

Modularity, Reuse and Hierarchy: Measuring Complexity by Measuring Structure and Organization

Gregory S. Hornby¹

*University of California Santa Cruz¹
NASA Ames Research Center, Mail Stop 269-3
Moffett Field, CA 94035-1000
hornby@email.arc.nasa.gov*

Abstract

To develop better complexity measures, a reasonable approach is to base them on those principles of design that designers use. Modularity, reuse and hierarchy (MR&H) have been identified by engineers as useful principles for designing complex systems, and these characteristics can be seen in Nature. Here we develop metrics for each of MR&H, and then use them to develop several metrics of *structure and organization*. Further, we propose to evaluate complexity measures both empirically and on a set of abstract object-construction examples. After applying these tests to a handful of previously defined complexity measures, as well as ones we define here, we find that only two of our measures pass both sets of tests.

Key words: design, evolutionary algorithm, evolutionary design, complexity, structure, organization

1 Introduction

Over the years various methods have been proposed for measuring the complexity of an object, such as Algorithmic Information Content (AIC) [1–3], Logical Depth [4], and Sophistication [5]. These metrics vary in how intuitively they measure complexity, with definite cases in which they are counter intuitive. For example, consider two strings (or computer programs) of equal length, with the first consisting of a random sequence of symbols and the second having hierarchies of regularities. The AIC of the first string will be higher than the second whereas we are intuitively inclined to think of the second string as more complex. In addition, it is not always clear that what is being measured is a meaningful property of an object. Of interest is the development of a better method for measuring complexity

that produces results that are more intuitive and is a measure of meaningful design characteristics.

One approach to developing better complexity measures is to take the view that there are fundamental principles of scalable design and that the complexity of an object is well correlated with the degree to which it has these characteristics. Continuing along this line of thought, then to develop better complexity measures we should look to design-related disciplines to see which characteristics have been identified as necessary for scalable design. In engineering and software development sophisticated artifacts are achieved by exploiting the principles of modularity, reuse, and hierarchy (MR&H) [6–8], and these characteristics can also be seen in the artifacts of the natural world. Assuming that the principles of MR&H are necessary to achieve scalability, then a meaningful and intuitive set of complexity metrics would be based on measures of MR&H.

Here we define measures of MR&H, as well as several variations of combining them into a single metric of *structure and organization* and compare them against existing measures of complexity. To define metrics of MR&H in a way that generalizes across different design domains, we need an abstract model of an object that can be analyzed. While the field of Complexity has a tradition of working with string-based programs that produce strings, here we use graph structures since they are a more powerful data structure than are strings. An object can be encoded by a graph-structured *design program* which, when executed or compiled out, produces a tree-structured *assembly procedure* for the object. It is on an object’s design program and assembly procedure that we define our metrics.

To demonstrate the usefulness of using MR&H to measure complexity, we propose two types of tests for evaluating a complexity measure and apply them to our measures as well as to a handful of existing ones. First, complexity measures can be compared by using an automated design system to create a number of designs for various sizes of a scalable design problem. When applied to the best designs of each “size” of the problem, we expect a good complexity measure to produce monotonically increasing values as the problem size scales up. Second, complexity measures can be compared on abstract examples to determine whether or not they have certain properties. We suggest three such properties: random designs should score low; combining an object with itself should produce at most a small increase in complexity; and an object built from joining two sub-objects should not be much more complex than either. After applying these tests to the different complexity measures we find that only two of our measures of structure and organization pass both sets of tests.

The rest of this paper is organized as follows. First, we describe our model of design representations (in Section 2), since this is needed to define the metrics operate on them. Then we present our measures of Modularity, Reuse and Hierarchy (in Section 3), followed by several methods for combining these measures into a com-

posite measure of structure and organization (in Section 4), and then describe the other measures of complexity we compare against (in Section 5). Next we describe our experimental setup for evaluating the different measures on different sizes of a design problem (in Section 6) and present the results of applying this empirical test to the different complexity measures (in Section 7). We then propose three properties which a complexity measure should have and use this as a second test with which we evaluate complexity measures (in Section 8). Based on the results from evaluating the different complexity measures on our two tests we conclude that the best measures are two of our measures of structure and organization. Finally we close with a discussion (in Section 9) and a summary of this work (in Section 10).

2 Design Encodings are Programs

Before defining the various complexity measures, it is worth describing the paradigm under which these measurements are taken. To define metrics of a design or artifact in a way that generalizes across various types of domains, we take measurements on the data structures that encode these artifacts rather than take measurements of actual fabricated artifacts. These data structures can be thought of as a forest of tree-structured design-construction operators, in which each tree describes the assembly of some parts, or sub-assemblies, into a larger one (see Figure 1(a)). Since this data-structure defines how the artifact is “built,” we consider it a *design program* for building it. Just as computer programs have procedure calls and iterative loops, so too can design programs have analogous constructs. If we add links to the trees to represent the jumps in control-flow of these procedure calls and iterative loops, this results in this forest of trees becoming one large, inter-connected graph comprising of multiple sub-graphs for each sub-assembly, and sub-sub-graphs for the sub-sub-assemblies, and so on (see Figure 1(b)). Continuing with the “a design encoding is really a design program” metaphor, these graph-structured design programs can be executed to produce a tree of design-construction operators, called an *assembly procedure*.

