences in possible sub-derivation trees generated from Grammars 1 and 2 may be seen. Figure 5(a) demonstrates how the most simple representation, Grammar 1, generates a sub-derivation tree that yields the phenotype: *bedroom:4,9,:6,-2*. Similarly, the sub-derivation tree generated by Grammar 2 in Figure 5(b) creates an identical phenotype, but with a different sub-derivation tree. Note how Figure 5(b) has the additional `<pos>` and `<size>` non-terminals. These encapsulate the `<x>` and `<y>` non-terminals allowing them to be modified or captured independently or as a group. The same principal applies to the addition of the `<combiRooms>` non-terminal in Grammars 3 and 4. The next section shows how these grammars with different levels of modularity perform in comparison to each other on an evolutionary design problem at five levels of difficulty.

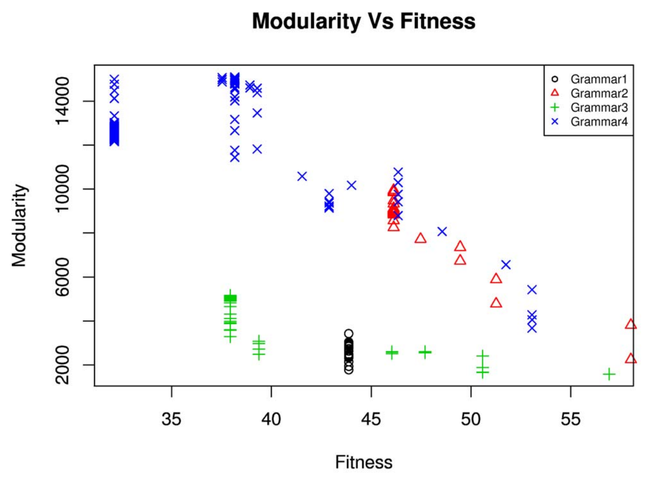
IV. RESULTS AND DISCUSSION

Fifty independent runs for each grammar/target combination were carried out using Grammatical Evolution in Java (GEVA) [12]. Figure 7 shows the correlation between modularity and fitness for each of the problems. It is easy to see that for any of the problems and grammars used in this work, as more modularity is found in the population, fitness decreases (in GEVA fitness is minimized). It is also interesting to note how in four out of five targets, the grammars which facilitate more modular derivation trees perform better than or as well as the less modular grammars.

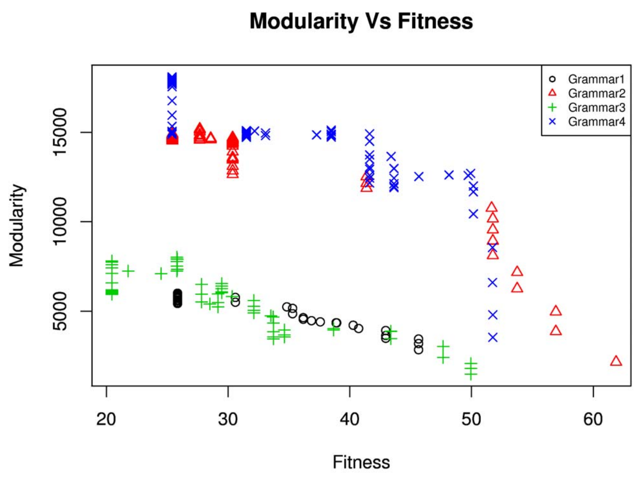
Starting with the simplest problem, Figure 7(a) shows how the most modular grammar, Grammar 4, out-performs all the others. For the next problem in order of difficulty, Figure 7(b) shows the more modular Grammars 2 and 4 slightly lagging behind the less modular Grammar 3, but performing approximately equally as well as the least modular Grammar 1. The middle problem in difficulty, Figure 7(c), once again demonstrates how the more modular grammars are able to perform at least as well as or better than those producing less modular solutions. The fourth problem, Figure 7(d) shows some of the most encouraging results as the most modular grammar outperforms all the others with high concentrations of very modular solutions around the best fitness values found for the run. For the hardest problem, Figure 7(e) shows all the grammars performing similarly in terms of the best fitness values found, but the more modular grammars start with better fitness values and have wider dispersion around the best fitness values they found. The only definite trend that can be seen in regard to the relationship between each grammar's fitness and modularity value is the correlation that as fitness improves (decreases) the modularity also increases. Since this is universal across all difficulty of problems and grammars, there is a definite relationship between increases in modularity and increases in performance.

