Toward the Computer-Automated Design of Sophisticated Systems by Enabling Structural Organization

Gregory S. Hornby
University of California Santa Cruz
NASA Ames Research Center, Mailstop 269-3
Moffett Field, CA
hornby@email.arc.nasa.gov

ABSTRACT

Since computer-automated design systems have been been shown to be human-competitive in the design of single parts, of interested is scaling them up to producing more sophisticated systems. Here we argue that for computer-automated design systems to scale to design more sophisticated systems they must be able to produce designs with greater structural organization. By structural organization we mean the characteristics of modularity, regularity and hierarchy (MR&H), characteristics that are found both in man-made and natural designs. We claim that these characteristics are enabled by implementing the attributes of combination, control-flow and abstraction in the representation. To defend this claim we define metrics for measuring the three components of structural organization – modularity, regularity and hierarchy – and then 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 are enabled. We find that the best designs are achieved when all three of these attributes are present, thereby supporting our claim. Finally, we demonstrate the value of our metrics by comparing them against existing complexity measures and show that our metrics better correlate with good designs than do the other metrics.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design

Keywords

Representation, Computer-Automated Design, Design, Complexity, Structural Organization, Evolutionary Algorithms, Evolutionary Design, Open-ended design

CSE'07, Jan. 11–12, 2007, Santa Monica, CA, USA.

1. INTRODUCTION

As computers become more powerful, software design tools are becoming increasingly more powerful tools which enable architects and designers to express their ideas. In addition, the use of artificial intelligence techniques, such as evolutionary algorithms, has enabled these software packages to assist in the design process themselves. Already automated design systems based on evolutionary algorithms have been used for antennas [12], flywheels, load cells [23], trusses [21], robots [19], and more [3, 4]. These techniques have been shown to be useful in industry, and not just in academic research, for example the dynamic gait in Sony's entertainment robot, AIBO, of which tens of thousands of robots were sold world wide, was evolved using an evolutionary algorithm [14] (Figure 1(a)) and the X-band antenna which was used successfully on NASA's Space Technology 5 mission was produced with an evolutionary algorithm [12] (Figure 1(b)). While both of these examples greatly outperformed their human-designed counterparts, of interest is scaling these computer automated-design (CAD) techniques from producing simple parts of larger systems to automatically producing designs for entire complex engineering systems.

(a)

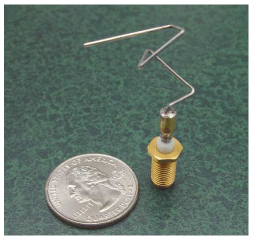
(b)

Figure 1: Two examples of computer-automated design techniques used in industry: (a) dynamic gaits were evolved for Sony's AIBO; and (b) the X-band antenna evolved for NASA's ST5 Mission.

For CAD systems to scale from evolving simple subsystems to producing complex systems with hierarchies of subsystems, the sophistication of what can be produced must continue to increase. Necessary for improving the ability of computer-automated design (CAD) systems to scale to more sophisticated structures is a better understanding of the types of design characteristics that are necessary for improving scalability, metrics for measuring them, and an understanding of which attributes of a computer-automated design system enable these types of characteristics.

Already various metrics exist for measuring what has been loosely defined as *complexity*, such as Algorithmic Information Content (AIC) [5, 16, 25], Logical Depth [2], and Sophistication [17]. These metrics vary in their degree of intuitiveness in measuring complexity, with definite cases in which their results are counter-intuitive. For example, the AIC of a random string will score higher than a string of the same length with hierarchies of regularities, whereas we are inclined to think that a string with the patterns is more complex. More importantly, none of them lead to a path for developing CAD systems that can scale in complexity. A more useful approach may be to toss aside these measures, look to existing design systems that produce sophisticated designs and develop new metrics that explicity measure those characteristics that we are after.

In engineering and software development sophisticated artifacts are achieved by exploiting the principles of modularity, regularity, and hierarchy [15, 20, 26], 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 by constructing a CAD system capable of producing designs with these characteristics we can hope to achieve more scalable computer-automated design. Breaking down a CAD system into its separate modules yields the representation for encoding designs, the search 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 a CAD system to create hierarchies of reused modules should be independent of how designs are scored. In addition, the search algorithm for exploring the space of designs can only find designs that can be expressed by the chosen representation. Thus for a CAD system to achieve MR&H it must use a representation capable of encoding designs with these characteristics.

