Measuring, Enabling and Comparing Modularity, Regularity and Hierarchy in Evolutionary Design

Gregory S. Hornby
QSS Group Inc., NASA Ames Research Center, Mailstop 269-3
Moffett Field, CA
hornby@email.arc.nasa.gov

ABSTRACT

For computer-automated design systems to scale to complex designs they must be able to produce designs that exhibit the characteristics of modularity, regularity and hierarchy – characteristics that are found both in man-made and natural designs. Here we claim that these characteristics are enabled by implementing the attributes of combination, control-flow and abstraction in the representation. To support this claim we use an evolutionary algorithm to evolve solutions to different sizes of a table design problem using five different representations, each with different combinations of modularity, regularity and hierarchy enabled and show that the best performance happens when all three of these attributes are enabled. We also define metrics for modularity, regularity and hierarchy in design encodings and demonstrate that high fitness values are achieved with high values of modularity, regularity and hierarchy and that there is a positive correlation between increases in fitness and increases in the measured values of modularity, regularity and hierarchy.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: General

General Terms

Algorithms, Design

Keywords

Representation, Computer-Automated Design, Design, Evolutionary Algorithms, Evolutionary Design, Open-ended design

1. INTRODUCTION

In recent years evolutionary algorithms (EAs) have had increasing success in producing results that are competitive with traditional human design. For EAs to continue to gain wider acceptance and industrial usage the complexity of what can be evolved must continue to increase. Necessary for improving the ability of evolutionary design systems (EDSs) to scale to more complex structures is a better understanding of the types of complexity we are interested in, along with metrics for measuring them, and an understanding of which attributes of an evolutionary design system enable these types of complexity.

In engineering and software development complex artifacts are achieved by exploiting the principles of modularity, regularity, and hierarchy [8] [11] [14], and these characteristics can also be seen in the artifacts of the natural world. Assuming that the principles of modularity, regularity and hierarchy (MR&H) are necessary to achieve scalability then scalable evolutionary design can only be achieved by constructing an EDS capable of producing designs with these characteristics. Breaking down an EDS into its separate modules yields the representation for encoding designs, the algorithm for exploring the space of designs that can be represented, and the fitness function for scoring the goodness of a particular design. Ideally, the ability of an EDS to create hierarchies of reused modules should be independent of how designs are scored. In addition, the algorithm for exploring the space of designs can only find designs that can be expressed by the chosen representation. Thus for an EDS to achieve MR&H it must use a representation capable of encoding designs with these characteristics.

In recent years there has been a handful of research into classes of representations and the different attributes they can have. Angeline classifies representations by whether or not they allow reuse of genotypic elements, and then whether or not the evolved generative system is local to each individual or shared across the population [1]. Bentley and Kumar distinguish between representations which directly encode an object and then distinguish between those that indirectly encode an object implicitly like cellular automata or explicitly like a computer program [2]. Komosinski and Rotaru-Varga list several characteristics of representations for a creature design problem, of which modularity, compression and redundancy are generalizable to other design domains [9]. Stanley and Miikkulainen take five attributes of embryogenies from natural biology as their dimensions for classifying representations – cell fate, targeting, heterochrony, canalization and complexification – but it is difficult to apply these attributes to representations that are not models of developmental biology [13]. In general, these investigations do not explain how different attributes of a representation enable modularity, regularity or hierarchy.

Previously we have argued that the fundamental attributes

GECCO'05, June 25–29, 2005, Washington, DC, USA.
ACM 1-59593-010-8/05/0006.

of design representations are combination, control-flow and abstraction [5, 6, 7]. Here we claim that these attributes enable modularity, regularity and hierarchy and lead to improved evolutionary performance. To support this claim we compare evolutionary performance on different sizes of a table design problem using representations with different attributes enabled and find that better performance is achieved with representations having more features implemented. In addition we define metrics for measuring MR&H in evolved designs and show that in offspring increases in MR&H values are more strongly correlated to increases in fitness than decreases in fitness, and similarly that decreases in MR&H values are more strongly correlated with decreases in fitness than increases in fitness.

2. MODULARITY, REGULARITY AND HIERARCHY

To improve the complexity of what can be evolved with EDSs, we need definitions and metrics for the types of complexity that we are interested in. Here we claim that modularity, regularity and hierarchy are the characteristics of interest and the ability to produce designs with these characteristics is determined by the representation. Consequently, one way to measure the complexity of an evolved object is to measure its modularity, regularity and hierarchy. Before creating metrics for measuring MR&H it is first useful to give an overview what we mean by these terms.

We define modularity as an encapsulated group of elements that can be manipulated as a unit. This form of modularity is related to the building block hypothesis of genetic algorithms (GAs) [4], which states that GAs work by testing groups of basic components and combining them to form highly fit solutions. Modularity also helps enable both regularity and hierarchy. Regularity is a repetition or similarity in a design. Here we focus on the reuse of genotypic elements in creating the phenotype since this form of regularity has been shown to improve scalability of EDSs in two ways: by allowing larger moves through the design space through the manipulation of pre-adapted modules; and by capturing design dependencies into a single location in the genotype thereby improving the ability of variation operators to perform coordinated changes in the design [5, 6]. Hierarchy is the number of layers of encapsulated modules in the structure of a design.