Viewing an object as a graph differs significantly from what is commonly done in the field of Complexity, which is to examine both the object and the program which generates the object as strings, which are typically the input and output of a Turing Machine. Reducing an object and its generating program to strings removes the connections which describe couplings and structure and it also severely limits what can be measured. For example, if a procedure in a design program is called from multiple locations this can be represented in a graph by using links from the calling nodes to the head of the procedure, whereas in a string there is no way to encode this. Similarly, if an assembly in a design has three sub-assemblies attached to it, a graph-structured representation can encode this as a node with three nodes attached to it, whereas with a string-based representation this information cannot be encoded without using labeled nodes with special meanings.

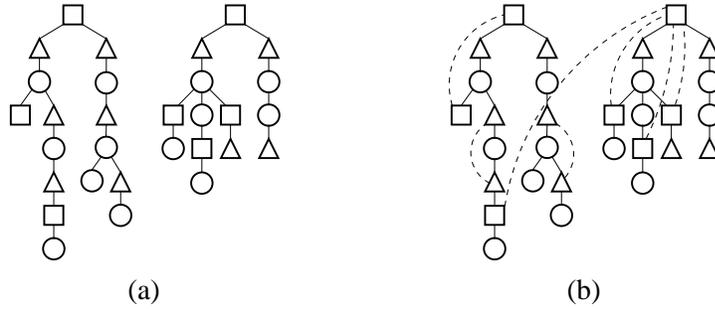


Fig. 1. A graphical rendition of a design program in which different shapes represent different types of operators: (a) the tree structured procedures in the program; (b) the tree-structured procedures in the program with the links added to show procedure calls and the extent of iterative loops.

This paradigm of considering a design as consisting of three parts – design program, assembly procedure and the resulting design – is a kind of extension of the traditional paradigm for investigating ideas in Complexity. Existing complexity measures commonly measure the complexity of a string of symbols by measuring attributes of the minimal, or near minimal, program that generates it. This two part paradigm of a program and the string it generates corresponds with the first two parts (design program and assembly procedure) of the model described here. The third part is added by using the string of symbols as a linear assembly procedure in which the symbols are used as operators for “constructing” a design, much like the design-construction system described in Section 6.2.

3 Measuring Modularity, Reuse and Hierarchy

The method for measuring MR&H comes out of what is meant by these terms. *Modularity* is defined as an encapsulated group of elements that can be manipulated as a unit, *reuse* is a repetition or similarity in a design, and *hierarchy* is the number of layers of encapsulated modules in the structure of a design.

Each of MR&H aids the scalability of evolutionary design systems in different ways. For larger and more sophisticated artifacts, being able to hierarchically create levels of nested modules is needed to break things down so no one module is too large and sophisticated to evolve on its own. This is analogous to Simon’s parable of two watchmakers, which illustrated how the ability to hierarchically create and manipulate modules greatly improves the rate at which more structurally sophisticated artifacts can be built [9]. Being able to reuse design modules is helpful in two ways. First, a module that is useful in one part of the design may be useful somewhere else, so creating modules is a way of scaling the basic unit of variation. Second, reuse of a parameter, assembly or function is a way of capturing design dependencies into a single location in the design program thereby enabling design-

ers (or search algorithms) to more easily make coordinated changes in the design [10,11]. We now define our measures of MR&H.

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 program that can be manipulated as a unit. Since a label to a procedure can be manipulated as a unit, each procedure in the design program 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 constitutes a module, hence each iterative block is one module in the program. As well as counting modules in the design program (which we label M_p , for modules in the program) we can also count the number of occurrences of modules in the design itself, M_d . 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.

Reuse: is a measure of the average number of times elements of the design program are used to create the resulting design. Here we measure three types of reuse. The first, overall reuse, R_a , is calculated by dividing the number of symbols in an object's assembly procedure by the number of symbols in the design program that generates it. Second, reuse of build symbols, R_b , is the average number of times a design constructing operator – as opposed to an operator that is a conditional, iterative statement or procedure call – is used. Third, reuse of modules, R_m , is the average number of times modules are reused in the design and is calculated by dividing M_d by M_p .

Hierarchy: The hierarchy of a 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 zero. 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 [12], but whereas there the measure is of basic steps in constructing an object, here we are measuring steps of nested modules.

As defined, these measures of MR&H apply to any programming language, and are thus comparable on the same systems as existing complexity measures, such as AIC, Logical Depth and Sophistication. These measures can also be generalized to any representation with a hierarchical graph structure, such as the set of parts used to describe a complex assembly in a CAD/CAM package, and any system that can be described as a hierarchical graph structure, such as a regular expression, context-free grammar or unrestricted grammar. For example, the regular expression $(0 + 10)^*((11)^* + 001)^*$ can be thought of as a design program consisting of 19 symbols with the modules 0, 10, 11, 001, $0 + 10$, and $(11)^* + 001$. Similarly, the modularity, reuse and hierarchy scores of strings in this language could be measured. The string 1010100011111001 is in this language and consists of three occurrences of module

10 and three occurrences of module $(11)^* + 001$, one of which is two occurrences of module 11 and the other two are occurrences of module 001. It has a design modularity, M_d , of 10, an hierarchy, H , of 2 and its reuse scores are an R_a of 0.84, an R_b of 2.0 and an R_m of 1.67.