To be able to develop representations which can encode designs with MR&H we need to understand the fundamental attributes of design representations. One way to analyze representations is to consider them as a kind of programming language and, using this metaphor, the properties of programming languages can be used to formally classify representations. From [1], the fundamental properties of programming languages are: combination, the ability to hierarchically create more powerful expressions from simpler ones; abstraction, the ability to name compound elements and manipulate them as units and formal parameters to procedures; and control-flow, such as conditional expressions and iterative constructs (although not necessary in languages such as Scheme, it is convenient in ones such as Pascal and C++).

Thus for CAD systems to scale they need better representations, by which we mean representations which can encode for structural organization (modularity, regularity and hierarchy). Encoding MR&H is enabled by using representations with the attributes of combination, control flow and abstraction. Measuring the degree of structural organization in a design – a more useful metric than those measuring "complexity" – is best done by explicitly measuring the amount of MR&H in a design.

To support our claims we present results on a design problem using an evolutionary algorithm (EA) and a handful of different representations. Each representation has a different combination of the characteristics of MR&H enabled through the use of different combinations of combination, control-flow and abstraction. Results of our experiments show that enabling more of modularity, regularity and hierarchy lead to better performance with our EA. In our runs we also measure the amount of MR&H that occurs in each evolved design and compare these scores against various complexity measures. Of these metrics, our metrics for MR&H are better correlated with the fitness of good designs than are the others.

2. PROPERTIES OF REPRESENTATIONS

Representations for computer-automated design can be divided into *parameterizations* and *open-ended* representations. With parameterizations, the topology of the design is prespecified and the search algorithm is limited to performing numerical optimization on the set of parameters. In contrast, open-ended representations enable the CAD system to search through arbitrary topologies thereby allowing novel types of designs to be discovered. Since we are interested in being able to automatically produce novel solutions to complex engineering problems, and not merely optimizing the parameters of existing designs, we focus on open-ended representations.

Necessary for improving the power of open-ended design representations is knowing their fundamental attributes. Because the mapping from an encoded design to an actual artifact can be considered a computational process, open-ended representations can be thought of as *programming languages*, with encoded designs being *programs* in this language. With this analogy, features of programming languages can be used to understand and classify different approaches to the underlying representations of CAD systems.

Based on work in programming languages [1], we have claimed [9, 10, 13] that the fundamental properties of open-ended representations are:

- **Combination**: The ability to hierarchically create more powerful expressions from simpler ones. While the subroutines of genetic programming (GP [18]) allow explicit combinations of expressions, combination is not fully enabled by mere adjacency or proximity in the strings utilized by typical representations in genetic algorithms.
- **Control-flow**: All programming languages have some form of control of execution, which permits the conditional and repetitive use of structures. Two types of control-flow are conditionals and iterative expressions. Conditionals can be implemented with an `if`-statement, as in GP, or a rule which governs the next state in a cellular automata. Iteration is a looping ability, such as with the `for-next` loop in the *C* programming language.

- **Abstraction**: This consists of the ability to encapsulate a group of expressions in the language, enabling them to be manipulated/referenced as a unit, and the ability to pass parameters to procedures.

In implementation, these elements can be parceled out to different mechanisms, such as branching, variables, bindings, recursive calls, but are nonetheless present in some form in all programmable systems.

3. ENABLING MODULARITY, REGULARITY AND HIERARCHY

While various metrics already exist for measuring complexity, none of them provide useful guidance as to how to engineer better CAD systems that produce designs for more complex systems. To improve the sophistication of what can be evolved with CAD systems, we need definitions and metrics for the types of characteristics of the types of complexity that are useful for improving scalability. Here we claim that the ability to produce designs with good structural organization, that is designs with the characteristics of modularity, regularity and hierachy is the way to improving the scalability of CAD systems. Consequently, the useful way of measuring complexity of a design is through metrics of MR&H. Before creating metrics for measuring MR&H it is first useful to give an overview of 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) [8], 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 elements in the encoding of a design since this form of regularity has been shown to improve scalability of CAD systems 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 encoding thereby improving the ability of variation operators to perform coordinated changes in the design [9, 10]. *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 an object's encoding and not of the design itself. Since representations are a language for describing designs, 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 [9]. 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 elements in the design encoding that can be manipulated as a unit. Since a label to a procedure can be manipulated as a unit, each procedure in the design encoding counts as one toward the encoded modularity value. In addition, the ability to change the iteration counter means that the group of encoded elements inside an iterative block also constitute a module, hence each iterative block is one module in the encoding. Thus modularity is enabled by *abstraction* and *iteration*. As well as counting modules in the encoded design we can also count the number of occurrences of modules from the encoding in the design itself. In this case each procedure call counts as one toward the design modularity value and each iteration of an iterative block adds one to the modularity value of the design.