To create metrics of MR&H that generalize across different design domains (graph structures, 3D solid objects, computer programs, . . .) it is useful to define them in terms of the structure of an object's encoding and not of the phenotype. Since representations are a language for describing the phenotype, they have the same fundamental attributes as programming languages: combination; control flow, including conditionals and iteration; and abstraction, consisting of parameters, labeled procedures and the ability to call procedures recursively [5]. Using these attributes we can then show how modularity, regularity and hierarchy can be achieved and can define metrics for measuring them.

Modularity: The modularity value of a design is a count of the number of structural modules in it, which we define as an encapsulated group of genotypic elements that can be manipulated as a unit. Since a label to a procedure can be manipulated as a unit, each procedure in the genotype counts as one toward the genotypic modularity value and in compiling to the phenotype each procedure call counts as one toward the phenotypic modularity value. In addition, the ability to change the iteration counter means that the group of genotypic elements inside an iterative block also constitute a module, hence each iterative block is one genotypic module and each iteration of an iterative block adds one to the phenotypic modularity value. Thus modularity is enabled by *abstraction* and *iteration*.

Regularity: The type of regularity that has been shown to be useful in improving evolvability is a reuse of genotypic elements in creating the phenotype [5, 7]. The reuse in a design is a measure of the average number of times each genotypic element is used in creating the phenotype. It is enabled through either *iteration* or recursion through *abstraction*. Reuse can also be enabled by the `goto`, but we leave this out as a desired feature of representations since it is considered detrimental to good programming.

Hierarchy: The hierarchy of an encoded design is a measure of the number of nested layers of modules, such as through iteration or abstraction. A genotype with no modules has a hierarchy of one. Each nested module, whether a successful call to a labeled procedure or a non-empty iterative block, increases the hierarchy value by one. This is similar to measures of the depth of an object's assembly sequence [3], but whereas there the measure is of basic steps in constructing an object, here we are measuring modules of basic steps. Modules are enabled through *abstraction* and *iteration* and necessary for the creation of nested layers of modules is *combination*.

These measures of design complexity more intuitively measure the structural complexity of an object than the Kolmogorov complexity, which measures the amount of information needed to specify an algorithm [10]. For example the algorithmic information content (AIC) of an algorithmically random bit string, by which is meant one with no regularities, is the number of bits in the smallest algorithm which generates that string. Since the string has no regularities it cannot be compressed so its AIC is the size of the string plus the overhead necessary for the `print` operator. In contrast, in measuring the design complexity of this string by measuring the modularity, regularity and hierarchy values in mapping from the genotype (the uncompressed string is both the genotype and phenotype since it is incompressible) to the phenotype we find its modularity value is 1, since it consists of one module, its regularity value is 1, since there are no reused symbols, and its hierarchy value is 1, since it has only one layer of modules. The values of 1, 1 and 1 for MR&H match our intuition that this random string does not have a complex structure.

3. REPRESENTATIONS

To give an example of the use of our metrics for modularity, regularity and hierarchy we now describe the representational language for GENRE [5], our EDS. The representation used here is a kind of macro-expansion computer language within which design-constructing programs are written. A tree-structure is used for the design programs in which each node in the tree is an operator. Operators can be procedure calls, control-flow operators, or design construction operators. Designs are created by compiling a design program into an assembly procedure of construction operators and then executing this assembly procedure to construct the artifact it encodes.

The representational framework that we use is similar in style to genetic programming (GP) trees with automatically defined functions (ADFs) and also to tree-structured production systems. The following example of a design encoding using this representation consists of two labeled procedures, `Proc_0` and `Proc_1`, each with two parameters:

$Proc\_0(4.0, 2.0)$:

$Proc\_0(n_0, n_1)$:
$n_0 > 4.0 \rightarrow$ rotate-z(1) [Proc_0(1.0,2.0) repeat(2) [forward(n_1/2) [repeat-end [Proc_1(n_0+2.0,2.0) [forward(1)]] [] []]]]
true $\rightarrow$ rotate-z(1) [repeat(4) [rotate-y(1) [forward(n_1+1.0) repeat-end [rotate-x(1)]]] []]

$Proc\_1(n_0, n_1)$:
$n_0 > 1.0 \rightarrow$ forward(2) [Proc_1(1.0,n_1+1.0) [forward(1)] rotate-y(2) [[] Proc_1(1.0,n_1+1.0) [forward(1)]] Proc_1(n_0-2.0,n_1-1.0) [end-proc]]
$n_0 > 0.0 \rightarrow$ rotate-y(1) [[] backward(n_1) [end-proc []]]

This language has the ability to combine operators to form more complex expressions in a tree-structure, has conditionals and iterative blocks, and has labeled procedures that can take parameters as well as be called recursively.