In the rest of this paper we use MRH to refer to the metrics for modularity, reuse and hierarchy and MR&H to refer to the characteristics of modularity, reuse and hierarchy. Next, in Section 4 we discuss how to combine these measures of MRH into a single measure, which we call a measure of *structure and organization*.

4 A Single Metric for Structure and Organization

Each of the proposed metrics of modularity, reuse and hierarchy measure different aspects of the structure and organization of an object. Of interest is combining the scores of these three metrics into a measure of structure and organization with a single value.

One method for combining the three scores of MRH into a single value is by treating each of them as the orthogonal axes of a 3D system and then using the length of the vector from the origin as the measure of structure and organization of an object.

$$SO_1 = \sqrt{M_p^2 + R_a^2 + H^2} \quad (1)$$

$$SO_2 = \sqrt{M_p^2 + R_m^2 + H^2} \quad (2)$$

A problem with this approach is that the different metrics vary in their range, and a small change in hierarchy will generally have little impact on the overall structure and organization measure of an object since hierarchy usually has the smallest value.

Another method for combining the three MRH scores is to multiply them together.

$$SO_3 = M_p \times R_a \times H \quad (3)$$

$$SO_4 = M_p \times R_m \times H \quad (4)$$

This approach has the desirable property that a change of $X\%$ in any one of the MRH values will result in the same $X\%$ change in the overall measure of structure and organization.

Of concern with the above approaches to measuring structure and organization is that they do not take into account the size of the object or the size of the program that generates it. For example, a large object with a small percentage of its information organized into some structure can out score a much smaller object which has a

small, maximally-organized, design program. Two ways to normalize structure and organization scores for size are to divide by the size of the object and to divide by the size of the design program (which is the amount of information in the object).

$$SO_5 = \frac{M_p \times R_a \times H}{DesignSize} \quad (5)$$

$$SO_6 = \frac{M_p \times R_m \times H}{DesignSize} \quad (6)$$

$$SO_7 = \frac{M_p \times R_a \times H}{AIC} \quad (7)$$

$$SO_8 = \frac{M_p \times R_m \times H}{AIC} \quad (8)$$

5 Other Complexity Metrics

To demonstrate that the MRH metrics of structure and organization are meaningful we evaluate them, and a handful of other complexity measures, on two types of tests. Measures that were selected are those that are relatively straightforward to compute or approximate and which we thought had a reasonable chance at being relevant. Examples of measures we left out are: Arithmetic Complexity [13], Cognitive Complexity [14], Dimension of Attractor [15], Ease of Decomposition [16], Logical Complexity [13], Mutual Information [17], Number of Inequivalent Descriptions, Number of States in a Finite Automata [18], Number of Variables, and Thermodynamic Depth [19]. All of these measures, as well as many others, are reviewed in [16]. In addition we also add some additional measures of to serve as a kind of control variables. We now review the different complexity metrics which we evaluate.

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 [1], Kolmogorov [2], and Solomonoff [3]. The AIC of a given string is the length, in number of symbols, of the shortest program that produces that string. Other complexity measures are very similar, such as counting the number of lines of code in a computer program [16]. Here we measure AIC as the number of symbols in the design program.

Design Size (DS) is a measure of the size of what is encoded by the design program, and here we measure this by counting the number of symbols in the assembly procedure. This contrasts with AIC, which counts the number of symbols in the program that generates the assembly procedure.

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 it [4].

This can also be considered computational complexity, in that it is a measure of the amount of computational time that is spent to compute the assembly procedure. In our experiments we calculate Logical Depth as the number of symbols that are processed in generating the assembly procedure from the design program.

Sophistication is a measure of the structure of a string by counting the number of *control* symbols in the program used to generate it [5]. In trying to measure the structure of a string, the goal for this measure is similar to the goal of the MRH metrics. Here we calculate the sophistication of a design by counting the number of control symbols – that is, procedure symbols, loop symbols, and 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 design-constructing interpreter and we call them *build* symbols, since they are used to generate a design.

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 [16]. The representation used here can be thought of as a kind of grammar, with different procedures being different grammar rules. Thus to calculate the grammar size of an assembly procedure we use the design program that produces it as the grammar and count the number of production rules in it.

Connectivity: more complex systems have greater inter-connectedness between components, thus the connectivity of a system can be used as a complexity measure [16]. 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 a design we use the connectivity of the design program (in graph form) that is used to generate it.

Number of Branches: related to the previous measure of complexity, we propose another measure of the structure of a graph: 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. We apply this measure to the assembly procedure.

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 [20] or depth and nesting, called Syntactic Depth [21]. We apply this measure to the assembly procedure.