The plots in Figures 7(a) - 7(e) also give some information about the diversity of the population in terms of fitness. Each graph contains the same amount of data points for each grammar. Starting with the easiest problem, Figure 7(a), relatively few points can be seen for each grammar. This suggests that the modularity/fitness values are clustered very tightly together and that evolution is more or less neutral for a large portion of the run. As the targets get harder to match, a larger amount of points are visible on the graphs showing how the fitness and modularity values are changing over the course of evolution. By the second hardest problem, Figure 7(d), the trails created by different grammars are quite distinct and more filled out. This shows how evolution explores even more of the search space at the different values of fitness. Even the clusters of points where fitness stops improving are more spread than the easier problems showing how evolution may be neutral in terms of fitness, but not in terms of modularity. This can be enforced by Figure 8 as it shows how the improvements in fitness greatly slowed down after a relatively short number of generations. Figure 7(e), the plot of the most difficult problem, shows the distinct trails of the evolutionary search as well as the more modular solutions creating larger clusters of points where the change in fitness has more or less halted, meaning there is more exploration of solutions with very similar fitness values by these grammars. This can also be enforced by looking at Figure 9. It shows a similar trend to the previous problem where fitness only slightly improves over time, but there is still more exploration going on in terms of modularity. The statistical significance of the best fitness and total modularity plots can be found in Table II.

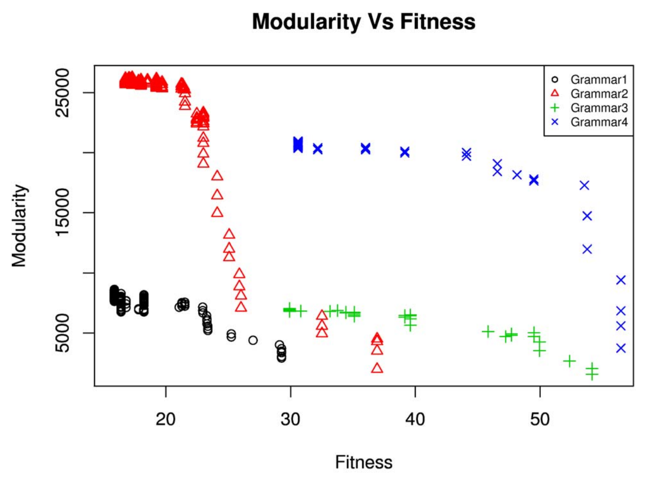
Further explanation of the shapes of the plots in Figure 7 can be attributed to the definition of modularity and the fitness function used in this work. Firstly, the definition and measure of modularity used here is quite simple. Recall that the modularity value of an individual was calculated by simply counting all the non-terminals of types `<room>` , `<pos>` , and `<size>` , and a module is defined as any sub-



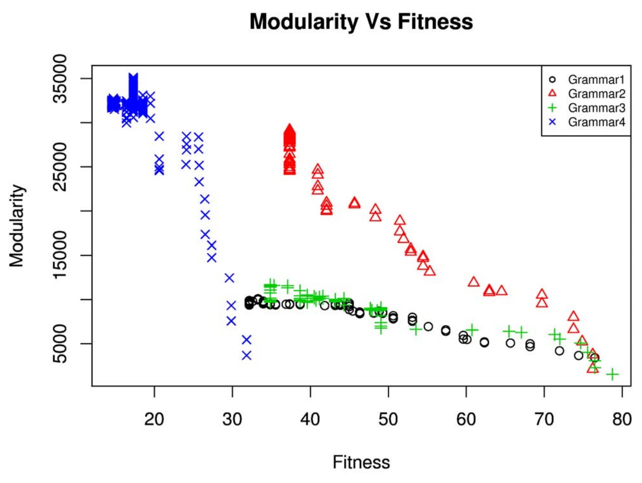
(a) Easy Target



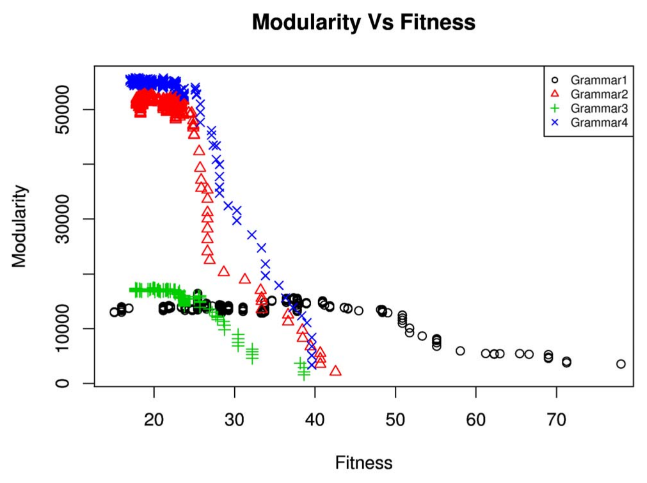
(b) Easy-Medium Target



(c) Medium Target



(d) Medium Hard Target



(e) Hard Target

Fig. 7. Scatter plots showing the correlation between modularity and best fitness