To compile this program an iterative rewriting-like process is used to expand iterative loops and replace procedure names with the encapsulated set of operators to which they point. When the above program is started with the call `Proc_0(4.0,2.0)` it is initially re-written as:

```
rotate-z(1) [ Proc_0(1.0,2.0) repeat(2) [ forward(1)
[ repeat-end [ Proc_1(6.0,2.0) [ forward(1) ] ] []
[] ] ] ]
```

Then, after four more rewriting steps, the following is the final assembly procedure:

```
rotate-z(1) [ rotate-z(1) [ rotate-y(1) [ forward(3)
rotate-y(1) [ forward(3) rotate-y(1) [ forward(3)
rotate-y(1) [ forward(3) rotate-x(1) ] ] ] ] [] ]
forward(1) [ forward(1) [ forward(2) [ rotate-y(1)
[ [] backward(3) [ forward(1) [] ] ] rotate-y(2)
[ [] rotate-y(1) [ [] backward(3) [ forward(1)
[] ] ] ] forward(2) [ rotate-y(1) [ [] backward(2)
[ forward(1) [] ] ] rotate-y(2) [ [] rotate-y(1) [
[] backward(2) [ forward(1) [] ] ] ] forward(2) [
rotate-y(1) [ [] backward(1) [ forward(1) [] ] ]
rotate-y(2) [ [] rotate-y(1) [ [] backward(1) [
forward(1) [] ] ] ] forward(1) ] ] ] [] [] ] [] []
] ]
```

Since there are no more procedures or iterative loops to expand rewriting stops with this last assembly procedure.

A graphical version of this design encoding is shown in figure 1.a along with a sequence of images which show each step of the compilation process, figures 1.a to g. In these images cubes represent labeled procedures and the calls to them, pyramids represent control-flow operators, and construction operators are represented by spheres.

To create designs with this representation the final assembly procedure is interpreted by a design constructor. The example design program uses the following construc-

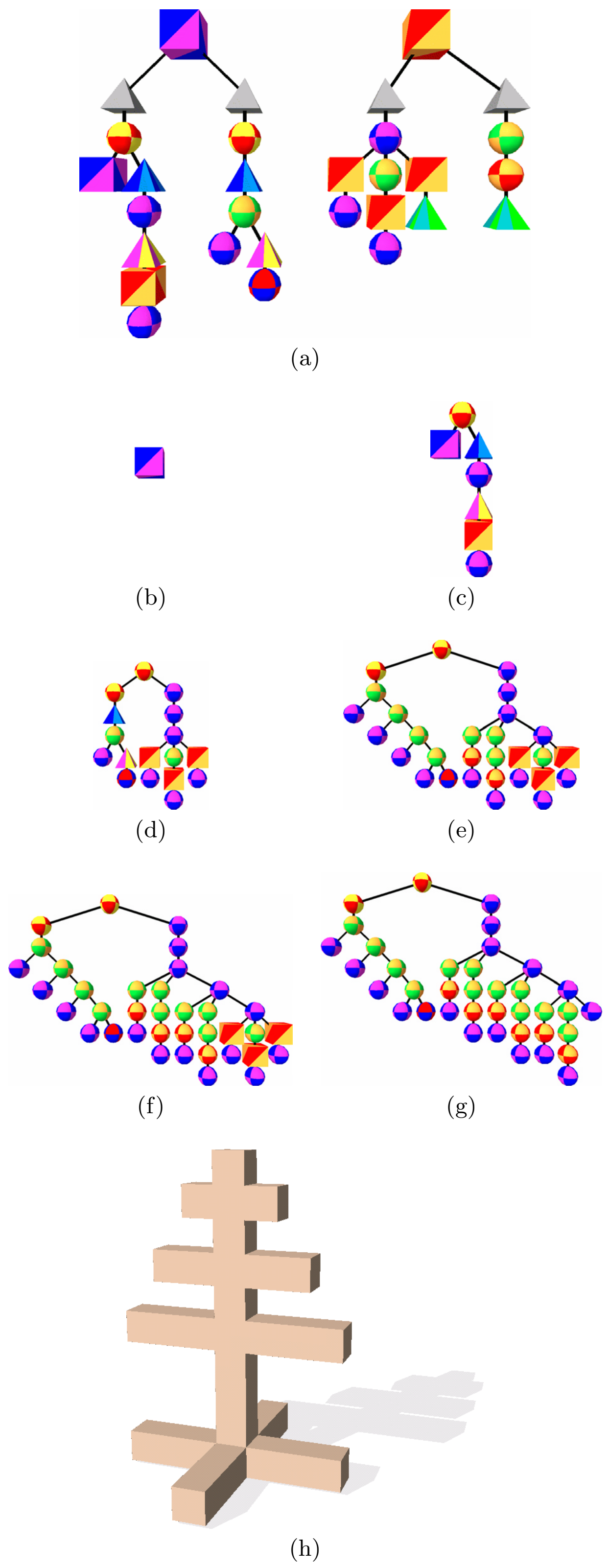


Figure 1: Graphical version of: (a) an example genotype; (b-g) the sequence of assembly procedures produced during its compilation process; and (h) the three-dimensional object that is constructed from the final assembly procedure.