Regularity: The type of regularity that has been shown to be useful in improving evolvability is a reuse of encoding elements in creating the design [9, 13]. The reuse in a design is a measure of the average number of times each element in the encoding is used in creating the design. 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 design encoding 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 measuring the depth of an object's assembly sequence [7], 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*.

In the rest of this paper we use `MRH` to refer to these metrics of structural organization and MR&H to refer to the characteristics of modularity, regularity and hierarchy.

4. REPRESENTATIONS

To give an example of the use of our metrics for modularity, regularity and hierarchy we now describe the representational language for GENRE [9], the CAD system we use for conducting our experiments. 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 generate the artifact.

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) re-
                  peat(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 sophisticated 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 2.a along with a sequence of images which show each step of the compilation process, figures 2.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 construction operators: `backward`(n), place a sequence of n cubes in the current negative X direction; `forward`(n), place a sequence

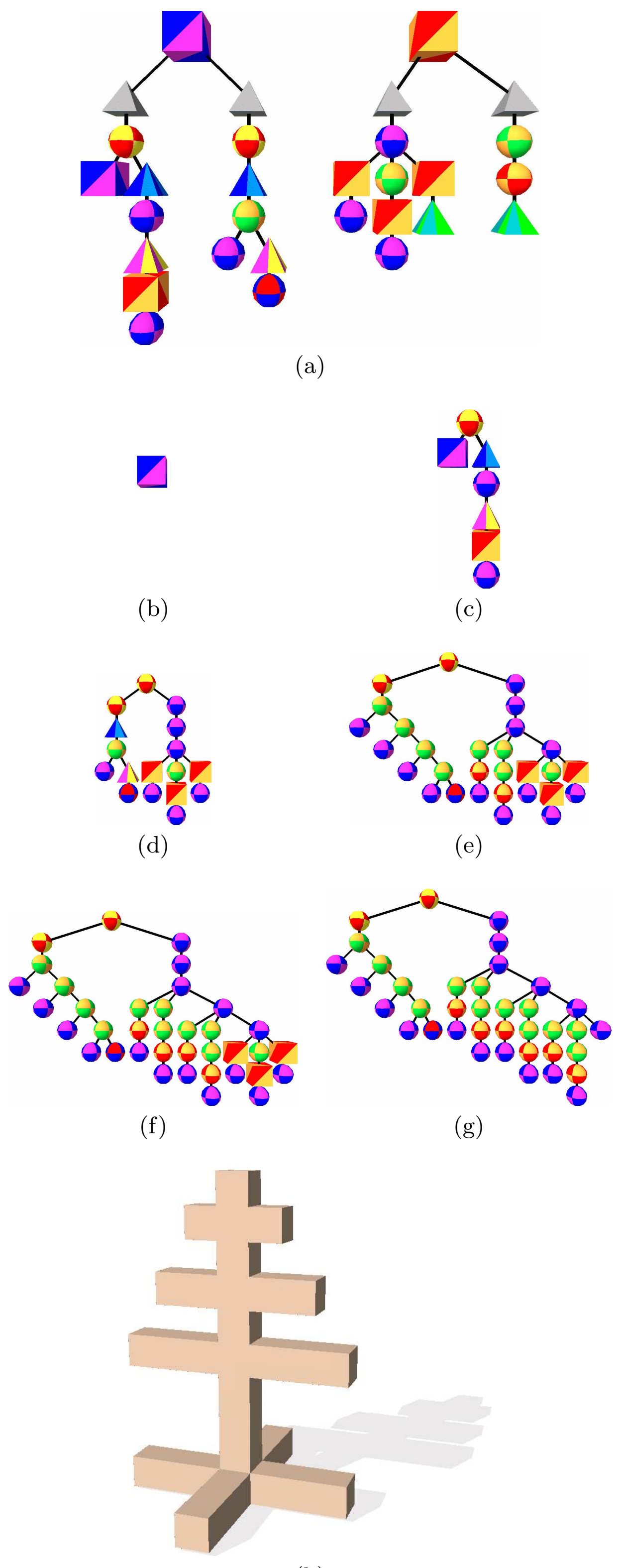

Figure 2: Graphical version of: (a) an example encoding (program); (b-g) the sequence of assembly procedures produced during its compilation process; and (h) the three-dimensional object that is constructed from the final assembly procedure.

of n cubes in the current positive X direction; `rotate-x`(n), rotate the current heading by $n \times 90^\circ$ about the X axis; `rotate-y`(n), rotate the current heading by $n \times 90^\circ$ about the Y axis; and `rotate-z`(n), rotate the current heading by $n \times 90^\circ$ 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 2.h.