6 Experimental Setup for the Empirical Test

The first test on which we evaluate complexity measures is to empirically test them on different “sizes” of a class of designs. For this test we use an evolutionary algorithm [22,23] to evolve designs for different sizes of a design problem and then apply the different measures to the best evolved designs of each size. This test uses the assumption that as we scale up a design problem, a more “complex” design is needed to produce good designs for it. Consequently, for this test a good complexity measure is one whose values grow monotonically with the increase in design size. We now describe the test problem and the evolutionary design system, GENRE, used for these experiments.

6.1 Test Problem

For the empirical test, the design problem we use is that of producing a three dimensional table out of cubes. A table is evaluated by first determining whether or not it will fall over, which is done by testing whether or not its center of mass falls within its footprint. Tables which are found to fall over are given a fitness score of zero, and tables which are found to stand up are further evaluated using a function of their height, surface structure, stability and the number of excess cubes used [24,11]. 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,

$$f_{height} = \text{the height of the highest cube, } Y_{max}. \quad (9)$$

$$f_{surface} = \text{the number of cubes at } Y_{max}. \quad (10)$$

$$f_{stability} = \sum_{y=0}^{Y_{max}} f_{area}(y) \quad (11)$$

$$f_{area}(y) = \text{area in the convex hull at height } y. \quad (12)$$

$$f_{excess} = \text{number of cubes not on the surface.} \quad (13)$$

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

$$\text{fitness} = f_{height} \times f_{surface} \times f_{stability} / f_{excess} \quad (14)$$

This problem can be scaled by varying the size of the grid. In our experiments we perform runs with sizes from $20 \times 20 \times 20$ to $80 \times 80 \times 80$.

6.2 Representation

To encode tables, the representation used by GENRE is a kind of program which specifies how to construct a table. This program consists of a forest of tree-structured procedures in which each node in the tree is an operator, and 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 following example of a design encoded with GENRE's representation consists of two labeled procedures, `Proc_0` and `Proc_1`, each with two parameters, and the initial call to the program, `Proc_0(4.0, 2.0)`:

Proc_0(4.0, 2.0) :

Proc_0(n_0, n_1) :

$n_0 > 3.0 \rightarrow$ 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 []]]

Graphical versions of this design program are shown in Figures 1 and 2(a).

To generate the assembly procedure for this design program it is executed, starting with the statement `Proc_0(4.0, 2.0)`. This results in the following 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)
```

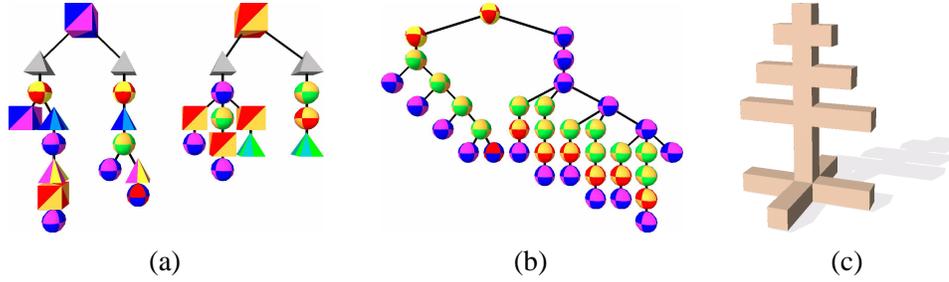


Fig. 2. This figure contains: (a) a graphical version of an example design encoding; (b) the assembly procedure it produces; and (c) the resulting design.