tion operators: `backward`(n), place a sequence of n cubes in the current negative X direction; `forward`(n), place a sequence of n cubes in the current positive X direction; `rotate-x`(n), rotate the current heading $n \times 90^\circ$ about the X axis; `rotate-y`(n), rotate the current heading $n \times 90^\circ$ about the Y axis; and `rotate-z`(n), rotate the current heading $n \times 90^\circ$ about the Z axis.

With this design-construction language a design starts with a single cube in a three-dimensional grid and new cubes are added with the operators `forward()` and `backward()`. The current state, consisting of location and orientation, is maintained with the addition of cubes resulting in a change in the current location and the `rotate-xyz()` operators change the current orientation. A branching in the assembly procedure results in a split in the construction process with construction continuing with each child subtree working with its own copy of the construction state. The solid object that is created by executing this assembly procedure is shown in figure 1.h.

Using the genotype and compilation process the MR&H values of this object can be measured. The genotype has six modules that are used a total of 17 times giving a genotypic modularity value of 6 and a phenotypic modularity value of 17. The size of the genotype is 30 symbols and the size of the final assembly procedure is 38 symbols giving a regularity value of 1.27. In compiling the genotype there are five rewriting passes indicating that there are five levels of nested modules which gives a hierarchy value of 5.

4. EXPERIMENTS

In this section we present experiments to support our claim that modularity, regularity and hierarchy are necessary to improve the scalability of evolutionary design systems and that these characteristics are enabled by different features of design representations. For this we compare evolutionary performance on a design problem in which the objective is to evolve a table out of cubes in a 3D grid environment [6]. The fitness function to score tables is a function of its height, surface structure, stability and number of excess cubes used. Height is the number of cubes above the ground. Surface structure is the number of cubes at the maximum height. Stability is a function of the volume of the table and is calculated by summing the area at each layer of the table. Maximizing height, surface structure and stability typically results in table designs that are solid volumes, thus a measure of excess cubes is used to reward designs that use fewer bricks,

$$\begin{aligned}
f_{height} &= \text{the height of the highest cube, } Y_{max}.\\
f_{surface} &= \text{the number of cubes at } Y_{max}.\\
f_{stability} &= \sum_{y=0}^{Y_{max}} f_{area}(y)\\
f_{area}(y) &= \text{area in the convex hull at height y.}\\
f_{excess} &= \text{number of cubes not on the surface.}
\end{aligned}$$

To produce a single fitness score for a design these five criteria are combined together:

$$\text{fitness} = f_{height} \times f_{surface} \times f_{stability} / f_{excess} \qquad (1)$$

The EA used for all experiments is generational, using a population size of 100 individuals, an elitism of two, and new individuals are created with an equal probability of mutation and recombination. Parents for these individuals are selected with remainder stochastic sampling based on rank, using exponential scaling [12].

In the experiments presented in this section we use five representations with different combinations of MR&H enabled. These combinations are:

None: No features are enabled. In this case there are no forms of control-flow or abstraction and the genotype being evolved is a single procedural rule with a single body in which the condition always succeeds. None of modularity, regularity or hierarchy are enabled with this representation.

M: Labeled procedures are enabled, but not iteration and only the first procedure, `Proc_0`, can call any other procedures. With this representation *modularity* is enabled through abstraction, but reuse is not enabled since there is neither iteration or recursion (the first procedure is not allowed to call itself) and hierarchy is limited to at most two levels. For this representation at most ten procedures can be used, with each procedure having one conditional with a subtree of at most 500 symbols.

MR: Iteration and labeled procedures are used but only the first procedure, `Proc_0`, is able to call other procedures and have iterative loops. Through these features *modularity* and *reuse* are enabled. Hierarchy is limited to two levels by only allowing iterative loops and procedure calls in the first procedure, `Proc_0`, which is not allowed to call itself. As with MH, at most ten procedures can be used with each procedure having three conditionals, since reuse of genotypic symbols is allowed each subtree is limited to at most 25 symbols.

MH: This representation has labeled procedures which can call each other, but no iteration nor can procedures be called more than once. Here *modularity* is enabled through abstraction and *hierarchy* is enabled through the use of nested modules but reuse is not allowed. With this representation at most ten procedures can be used, with each procedure having three conditionals, each with a subtree of at most 166 symbols.

MRH: This representation has all of the features allowed – control-flow, iteration, and labeled procedures with parameters that are able to call themselves recursively – consequently all three of *modularity*, *regularity* and *hierarchy* are enabled. The number and size of the procedures is configured the same as with MR: 10 productions, with 3 conditional subtrees, each with a maximum size of 25 symbols.