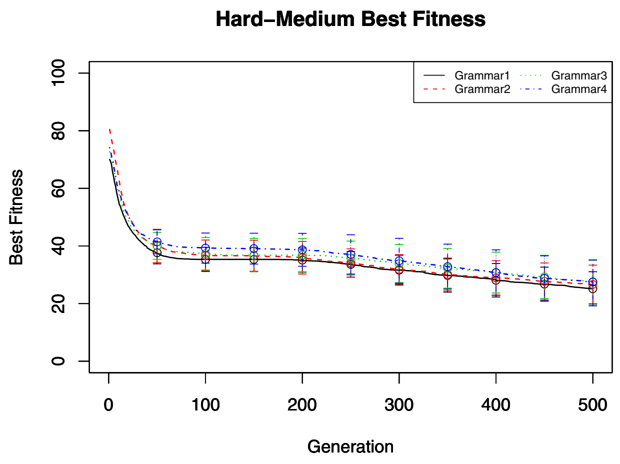


Fig. 8. Best fitness graph for the Hard-Medium problem

derivation tree or group of sub-derivation trees in an individual. Considering this, it is easy to see how modularity values increase to a certain level and vary only slightly from that point. Evolution finds a particular range of modularity values which it seems to think are optimum and remains within that range. The fitness function also contributes to the shape of these plots by punishing and/or rewarding individuals based on the number of modules they have, according to the above definition of modularity. This promotes individuals with a certain modularity value and discourages other individuals that stray to far from this value. A possible remedy for this could be to include some notion of usefulness of a module in the definition of modularity. This would show a more linear correlation between fitness and the modularity value being measured because as more useful modules are discovered, the best fitness would be increasing as well.

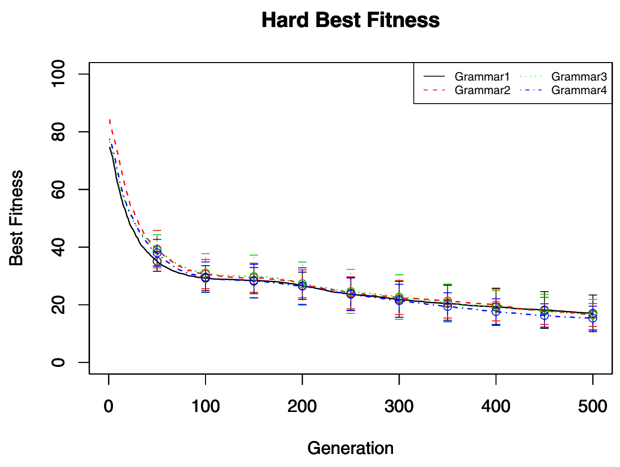


Fig. 9. Best fitness graph for the Hard problem

TABLE II
TOTAL MODULARITY VALUE T-TEST RESULTS

	Gram. 1	Gram. 2	Gram. 3	Gram. 4
	Easy Problem			
Gram. 1	1.0	0.0	$2.6\times^{-07}$	0.0
Gram. 2	0.0	1.0	0.0	6.6×10^{-12}
Gram. 3	2.6×10^{-7}	0.0	1.0	0.0
Gram. 4	0.0	6.6×10^{-12}	0.0	1.0
	Easy-Medium Problem			
Gram. 1	1.0	0.0	$5.0\times^{-44}$	0.0
Gram. 2	0.0	1.0	0.0	3.1×10^{-14}
Gram. 3	5.0×10^{-44}	0.0	1.0	0.0
Gram. 4	0.0	3.1×10^{-14}	0.0	1.0
	Medium Problem			
Gram. 1	1.0	0.0	$7.8\times^{-03}$	0.0
Gram. 2	0.0	1.0	0.0	5.2×10^{-07}
Gram. 3	7.8×10^{-03}	0.0	1.0	0.0
Gram. 4	0.0	5.2×10^{-07}	0.0	1.0
	Hard-Medium Problem			
Gram. 1	1.0	0.0	$8.5\times^{-06}$	0.0
Gram. 2	0.0	1.0	0.0	8.6×10^{-13}
Gram. 3	8.5×10^{-06}	0.0	1.0	0.0
Gram. 4	0.0	8.6×10^{-13}	0.0	1.0
	Hard Problem			
Gram. 1	1.0	0.0	$1.1\times^{-07}$	0.0
Gram. 2	0.0	1.0	0.0	1.7×10^{-13}
Gram. 3	1.1×10^{-07}	0.0	1.0	0.0
Gram. 4	0.0	1.7×10^{-13}	0.0	1.0