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

5. OTHER COMPLEXITY METRICS

To demonstrate that the `MRH` metrics of structural organization are meaningful we compare them against complexity metrics that have been proposed. In developing methods for calculating the values of these complexity metrics, one consideration is that in our case the programs we are evolving are tree-structured as opposed to strings of symbols. In addition to complexity metrics that have been proposed in the literature as well as the `MRH` metrics of structural organization we are advocating, we include some additional metrics of "complexity" to serve as controls. We now review the different complexity metrics that we will use as well as the methods by which we will either calculate, or estimate, their values.

Algorithmic Information Content (AIC) is one of most well known and influential complexity metrics, having been used as a starting point for many others, and was invented separately by Chaitin [5], Kolmogorov [16], and Solomonoff [25]. The AIC of a given string is the length, in number of symbols, of the shortest program that produces that string. For this work we estimate the AIC by calculating the number of symbols in the design encoding, since this is the evolved program that is then compiled into the intermediate assembly procedure that is then executed to produce the 3D voxel structure. While it is likely that some of the programs could be compressed, using the size of the program is a simple upper bound on AIC, most long strings are incompressible [5], and our method is a correct measure of the size of the program that was evolved.

Assembly procedure size is the number of symbols in the assembly procedure that the program compiles to. That is, this measure is the *size* of the object. It is this assembly procedure that is then sent to the table-building module to build a 3D voxel structure and it is this tree of symbols that the other complexity metrics use to calculate a "complexity" value.

Logical Depth is a measure of the value of information and for a given string it is the minimum running time of a near-incompressible program that produces the string [2]. In this case we use the evolved program as the near-incompressible program and calculate the running time of this program as the number of symbols that are processed in generating the assembly procedure. This can also be considered computation complexity, in that it is a measure of the amount of computational time that is spent to compute the assembly procedure.

Sophistication this is a measure of the structure of a string by counting the number of *control* symbols in the program used to generate it [17]. In trying to measure the structure of a string, the goal for this measure is the same as the goal for the MRH metrics of complexity. Here we calculate the sophistication of a tree-structured assembly procedure by counting the number of control symbols – that is, procedure symbols, loop symbols, conditionals – in the program that is used to generate it.

Number of Build Symbols, whereas Sophistication is a measure of structure by counting the number of control symbols, we propose a counter measure which is a count of the number of non-control symbols in the program that is used to generate the assembly procedure. In our system, these non-control symbols are the operators that are used by the voxel-constructing interpreter and we call them *build* symbols, since they are used to build a 3D shape.

Grammar Size: any string that has a pattern can be expressed as being generated by a grammar. Simple strings, with simple patterns, generally have a simple grammar thus the size of the grammar needed to produce a string serves as a measure of complexity [6]. In fact, the representation used here comes out of early work at implementing parametric Lindenmayer systems [22], and while it is now more like Genetic Programming [18] the procedures can be thought of as grammar rules. To calculate the grammar size of an assembly procedure we use the program that produces it as the grammar and count the number of procedures / production-rules in the program.

Connectivity: more complex systems have greater interconnectedness between components, thus the connectivity of a system can be used as a complexity measure [6]. For a graph-structure, its connectivity is the maximum number of edges that can be removed before it is split into two non-connected graphs. To calculate the connectivity of our assembly procedure we use the connectivity of the program that is used to generate it instead and since this program is a forest of tree-structured data-structures, we estimate the connectivity of it by counting the number of loops and procedure calls.

Height: is the maximum number of edges that can be traversed in going from the root of the tree to a leaf node. Unlike other complexity metrics, which are based on strings, this measure is for trees. This measure of complexity is related to work in formal language theory in which ideas for measuring ease of comprehension are to measure the depth of postponed symbols [27] or depth and nesting [24].

Number of Branches: inspired by the previous measure of complexity, another measure of the structure of a graph is a count of number of nodes which are branch nodes – nodes which have two or more children. Strings have a very simple structure with no branching nodes, whereas a fully balanced binary tree will have roughly *lg(n)* branch nodes.