The different representations vary in the number of production rules and conditional subtrees they may use. Since generative representations can produce much larger phenotypes than their genotype size, for a fair comparison we set the maximum size of phenotypes for all representations to 5000 symbols and limit the maximum size of genotypes for generative representations to approximately 750 symbols.

In the first set of experiments we compare the five representations with different combinations of MR&H enabled on five different sizes of the table design problem (10x10x10, 20x20x20, ... 50x50x50). Table 1 contains the results of performing thirty trials with each combination of representation and problem size. Each entry in the table shows the average over these thirty trials of the best individual found after 4000 generations using a population size of 100 individuals. These results show that generally the best performance is achieved in a representation with more of the features of

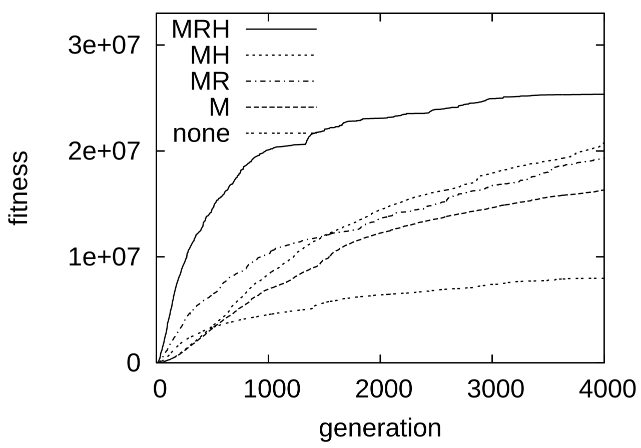


Figure 2: A comparison of evolution with different combinations of MR&H enabled on the 50x50x50 table problem. Each curve is the average of thirty trials.

MR&H enabled. Performance with just two features, MR and MH, is comparable on the larger three design spaces, on which both outperform search with just M enabled.

	10	20	30	40	50
none	9388	226124	1229000	3732797	7985324
M	9108	294458	1909880	5799474	16314567
MR	10732	325120	2007445	6367398	19309882
MH	8826	274245	1993205	8070569	20751354
MRH	12406	341586	1953056	9269713	25352851

Table 1: Performance results of the different representations on different sizes of the table problem. Each entry is the average of thirty trials of the best result after 400000 evaluations.

It is worth examining the performance of the different representations over the course of a run. Figure 2 shows the averaged-best performance over thirty trials of the different representations on the 50x50x50 table design problem. From this graph it can be seen that evolution with MR enabled increases in fitness rapidly at the beginning but is then overtaken by MH. Also, evolution with MR&H enabled in the representation increases very rapidly in fitness for the first 1000 generations, but then levels off more than the other representations. This suggests that reuse of genotypic elements is very helpful to rapidly get good solutions but it may be hard to later specialize a module. Interestingly, the representation with none of MR&H enabled initially outperforms both M and MH, but then is overtaken after 400 generations. One explanation for this is that it takes evolution some time to evolve a good decomposition of the problem into modules before it can take advantage of them.

Next we demonstrate that increased modularity, regularity and hierarchy go hand in hand with higher fitness. For this experiment we performed 25 trials using the MR&H representation with a population of 100 individuals evolved for 2500 generations on the 40x40x40 table design problem.

The graphs in figure 3 are scatter plots of fitness plotted against the different complexity measures defined in section 2 along with plots of the genotype and assembly procedure size and the number of modules in the genotype. In all cases higher fitness corresponds with greater modularity, regularity and hierarchy. With one or two modules, fitness does not go beyond 100 and the highest fitness values require ten or more modules in the genotype and 100 or more modules in the actual design. With an average reuse of 1.5 or less the best fitness is less than 1000 and then there is a large jump in fitness to over one million when reuse goes from 1.5 to 2.0 and another jump to over 7 million when reuse goes from 4 to 7. Similarly, with a hierarchy of 2 or less the best fitness achieved is under 500, but this jumps to almost one million for hierarchy values of 3 and then reaches the maximum fitness for hierarchy values of 7. In addition, using genotype length as an upper-bound estimate of AIC (figure 3.a), these plots show that better designs come from individuals which contain more information While higher complexity corresponds to higher fitness, these plots also show that the increase in fitness plateaus after a certain level of modularity, regularity and hierarchy are achieved.