V. CONCLUSIONS

This work set out to examine how the capabilities of grammars with varying levels of modularity perform differently when solving design problems of varying difficulty. The first conclusion that can be taken from the results of this work is that there is a correlation between improvements in fitness and increases in the modularity being expressed in individuals. This is plain to see in Figures 7(a) - 7(e). The second and equally important conclusion is that grammars capable of generating more modular individuals do not suffer the same stagnation of neutral evolution that occurs with less modular grammars on sufficiently difficult problems. While there may be little to no change in fitness over the course of a number of generations, the more modular grammars were still searching different solutions with different modularity values. This held even more true as problem difficulty increased, adding even more evidence to the widely known fact that modularity is essential for problem scalability.

This work opens the door to many possibilities for future experimentation. One of the first that comes to mind is the addition of ADFs (previously mentioned in Section II-B). ADFs have been studied with a variety of evolutionary computation algorithms [14] and are at least mentioned in texts outlining the features of genetic programming (GP) approaches [15]. It is widely known that ADFs add much to the power of GP and GP-based algorithms and that they enhance the modularity of the representation. Extending this work by the addition of ADFs would allow for more analysis of the modularity of different representations in grammar-based GP, and GE in particular. Another avenue for exploration after this work is the introduction of “smarter” crossover and mutation operators that operate on derivation trees created by GE’s genotype to phenotype mapping process instead of the genotypic integer arrays. Along these same lines, operators or measures could be implemented to identify useful modules and make them more likely to be preserved through the generations of an evolutionary run.

Acknowledgments

The authors would like to acknowledge the members of the Natural Computing Research and Applications Group for their discussions over the topics addressed in this work, especially James McDermott and Erik Hemberg for their discussions, comments, suggestions, and encouragement. This research is based upon works supported by the Science Foundation Ireland under Grant No. 08/RFP/CMS1115.

References

[1] P. J. Angeline and J. Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 25-26 Feb. 1993.

[2] E. D. de Jong, D. Thierens, and R. A. Watson. Defining modularity, hierarchy, and repetition. In R. Poli, S. Cagnoni, and et al., editors, *GECCO 2004 Workshop Proceedings*, Seattle, Washington, USA, June 2004.

[3] I. Dempsey, M. O'Neill, and A. Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*. Springer, 2009.

[4] O. O. Garibay. *Analyzing the Effects of Modularity on Search Spaces*. PhD thesis, University of Central Florida, 2008.

[5] J. K. Gershenson, G. J. Prasad, and S. Allamneni. Modular product design : A life-cycle view. *J. Integr. Des. Process Sci.*, 3(4):13–26, 1999.

[6] J. H. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, 1975.

[7] G. S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In H.-G. Beyer and U.-M. O. et al., editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1729–1736, Washington DC, USA, 25-29 June 2005. ACM Press.

[8] J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[9] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994.

[10] K. Krawiec and B. Wieloch. Functional modularity for genetic programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 995–1002, New York, NY, USA, 2009. ACM.

[11] H. Lipson, J. B. Pollack, and N. P. Suh. Promoting modularity in evolutionary design. In *Proceedings of DETC'01: 2001 ASME Design Engineering Technical Conferences*, 2001.

[12] M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, and A. Brabazon. GEVA: grammatical evolution in Java. *ACM SIGEVOlution*, 3(2):17–22, 2008.

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

[14] U.-M. O'Reilly. Investigating the generality of automatically defined functions. In *GECCO '96: Proceedings of the First Annual Conference on Genetic Programming*, pages 351–356, Cambridge, MA, USA, 1996. MIT Press.

[15] R. Poli, N. McPhee, and W. Langdon. *A Field Guide to Genetic Programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[16] J. Woodward. Modularity in genetic programming. In *In Genetic Programming, Proceedings of EuroGP 2003*, pages 14–16. Springer-Verlag, 2003.