```
[ ] ] ] ] 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) ] ] ] [ ] [ ] [ ] [ ] ] ]
```

A table is constructed by starting with a single cube in an otherwise empty 3D grid and then the assembly procedure is executed to add more cubes to the structure. Cubes are added to this design 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 there are three operators, `rotate-[x|y|z]()`, that change the current orientation in units of 90° about the appropriate axis. 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.

A graphical version of this design program is shown in Figure 2(a), along with the corresponding assembly tree of design-construction operators, Figure 2(b), and the resulting design, Figure 2(c). In the images of the design program and assembly procedure, cubes represent labeled procedures and the calls to them, pyramids represent control-flow operators, and construction operators are represented by spheres.

This example design can be analyzed using the metrics of MRH and the various complexity measures. The program has six modules which 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 reuse value of 1.27, and it has five levels of nested modules which gives a hierarchy value of 5. Its scores on the other complexity measures are: an AIC of 30; a Design size of 38; a Logical Depth of 124; a Sophistication of 21; 13 build symbols; a grammar size of 2; a connectivity of 5; 8 branches; and a height of 10.

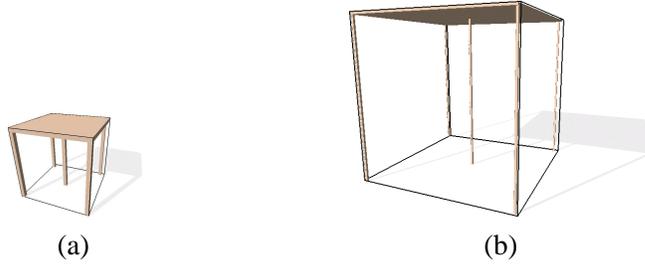


Fig. 3. Two of the best, and most structurally organized, of the evolved tables. The first (a) was evolved in the $20 \times 20 \times 20$ design space and the second (b) was evolved in the $80 \times 80 \times 80$ design space.

6.3 Evolutionary Algorithm

The EA used for these experiments is the Age-Layered Population Structure (ALPS) [25]. Unlike a traditional EA, ALPS maintains several layers of individuals of different age levels 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. The setup we use consists of 10 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. Tournament selection with a tournament size of 5 is used to select parents. In our experiments we run 15 trials with each configuration and each trial is run for one million evaluations.

7 Results of the Empirical Test

To compare complexity and MRH metrics we performed a number of evolutionary design runs on different sizes of a design problem. The design problem and evolutionary algorithm were described in the previous section, and for these experiments we evolved tables for four different grid sizes. Since we assume that as the design space is increased in size more complex designs will be needed, we are looking for complexity measures whose values scale up along with this increase.

Figure 3 contains images of two of the best and most structurally organized tables that were evolved. The smaller table, Figure 3(a), was evolved in the $20 \times 20 \times 20$ design space and has a fitness of 582221 and the following scores: AIC of 913; Design Size of 8007; Logical Depth of 10311; Sophistication of 89; 811 build symbols; a Grammar Size of 13; a Connectivity of 34; 1595 branches; a height of 155. Its MRH scores are: M_p is 34, M_d is 431; R_a is 8.8; R_b is 9.9; R_m is 12.7 and it has an H of 8. The larger table, Figure 3(b), was evolved in the $80 \times 80 \times 80$ design space and has a fitness of 600324286 and the following scores: AIC of 630; Design Size of 9753; Logical Depth of 14365; Sophistication of 90; 529 build symbols; a

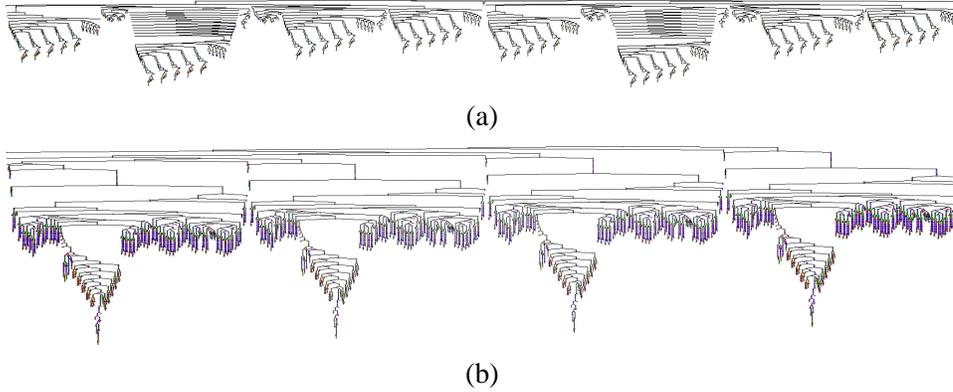


Fig. 4. A graphical rendition of the assembly procedures for constructing the two tables in Figure 3. The assembly procedure in (a) produces a table for the 20x20x20 design space and the assembly procedure in (b) produces a table for the 80x80x80 design space.

Grammar Size of 11; a Connectivity of 58; 1668 branches; and a height of 168. Its MRH scores are: M_p is 20, M_d is 2202; R_a is 15.5; R_b is 18.4; R_m is 110.1 and it has an H of 9. While these scores give examples of the differences that can happen, a better overall picture is gained from looking at the average values from a number of evolutionary runs on different sizes of the design problem.

Table 1 lists the average values over 15 trials of the different measures as applied to the best tables evolved on different sizes of the design problem ($20 \times 20 \times 20$, $40 \times 40 \times 40$, $60 \times 60 \times 60$, and $80 \times 80 \times 80$). As expected, the averaged best fitness monotonically increases along with an increase in size of the design space. The measures which have values that also monotonically increase in step with an increase in size of the design space are: Design Size, Logical Depth, M_d , R_m , and H. Of these it is not surprising that Design Size increases with the size of the design space and, given that the Design Size increases, it is also not surprising that Logical Depth (a measure of the running time of the program that creates the assembly procedure) also increases with size of the design space. Interestingly, the information in a design, AIC, does not grow monotonically with size of the design space or Design Size. In addition, none of the other measures grows monotonically with the size of the design space except some of the MRH measures: the amount of modularity in the design (M_d), the reuse of modules (R_m) and hierarchy (H).

Of the three measures of reuse, R_a , R_b and R_m , only modular reuse (R_m) monotonically increases with the size of the design space and the fitness of the best designs. This suggests that the type of reuse that is useful is not overall reuse (R_a) or reuse of build symbols (R_b), but the reuse of modules. By extension, this also suggests that those design representations which do not have the ability to hierarchically assemble and reuse modules (such as artificial genetic regulatory networks [26]) will not scale well.