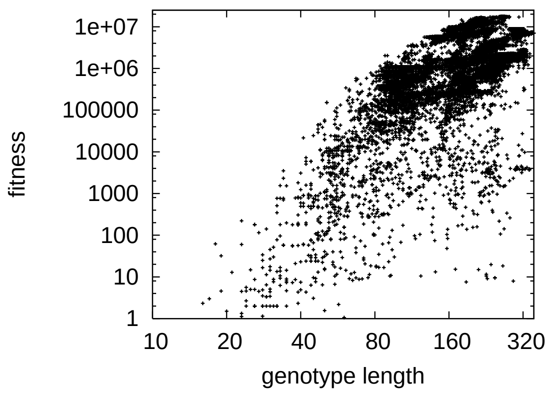
Finally, the importance of MR&H is demonstrated by examining the correlation between decreases/increases in fitness and decreases/increases in MR&H values. For offspring with different fitness than their parents, table 2 lists the percentage of times a fitness decrease/increase occurred along with a decrease/increase in genotype size, phenotype size, genotypic modularity, phenotypic modularity, hierarchy and reuse. Of the more than 5 million individuals evaluated, 2618943 individuals had lower fitness than their parents and 66843 individuals had higher fitness than their parents. For all six measures of complexity, a decrease in fitness was far more likely to occur with a decrease in MR&H value than with an increase in MR&H value. Similarly, but to a lesser degree, an increase in fitness was more likely to occur together with an increase in MR&H value than with a decrease in MR&H value. This indicates that better fitness is achieved through increasing MR&H.

	fitness decrease		fitness increase	
	val decr	val incr	val decr	val incr
geno size	55.12%	16.45%	26.55%	42.99%
pheno size	57.02%	16.50%	26.72%	44.35%
geno mod	23.24%	3.71%	6.75%	9.85%
pheno mod	32.21%	5.76%	9.19%	16.74%
hierarchy	16.76%	0.89%	3.17%	5.52%
reuse	47.82%	24.83%	30.51%	38.37%

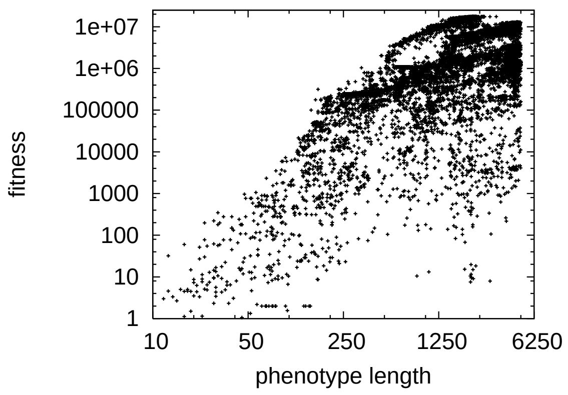
Table 2: For decreases/increases in fitness this table lists the percentage of times there was a corresponding decrease/increase in the measured value.

5. DISCUSSION

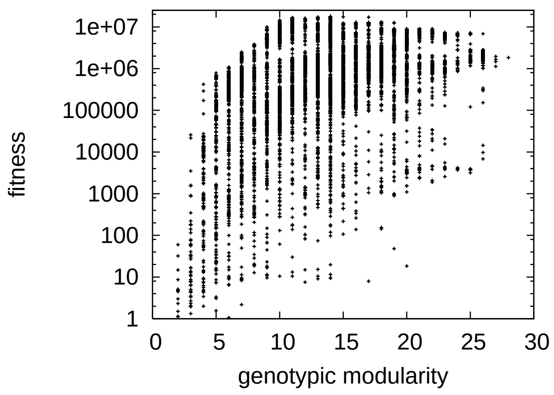
An intuitive understanding of the differences between evolution with MR&H enabled and not enabled can be gained by viewing the structure of the evolved genotypes and phenotypes. The best table evolved without any of MR&H enabled is shown in figure 4.a. It has randomly scattered holes on its surface and none of the legs on its corners go down more than a few levels. Since none of the features of the representation are used, its modularity, regularity and hierarchy values are all 1.0. Its lack of structural organization



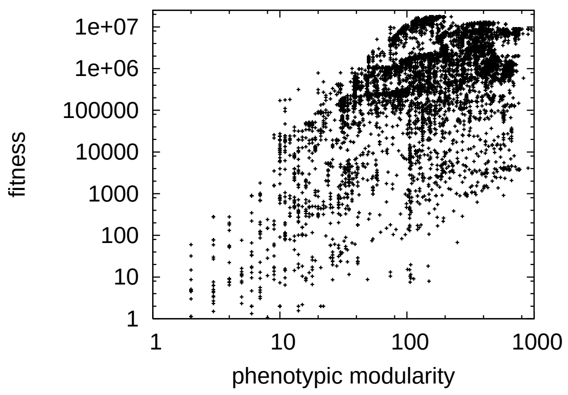
(a)



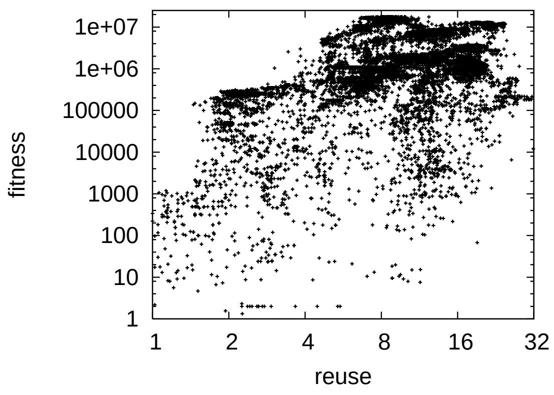
(b)