The metrics for **modularity**, **regularity** and **hierarchy** have been covered already in section 3. Here we give three metrics of regularity: overall reuse, reuse of build symbols, and modular reuse. Overall reuse, $\mathtt{R}$, which is measured by dividing the size of the assembly procedure by the size of the program. Reuse of *build* symbols, $\mathtt{R}_b$, is a measure of the average reuse of build symbols (non-control symbols) and is calculated by dividing the number of build symbols in the assembly procedure by the number of build symbols in the program. Modular reuse, $\mathtt{R}_m$, which is a measure of the average use of modules in the assembly procedure and is calculated by dividing the number of times modules are used in creating the assembly procedure by the number of modules in the program.

In section 4 we presented an example design and its corresponding encoding (program) along with its `MRH` scores (6:1.27:5) we now give its corresponding complexity scores. This design encoding has: an AIC of 30; an assembly procedure size of 38; a Logical Depth of 78; a Sophistication of 21; 13 build symbols; a grammar size of 2; a connectivity of 5; 8 branches; and a height of 10. In addition, the calculated values of modularity (in the program) is 6, regularity of all symbols is 1.27, regularity of build symbols is 2.92 regularity of modules is 2.83 and hierarchy is 5.

6. MR&H IMPROVES SEARCH

To demonstrate the usefulness of our `MRH` metrics and show that enabling MR&H improves the scalability of CAD systems we compare evolutionary design using various representations with different combinations of MR&H enabled and compare performance on different sizes of a design problem. The design problem we use is to produce a table out of voxels (cubes) in a 3D grid environment [10], for which the fitness function for scoring tables is a function of their 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 a modification of the standard generational EA called the Age-Layered Population Structure (ALPS) EA. This EA maintains several layers of individuals of different ages and continuously introduces new, randomly generated individuals into the first layer. It has been shown to work better than the canonical EA by better avoiding premature convergence [11]. The setup we use consists of 10 age layers, each with 40 individuals. In each layer the best 2 individuals from the previous generation are copied to the current generation and then new individuals are created with a 40% chance of mutation and 60% chance of recombination using tournament selection with a tournament size of 5.

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 program 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 and the body has an upper limit of 10000 symbols.

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 25 procedures can be used, with each procedure having three conditionals, and of those subtrees having a maximum size 1000 symbols.

MH: This representation has labeled procedures but no iteration. The labeled procedures can call each other, but a procedure can be called at most 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 25 procedures can be used, with each procedure having 3 conditionals, each with a subtree of at most 1000 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 25 procedures can be used with each procedure having three conditionals and again each of these three subtrees has a maximum size of 1000 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: 25 productions, with 3 conditional subtrees, each with a maximum size of 1000 symbols.

In our experiments we compare the five representations with different combinations of MR&H enabled on five different sizes of the table design problem (10x10x10, 20x20x20 and 40x40x40). Table 1 contains the results of performing 15 trials with each combination of representation and problem size. Each entry in the table shows the average over these 15 trials of the best individual found after one million evaluations. These results show that the best performance is achieved in a representation with more of the features of MR&H enabled: representation MR&H is always best; representation MR is always second best; and representation M always outperforms *none*. These results support our claim that enabling MR&H improves the performance of CAD systems.

	10	20	40
none	9388	226124	4822889
M	9108	294458	6780158
MH	8826	274245	7817185
MR	10732	325120	14604560
MRH	12406	341586	18003769

Table 1: Fitness scores of the different representations on different sizes of the table problem. Each entry is the average of 15 trials of the best result after 1000000 evaluations.

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 M and MH performed similarly in these experiments and the difference between them is only that M allows for only a single layer of nested modules whereas MH can have an arbitrary number. In both cases adding Reuse (going from M to MR and MH to MRH) results in large performance improvement. Also of interest is the large amount by which MR outperforms MH, which suggests that reuse is more important than hierarchical modules. Finally, representation MH goes from being the worst on the 10x10x10 size problem, to inbetween *none* and M on the 20x20x20 size problem to outperforming both on the 40x40x40 sized problem. This suggests that as the problem size increases, hierarchy becomes a more useful property to have enabled.