Of the two modularity measures, M_d monotonically increased along with the increase in fitness and size of the design space whereas M_p was higher in the $20 \times$

| | 20 ³ | 40 ³ | 60 ³ | 80 ³ |
|----------------------------------|-----------------|-----------------|-----------------|-----------------|
| Fitness ($\times 10^6$) | 0.56 | 18.1 | 123 | 440 |
| AIC | 719 | 768 | 680 | 775 |
| Design Size | 6769 | 9499 | 9739 | 9944 |
| Logical Depth | 9541 | 13421 | 14376 | 18011 |
| Sophistication | 79.9 | 70.53 | 74.0 | 85.4 |
| Number of Build Symbols | 626 | 684 | 593 | 676 |
| Grammar Size | 13.5 | 13.2 | 12.5 | 13.5 |
| Connectivity | 33.7 | 25.2 | 26.4 | 37.3 |
| Number of Branch Nodes | 1653 | 2087 | 1905 | 1825 |
| Height | 118 | 145 | 276 | 220 |
| Modularity (M_p) | 27.5 | 26.1 | 30.8 | 31.1 |
| Mod. in Design (M_d) | 377 | 547 | 1133 | 1329 |
| Reuse (R_a) | 12.1 | 14.0 | 16.6 | 15.7 |
| Reuse of Build Symbols (R_b) | 15.2 | 16.2 | 19.6 | 18.5 |
| Reuse of Modules (R_m) | 15.2 | 21.8 | 37.4 | 50.1 |
| Hierarchy (H) | 7.53 | 7.7 | 8.0 | 8.6 |

Table 1

A comparison of the resulting scores on the different metrics of the best tables evolved with the different representations. Results are the average over 15 trials.

20 \times 20 space than in the 40 \times 40 \times 40 space. Since M_d is a product of the number of modules in a design program (M) and the amount of reuse of these modules (R_m), it may be a more reliable measure of “complexity” because it is a product of two separate aspects: modularity and modular reuse. This suggests that measuring modularity alone is not a good overall measure of the complexity of an object and that combining the measures of all three characteristics of MR&H into a single measure may result in an even better measure of an object’s structure and organization.

Table 2 contains the scores for the different measures of structure and organization (SO) on the best design programs evolved for different sizes of the design problem. Of these eight measures of structure and organization, neither SO_1 and SO_5 increase monotonically along with the size of the design space. Since both of these use overall reuse and not modular reuse this suggests that modular reuse is more important than overall reuse. The other six measures of structure and organization do increase monotonically and, of these six, the four measures of structure and organization which use modular reuse (R_m) seem to scale better than those that use overall reuse (R_a).

| | | 20 ³ | 40 ³ | 60 ³ | 80 ³ |
|----------|---|-----------------|-----------------|-----------------|-----------------|
| | Fitness ($\times 10^6$) | 0.56 | 18.1 | 123 | 440 |
| SO_1 : | $\overline{MR_a \vec{H}}$ | 31.3 | 31.1 | 37.1 | 37.1 |
| SO_2 : | $\overline{MR_m \vec{H}}$ | 34.0 | 36.0 | 51.6 | 64.4 |
| SO_3 : | $M \times R_a \times H$ | 2013 | 2872 | 3708 | 4019 |
| SO_4 : | $M \times R_m \times H$ | 2889 | 4324 | 8643 | 11207 |
| SO_5 : | $\frac{M \times R_a \times H}{AssemSize}$ | 0.31 | 0.31 | 0.38 | 0.40 |
| SO_6 : | $\frac{M \times R_m \times H}{AssemSize}$ | 0.42 | 0.46 | 0.89 | 1.13 |
| SO_7 : | $M \times R_a \times H / AIC$ | 3.22 | 4.68 | 6.75 | 6.77 |
| SO_8 : | $M \times R_m \times H / AIC$ | 4.59 | 6.87 | 15.4 | 19.3 |

Table 2

Different ways of combining MRH scores to produce a single measure of structure and organization.

Overall, we conclude that the measures which pass the empirical test for this design problem are AIC, Design Size, Logical Depth, M_d , R_m , H , SO_2 , SO_3 , SO_4 , SO_6 , SO_7 , and SO_8 . Certainly there are limitations with any empirical test – in this particular one biases are introduced from such things as constraints on the maximum size of our design program and on the particular design problem we chose. While doing more empirical tests on different design problems using different automated design systems can improve the reliability of this test, another approach is to evaluate a complexity measure on an abstract object-construction example.

8 Object Construction Test

One shortcoming with some measures of complexity, such as AIC, is that they are not very intuitive. We can examine how intuitive these measures of structure and organization are by testing them on abstract object-construction examples. First, consider the AIC of an algorithmically random bit string, by which is meant one with no regularities. 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. Compare this to the MRH and structure and organization values of this

string: its modularity value is 0, since it has no modules, its reuse value is 1, since there are no reused symbols, and its hierarchy value is 0, since there is no modules to be nested. Using these values, its various structure and organization values ($SO_1 \dots SO_8$) are: 1, 1, 0, 0, 0, 0, 0, and 0. These values of 0 and 1 for the measures of MRH and structure and organization match our intuition that a random string does not have a sophisticated structure.

We can also compare how the different complexity measures scale as we scale the size of the random string. For AIC, Design Size and Logical Depth – the three measures that passed the empirical test of Section 7 – they all produce a complexity value of $length(A) + k$, for string A , where k is the overhead for performing the printing. These scores imply that longer random strings are more complex than shorter ones. In contrast, the MRH and SO measures all produce the same complexity measure of 0 or 1 for any random string: more randomness is not more complexity.