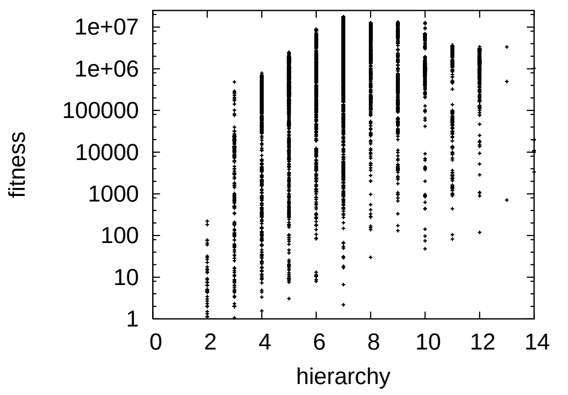
(c)



(d)



(e)



(f)

Figure 3: Measured values of: (a) genotype size; (b) assembly procedure size; (c) number of modules in the genotype; (d) modularity value; (e) reuse; (f) hierarchy value.

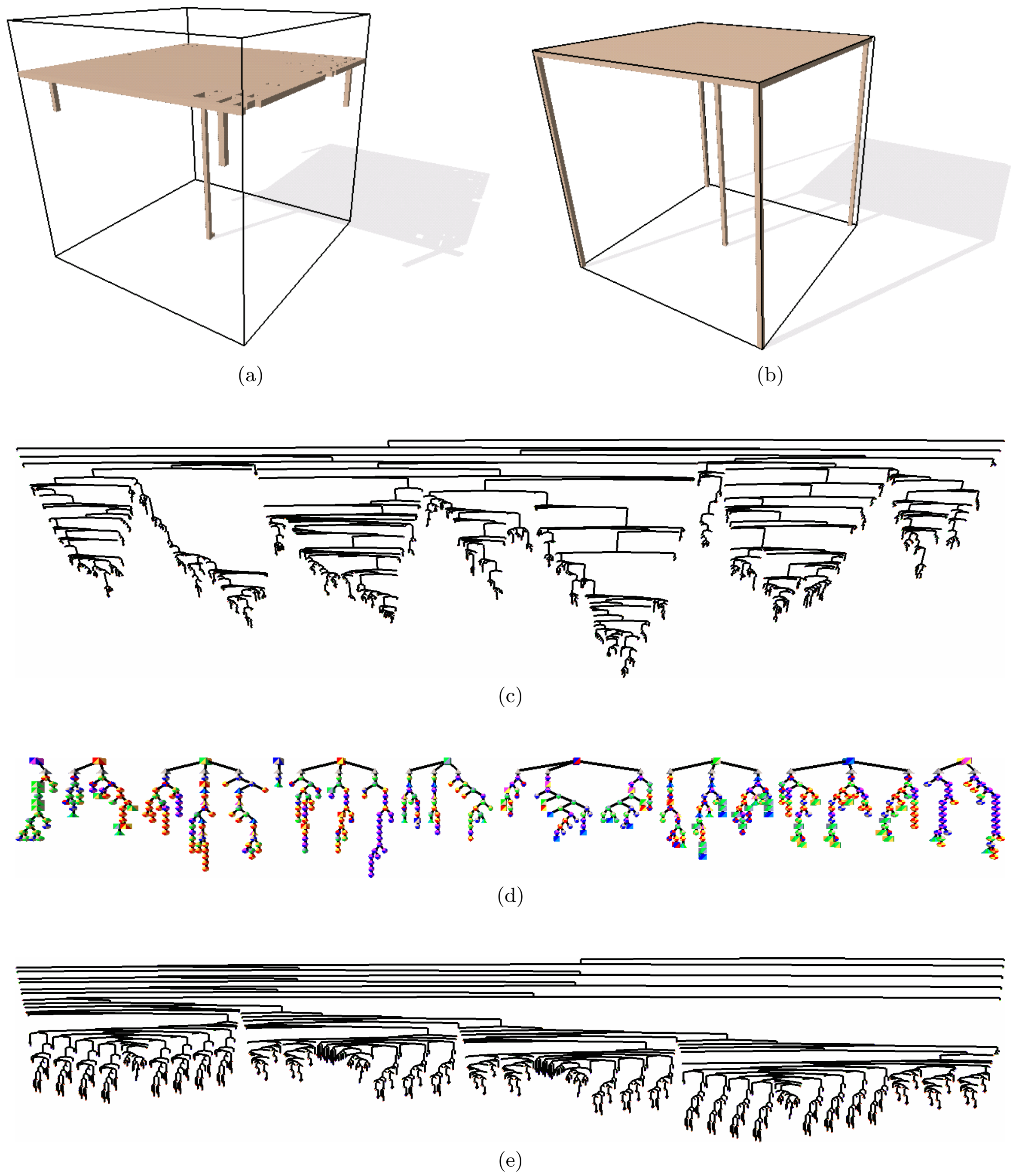


Figure 4: Results from the experiments on the 50x50x50 table design problem: (a) the best table evolved with none of MR&H enabled has a fitness of 25116510 (the black lines mark the limits of the 50x50x50 design space); its genotype, which is the same as the assembly procedure for constructing it, is shown in (c) and has 4999 operators; (b) the best table evolved with MR&H all enabled has a fitness of 60098951; its genotype is shown in (d) and has 495 operators; its resulting assembly procedure is shown in (e) and has 4871 operators.