7. COMPARING METRICS

To show that our `MRH` metrics for structural organization are useful, we compare them against various measures of structure and complexity. Table 2 lists the average score for each of the metrics as applied to the best tables evolved with the representations of section 6.

	none	M	MH	MR	MRH
Fitness ($\times 10^6$)	4.82	6.78	7.82	14.60	18.00
AIC	9250	1787	2450	2228	659
Assem. Size	9250	1787	2450	7142	8174
Log. Depth	9252	1809	2497	7334	20585
Sophistication	1	29.5	52.6	37.4	96.6
# Bld Sym	9250	1787	2450	2197	574
Grammar Size	1	6.9	9.5	4.6	8.8
Connectivity	0	0.2	7.2	11.1	29.8
Height	243	114	146	138	160
# Branches	2075	400	526	1509	1642
Mod. in Program	1	7.1	11.3	9.7	32.4
Mod. in Design	1	7.1	11.3	22.9	717
Reuse	1	1	1	3.6	13.9
Reuse_b	1	1	1	3.7	15.9
Reuse_m	1	1	1	2.3	24.0
Hierarchy	1	2	5.8	2	6.2
`MRH`	1.73	7.4	12.8	10.8	37.2
$\texttt{MR}_b\texttt{H}$	1.73	7.4	12.8	10.8	38.2
$\texttt{MR}_m\texttt{H}$	1.73	7.4	12.8	10.3	43.1
`M×R×H`	1	14.2	65.5	69.8	2792

Table 2: A comparison of the different complexity / structural-organization scores of the best tables evolved with the different representations.

The different scores for the best tables evolved with the different representations vary in how well they correlate to the average best fitness scores. AIC seems to go counter to fitness, with high AIC scores for the representation *none*, which has the lowest average best fitness, and low AIC scores for the representation *MRH*, which has the highest average best fitness. The metrics that correctly have the lowest score for the representation with lowest best fitness, *none*, and highest score for the representation with highest best fitness, *MRH*, are Sophistication, Connectivity, Modularity, the three Reuse metrics and Hierarchy. Of these metrics, most of them – Sophistication, Modularity, and Hierarchy – give higher scores for the representation with Modularity and Hierarchy (MH) enabled than they do for the representation with Modularity and Reuse (MR) enabled even though the representation MR has a higher best fitness than does MH. The only metrics that have the correct ordering are Connectivity and the number of Modules in the Design.

Each of the proposed metrics of modularity, reuse and hierarchy measure different aspects of the structural organization of an object. Of interest is combining the scores of these three metrics into a single value measure of structural organization. This can be done by treating each of modularity, reuse and hierarchy as the different axes of a 3D system and then using the length of the vector from the origin to the point in that as the single-value measure of structural organization of an object. Since we have three metrics for reuse, there are three MRH metrics of structural organization:

$$MRH = \sqrt{M^2 + R^2 + H^2} \quad (2)$$

$$MR_bH = \sqrt{M^2 + R_b^2 + H^2} \quad (3)$$

$$MR_mH = \sqrt{M^2 + R_m^2 + H^2} \quad (4)$$

While the scores of the three `MRH` metrics – `MRH`, $\texttt{MR}_b\texttt{H}$ and $\texttt{MR}_m\texttt{H}$ – do not have the same ordering as does the ordering for fitness (see Table 2) by scaling the contributions of M, R and H they could be made to match. One way would be to increase the weighting for Reuse and the other way would be to decrease the weighting for Hierarchy. Alternatively, rather than using the number of modules in the program as the value for the modularity score, the number of modules in the design could be used. Since reporting the number of modules in the design is equal to the number of modules in the program times the modular reuse this metric seems to capture reuse twice so another option is to just combine this score with the score for Hierarchy. In fact, multiplying Modularity in the Program, Reuse$_a$ and Hierarchy produces a score in which the ordering for the five representations matches their fitness ordering.

8. DISCUSSION

Enabling the hierarchical creation of reusable modules in a design improves the ability of generative representations to scale in sophistication and number of parts. In the first case, designs often have dependencies such that changing one component in a design requires the simultaneous change in another component. For example, in creating a design for a dining-room table the length of each table leg is dependent on the lengths of the other legs in the table and it is only useful to change the lengths of all legs together. By having a single description of a table leg, with references to this description at each place where it is used, all table legs are changed by changing this one description. Without reusable modules the CAD system must find and change all occurrences of a leg together, but this is feasible only when the dependencies are known beforehand and not when they are created during the search process. In the second case, as the of number parts in a design increases there is an exponential increase in the size of the design space. Since search consists of iteratively making changes to designs that have already been discovered, this increase in the design space reduces the relative effect of changing a single part in a design and increases the number of changes needed to navigate the design space. Increasing the amount of change made before re-evaluating a design is not a viable solution because this increase produces a corresponding decrease in the probability that the resulting design will be an improvement. With a generative representation the ability to combine and reuse previously discovered assemblies of parts, by either adding or removing copies, enables large, meaningful movements about the design space. Here the ability to hierarchically create and reuse organizational units acts as a scaling of knowledge through the scaling of the unit of variation.