Next, consider what happens to the different complexity values when an object, A_1 , is joined to itself to form a new object, A_2 . In this case the design program of the new object, A_2 , would be the same as for the original object, plus the module, $A_2 = A_1 + A_1$. As a result of this new module, the hierarchy of A_2 would be $H(A_1)$ plus 1 and the modularity would be $M_p(A_1)$ plus 1. Depending on the AIC of A_1 , the amount of reuse will be up to a factor of 2 larger for the new object since $R_a(A_2) = \frac{DS(A_1)+DS(A_1)+k}{AIC(A_1)}$, where k is the size of adding the new module and $DS(A)$ is the Design Size of A . As a result of these changes in MRH, the structure and organization values of SO_5 through SO_8 should be only slightly larger, but those of $SO_3(A_2)$ and $SO_4(A_2)$ will be roughly double that of A_1 . Consider what happens to other scores of complexity: AIC, Sophistication and Grammar Size increase slightly but Logical Depth doubles. Since A_2 is just two copies of A_1 , it is not clear that it should have twice the complexity of A_1 , thus we conclude that measures SO_5 through SO_8 scale intuitively, whereas $SO_3(A_2)$, $SO_4(A_2)$ and Logical Depth do not scale intuitively on this example.

Similarly, consider the case in which two completely different objects, A_1 and A_3 , with the same complexity and MRH scores, are combined to form a new object, A_4 : $A_4 = A_1 + A_3$. In this case the new module results in the hierarchy of the new object being one plus the hierarchy of either of its component objects: $H(A_4) = H(A_1) + 1 = H(A_3) + 1$. The modularity of this new object is equal to one plus the sum of its to component objects: $M(A_4) = M_p(A_1) + M_p(A_3) + 1$. Whereas both modularity and hierarchy increase, this new object has a reuse slightly less than both of its component objects since the size of the design is $DS(O_1) + DS(O_3)$ but the size of the design program is $AIC(O_1) + AIC(O_3)$ plus some additional symbols for specifying $A_4 = A_1 + A_3$.¹ Thus SO_3 and SO_4 would be (roughly)

¹ To be precise, the design programs for both A_1 and A_3 have a starting rule, one of these is kept and is changed to call the new rule, $A_4 = A_1 + A_3$, and the other starting rule is deleted so the AIC of A_4 is only a couple of symbols larger than $AIC(A_1) + AIC(A_3)$.

double in value for A_4 as they are for A_1 and A_3 , but SO_5 through SO_8 would change little since both AIC and design size would also (roughly) double in size. Not only would AIC for A_4 be roughly double that of either A_1 or A_3 , but so would Logical Depth, Sophistication, and Grammar Size. Just as combining an object with itself does not seem like it should lead to a doubling in complexity, neither does it seem that combining two completely different objects with the same complexity should lead to a doubling of complexity. Thus, as with the previous example, we find that the more intuitive measures of complexity are SO_5 through SO_8 .

To summarize the results of these three object-construction examples we can state some desirable properties of a measure of complexity:

- 1: The complexity value of a random string should be small.
- 2: The complexity value of an object joined to itself should be no more than slightly larger than that of the original object.
- 3: The complexity value of two objects joined together should not be smaller than the lesser value of the two original objects and should not be much larger than the greater value of the two original objects.

Using these principles, and the results of the experiments in Section 7, the best measures of complexity are SO_6 and SO_8 .

9 Discussion

One issue that has not yet been addressed in this paper is that multiple design programs can generate the same object. A common method for handling this is to take measurements on the shortest program that generates the object, as is done for AIC, or on a near minimal program, as is done for Logical Depth. Both of these approaches are valid and can be used for producing objective MRH and structure and organization measures of an object. At least for theoretical analysis this works fine, but for actual real-world objects it becomes somewhat problematic.

Given an object, along with the CAD/CAM files that make up its design, how does one measure its complexity? It can be impractical to create the minimal program, so performing measurements on the design program that was developed is not only a pragmatic solution but it has other advantages. Designs typically go through multiple revisions regardless of whether they were produced manually or evolved through a natural or artificial system, and the resulting design program captures part of this revision process. Similarly, a design program can be thought of as a point in the design space with a neighborhood of designs that are near it with minimal changes. Different design programs for the same object will have a different neighborhood of designs that are easily reachable. Measuring the complexity of an object by measuring the actual design program that was developed for it may be useful in

capturing something about how it was designed or where it is in the greater design space.

Finally, for an automated design system to be able produce designs with certain types and levels of complexity it must have a representation capable of encoding such designs and a search algorithm that can take advantage of the representation. For example, for an automated design system to be able produce designs with certain types and levels of complexity it must have a representation capable of encoding such designs. In the empirical experiments, the representation we used is a kind of computer program with combination, abstraction and control-flow implemented in a particular way. With a different representation one or more of MR&H, or some other type of design characteristic, may not be possible. In this case, on an empirical comparison the measured amount of that characteristic would be the same for all designs so it would not fail the empirical test. Unfortunately, we cannot make strong guarantees on the performance of the search algorithm so we cannot expect the empirical test to be completely reliable, and hence the need for evaluating complexity measures on abstract object-construction scenarios.