can be seen in the randomness of its assembly procedure, which is the same as its genotype and is shown in figure 4.c. In contrast the table evolved with all of MR&H enabled, figure 4.b, has straight legs going all the way from top to bottom and a fully filled surface. It has a genotypic modularity of 34, a phenotypic modularity of 1298 (the 34 modules in the genotype are collectively used a total of 1298 times), a hierarchy of 9 and reuse of 10. Its genotype is shown in figure 4.d and the structural complexity it produces can be seen in the complex, multi-level patterns in the assembly procedure it generates, figure 4.e.

From the different representational features – combination, control-flow and abstraction – used in the different representations additional conclusions can be drawn from these experiments. The representations MR and MH performed similarly in these experiments and the difference in features between them is that MR does not have abstraction but instead has iteration, whereas MH has abstraction and hierarchically nested procedures but no iteration or recursion. This suggests that adding iteration to a representation can compensate for not having abstraction (such as ADFs). Also of significance is the large performance difference between MH and MRH. The difference in features between MH and MRH is that MH does not have iteration or the ability to call procedures recursively whereas both of these are possible with MRH. One explanation is that ADFs are more useful if they can be called recursively, alternatively the difference in performance can be attributed to the addition of iterative loops.

6. CONCLUSION

In this paper we have argued that to improve the scalability of evolutionary design systems the types of complexity that need to be enabled are modularity, regularity and hierarchy (MR&H) – characteristics found in both man-made and natural designs. Furthermore, these three characteristics are enabled not by the fitness function or the search algorithm but by attributes of the representational language of the genotype. Here we borrowed from the field of computer programming languages to identify three attributes of design representations – combination, control-flow and abstraction – and claimed that these attributes enable MR&H in evolved designs and used them to define metrics of MR&H.

To support our arguments that MR&H are the design characteristics that must be enabled to improve scalability we ran experiments comparing five variations of a representation with different combinations of representational attributes enabled on five sizes of a 3D table design problem. In all, more than 300 million tables were generated and evaluated in these experiments. The results showed that search using representations with more of these attributes implemented generally achieved better performance. Further support for our claim that enabling MR&H improves scalability came in comparing the correlation between increases and decreases in fitness and in measured values of MR&H. There we found a much stronger correlation when both fitness and measured complexity value increased/decreased at the same time than when there was an increase in one and a decrease in the other.

In the future we expect that more improvements in the scalability of EDSs will come through further inspiration from the field of programming languages, such as with objects and object oriented programming. By implementing increasingly more powerful methods to hierarchically encode reusable modules future EDSs will be better able to produce ever more complex and interesting designs.

7. REFERENCES

[1] P. J. Angeline. Morphogenic evolutionary computations: Introduction, issues and examples. In J. McDonnell, B. Reynolds, and D. Fogel, editors, *Proc. of the Fourth Annual Conf. on Evolutionary Programming*, pages 387–401. MIT Press, 1995.

[2] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies of an evolutionary design problem. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Genetic and Evolutionary Computation Conference*, pages 35–43. Morgan Kaufmann, 1999.

[3] M. Goldwasser, J. Latombe, and R. Motwani. Complexity measures for assembly sequences. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 1581–1587, Minneapolis, MN, Apr. 1996.

[4] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[5] G. S. Hornby. *Generative Representations for Evolutionary Design Automation*. PhD thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA, 2003.

[6] G. S. Hornby. Functional scalability through generative representations: the evolution of table designs. *Environment and Planning B: Planning and Design*, 31(4):569–587, July 2004.

[7] G. S. Hornby and J. B. Pollack. Creating high-level components with a generative representation for body-brain evolution. *Artificial Life*, 8(3):223–246, 2002.

[8] C. C. Huang and A. Kusiak. Modularity in design of products and systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 28(1):66–77, 1998.

[9] M. Komosinski and A. Rotaru-Varga. Comparison of different genotype encodings for simulated 3d agents. *Artificial Life*, 7(4):395–418, 2001.

[10] M. Li and P. M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, Berlin, 1993.

[11] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[12] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York, 1992.

[13] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.

[14] K. Ulrich and K. Tung. Fundamentals of product modularity. *Issues in Design/Manufacture Integration - 1991 American Society of Mechanical Engineers, Design Engineering Division (Publication) DE*, 39:73–79, 1991.