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 programs and designs. The best table evolved without any of MR&H enabled is shown in figure 3.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 can be seen in the randomness of its assembly procedure, which is the same as its program and is shown in figure 3.c. In contrast the table evolved with all of MR&H enabled, figure 3.b, has straight legs going all the way from top to bottom and a fully filled surface. It has a modularity of 34 for the program, a modularity of 1298 for the design (the 34 modules in the program are collectively used a total of 1298 times), a hierarchy of 9 and reuse of 10. Its program is shown in figure 3.d and the structural organization it produces can be seen in the sophisticated, multi-level patterns in the assembly procedure it generates, figure 3.e.

Finally, compare the `MRH` metrics against AIC, which measures the amount of information needed to specify an algorithm for describing a string. For example the 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 structural organization of this string by measuring the modularity, regularity and hierarchy values in mapping from the encoding (the uncompressed string is both the encoded design and the design itself since it is incompressible) to the design 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 `MRH` match our intuition that this random string does not have a sophisticated structure. In fact, we could say that the information contained in the string is not organized in a sophisticated structure and thus to give an idea of how complex an object is it may be useful to state both the amount of information in it as well as the degree to which that information is organized.

9. CONCLUSION

The designs that we can achieve are limited only by our imagination and the tools we use, similarly the designs that CAD systems can achieve are limited only by the representations with which they operate. In this paper we have argued that to improve the scalability of CAD systems the types of characteristics 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 design encoding. 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 `MRH`.