10 Conclusion

To develop better complexity measures, a reasonable approach is to base them on those principles of design that designers use. Modularity, reuse and hierarchy have been identified by engineers as useful principles for designing complex systems, and these characteristics can be seen in Nature. Here we developed measures for each of MR&H, and also used these three measures to develop several measures of structure and organization.

To evaluate our proposed complexity measures we compared them against existing complexity measures on two different tests. Working with the hypothesis that in scaling the size of a design problem more complex solutions are required to solve it, we performed an empirical comparison of different complexity measures using an evolutionary algorithm to evolve solutions to different sizes of a design problem. Of the pre-existing complexity measures, only Design Size and Logical Depth produced values that monotonically increased with the scaling of the problem. Of the measures proposed in this paper, modularity in the design (M_d), reuse of modules (R_m) and hierarchy (H) all scaled appropriately, as did most of our measures of structure and organization. In addition, we proposed three desirable properties of a complexity measure: random designs score low; combining an object with itself should result in, at most, a small increase in complexity; and combining two objects with the same complexity should result in the new object having a complexity that is, at most, a small increase in complexity of its two components. While none of the existing measures of complexity meet all three of this criteria, two measures of structure and organization meet them as well as pass the empirical test. As a result,

we conclude that the best measures of complexity are the two measures of structure and organization SO_6 and SO_8 . These two measures are the product of multiplying the MR&H measures together, and then normalizing by either dividing by AIC (for SO_6) or by dividing by the design size (for SO_8).

The measures we have proposed in this paper are a first attempt at constructing complexity measures based on principles of design. Future work in developing better complexity measures may use different methods for measuring MR&H, combine them in different ways, or use other design characteristics. Regardless of how new complexity measures are developed, we advocate that they are validated both empirically and on abstract object-construction scenarios.

References

- [1] G. J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the Association of Computing Machinery*, 13:547–569, 1966.
- [2] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–17, 1965.
- [3] R. J. Solomonoff. A formal theory of inductive inference. *Information and Control*, 7:1–22,224–254, 1964.
- [4] C. H. Bennett. On the nature and origin of complexity in discrete, homogenous, locally-interacting systems. *Foundations of Physics*, 16:585–592, 1986.
- [5] M. Koppel. Complexity, depth and sophistication. *Complex Systems*, 1:1087–1091, 1987.
- [6] 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.
- [7] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.
- [8] 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.
- [9] H. A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, 1969.
- [10] G. S. Hornby. *Generative Representations for Evolutionary Design Automation*. PhD thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA, 2003.
- [11] 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.

- [12] M. Goldwasser, J.-C. Latombe, and R. Motwani. Complexity measures for assembly sequences. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 1581–1587, Minneapolis, MN, April 1996.
- [13] J.-Y. Girard. *Proof Theory and Logical Complexity*, volume 1. Elsevier Science Publishing Company, New York, NY, 1987.
- [14] J. S. Kelly. Social choice and computational complexity. *Journal of Mathematical Economics*, 17(1):1–8, February 1988.
- [15] G. L. Baker and J. P. Gollub. *Chaotic Dynamics: An Introduction*. Cambridge University Press, Cambridge, UK, second edition, 1996.
- [16] B. Edmunds. *Syntactic Measures of Complexity*. PhD thesis, Dept. of Philosophy, University of Manchester, 1999.
- [17] C. C. H. Adami and N. J. Cerf. Complexity, computation, and measurement. In T. Toffoli, M. Biafore, and J. Leao, editors, *Proc. 4th Workshop on Physics and Computation*, pages 7–11, Boston, MA, November 1996.
- [18] B. R. Gaines. On the complexity of causal models. *IEEE Transactions on Systems, Man and Cybernetics*, 6:56–59, 1976.
- [19] S S. Lloyd and Pagels. Complexity as thermodynamic depth. *Annals of Physics*, 188:186–213, 1988.
- [20] V. H. Yngve. A model and an hypothesis for language structure. In *Proceedings of the American Philosophical Society*, pages 444–466, 1960.
- [21] B. K. Rosen. Syntactic complexity. *Information and Control*, 24:305–335, 1974.
- [22] K. A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.
- [23] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, third edition, 2000.
- [24] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Congress on Evolutionary Computation*, pages 600–607. IEEE Press, 2001.
- [25] G. S. Hornby. ALPS: The age-layered population structure for reducing the problem of premature convergence. In M. Keijzer et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference, GECCO-2006*, pages 815–822, New York, NY, 2006. ACM Press.
- [26] S. Kumar and P. J. Bentley. *On Growth, Form and Computers*. Elsevier Academic Press, 2003.