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 the representational at-

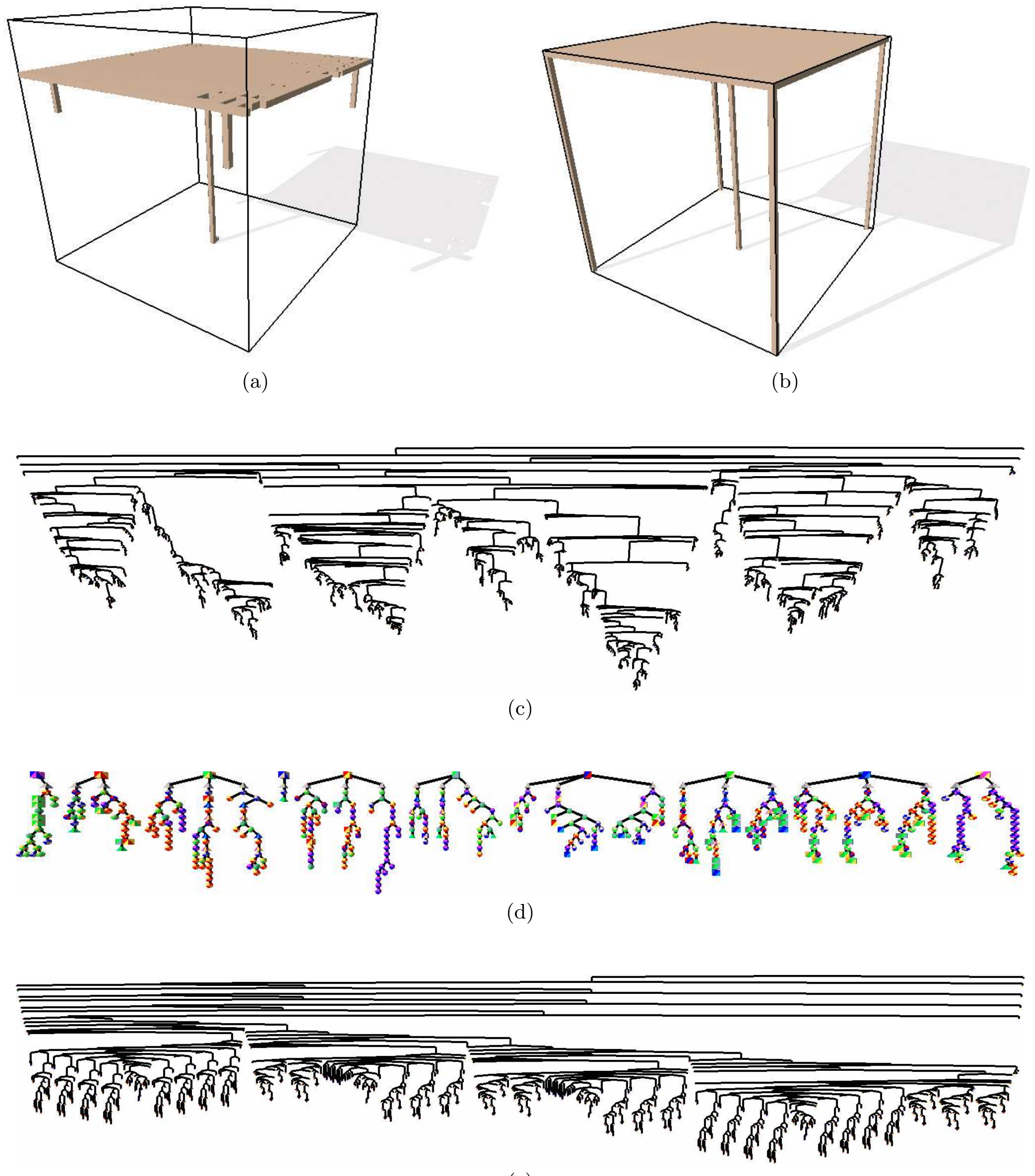

Figure 3: Results from the experiments on the 50x50x50 table design problem: (a) the best table evolved with none of MR&H enabled has a fitness of 25116510 (the black lines mark the limits of the 50x50x50 design space); its encoding, which is the same as the assembly procedure for constructing it, is shown in (c) and has 4999 operators; (b) the best table evolved with MR&H all enabled has a fitness of 60098951; its program is shown in (d) and has 495 operators; its resulting assembly procedure is shown in (e) and has 4871 operators.

tributes MR&H enabled on different sizes of a 3D table design problem. The results show that enabling modularity, regularity and hierarchy in the representation improves the performance of search with a CAD system. In comparison against various complexity metrics our modularity, regularity, and hierarchy metrics of structural organization (MRH) are among the ones that are well correlated with good table designs.

In the future we expect that more improvements in the scalability of CAD systems 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 CAD systems will be better able to produce ever more sophisticated engineering systems.

10. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill, second edition, 1996.

[2] C. H. Bennett. On the nature and origin of complexity in discrete, homogenous, locally-interacting systems. *Foundations of Physics*, 16:585–592, 1986.

[3] P. J. Bentley, editor. *Evolutionary Design by Computers.* Morgan Kaufmann, San Francisco, 1999.

[4] P. J. Bentley and D. W. Corne, editors. *Creative Evolutionary Systems.* Morgan Kaufmann, San Francisco, 2001.

[5] G. J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the Association of Computing Machinery*, 13:547–569, 1966.

[6] B. Edmunds. *Syntactic Measures of Complexity.* PhD thesis, Dept. of Philosophy, University of Manchester, 1999.

[7] 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.

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

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

[10] 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.

[11] G. S. Hornby. ALPS: The age-layered population structure for reducing the problem of premature convergence. In M. K. et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference, GECCO-2006*, pages 815–822, New York, NY, 2006. ACM Press.

[12] G. S. Hornby, J. D. Lohn, and D. S. Linden. Computer-automated evolution of an X-band antenna for NASA's Space Technology 5 mission. *IEEE Transactions on Evolutionary Computation*, accepted.

[13] 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.

[14] G. S. Hornby, S. Takamura, T. Yamamoto, and M. Fujita. Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410, 2005.

[15] 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.

[16] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–17, 1965.

[17] M. Koppel. Complexity, depth and sophistication. *Complex Systems*, 1:1087–1091, 1987.

[18] J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, Cambridge, MA, 1992.

[19] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406:974–978, 2000.

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

[21] Z. Michalewicz, D. Dasgupta, R. G. L. Riche, and M. Schoenauer. Evolutionary algorithms for constrained engineering problems. *Computers and Industrial Engineering Journal*, 30(2):851–870, 1996.

[22] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[23] G. Robinson, M. El-Beltagy, and A. Keane. Optimization in mechanical design. In P. J. Bentley, editor, *Evolutionary Design by Computers*, chapter 6, pages 147–165. Morgan Kaufmann, San Francisco, 1999.

[24] B. K. Rosen. Syntactic complexity. *Information and Control*, 24:305–335, 1974.

[25] R. J. Solomonoff. A formal theory of inductive inference. *Information and Control*, 7:1–22,224–254, 1964.

[26] K. Ulrich and K. Tung. Fundamentals of product modularity. In *Proc. of ASME Winter Annual Meeting Symposium on Design and Manufacturing Integration*, pages 73–79, 1991.

[27] V. H. Yngve. A model and an hypothesis for language structure. In *Proceedings of the American Philosophical Society*, pages 444–466, 1960.