Functional Scalability through Generative Representations: the Evolution of Table Designs

Gregory S. Hornby

QSS Group Inc., NASA Ames Research Center
Mail Stop 269-3, Moffett Field, CA 94035-1000
hornby@email.arc.nasa.gov

Abstract

One of the main limitations for the functional scalability of automated design systems is the representation used for encoding designs. We argue that *generative representations*, those which are capable of reusing elements of the encoded design in the translation to the actual artifact, are better suited for automated design because reuse of building blocks captures some design dependencies and improves the ability to make large changes in design space. To support this argument we compare a generative and non-generative representation on a table design problem and find that designs evolved with the generative representation have higher fitness and a more regular structure. Additionally the generative representation was found to better capture the height dependency between table legs and also produced a wider range of table designs.

Key words: Generative representation, Representation, Genetic algorithms, Evolutionary design, Lindenmayer Systems (L-systems)

1 Introduction

Computer automated design systems have been used to design a variety of different types of artifacts such as antennas (Lohn *et al.* , in press), flywheels, load cells (Robinson *et al.* , 1999), trusses (Michalewicz *et al.* , 1996), robots (Lipson & Pollack, 2000), and more (Bentley, 1999) (Bentley & Corne, 2001). While they have been successful at producing simple, albeit novel artifacts, a concern with these systems is how well their search ability will scale to more complex design spaces (Drexler, 1989) (Pollack *et al.* , 2001). To improve functional scalability we can look at those fi elds which regularly create large, complex artifacts. In engineering and software development, complex artifacts are achieved by exploiting the principles of regularity, modularity and hierarchy (Ulrich & Tung, 1991) (Huang &

Kusiak, 1998) (Meyer, 1988), which can be summarized as the hierarchical reuse of building blocks.

While the optimization algorithm can affect the degree of reuse in a design, the ability to create structures which reuse building blocks is limited by the ability of the representation to encode them. For example, in optimizing the dimensions on a blueprint, the design system can only produce designs that fall in the pre-specified parameter space. A limitation with this type of representation is that no modification of the search algorithm can affect the degree of reuse in resulting designs, nor is the hierarchical construction of building blocks possible. Thus the ability to automatically generate structures which have a reuse of subassemblies is strongly dependent on the representation used by the design system.

The different types of representations for computer-automated design systems can be classified by how they encode designs. First, designs can be split into *parameterizations* or *open-ended* representations. Parameterizations consist of a set of values for the dimensions of a pre-defined blueprint and open-ended representations are those in which the topology of a design is changeable. Since one of the goals of automated design systems is to achieve truly novel artifacts, we focus on open-ended representations because it is difficult for a parameterization to achieve a type of design that was not conceived of by its creators. A fundamental distinction between open-ended representations is whether they are *non-generative* or *generative*. With a non-generative representation each representational element of an encoded design can map to at most one element in a designed artifact. The two subcategories of non-generative representations are *direct* and *indirect* representations. With a direct representation, the encoded design is a blueprint in which elements can be added/removed in addition to changing their parameters, and with an indirect representation there is a translation or construction process in going from the encoding to the blueprint. A **generative representation** is one in which an encoded design can reuse elements of its encoding in the translation to an actual design. The two subcategories of generative representations are *implicit* and *explicit*. Implicit, generative representations consist of a set of rules that implicitly specify a shape, such as through an iterative construction process similar to a cellular automata and explicit, generative representations are a procedural approach in which a design is explicitly represented by an algorithm for constructing it.

Previously we have argued that the advantage of generative representations over non-generative representations is that they incorporate useful bias into their structure (Hornby & Pollack, 2002). Here we refine and extend these arguments in two ways and support these claims by comparing optimization with a generative representation against optimization with a non-generative representation on our table design problem (Hornby & Pollack, 2001). First we argue that optimization performs better with a generative representation because generative representations are better at capturing some types of design dependencies than non-generative representations. We support this argument by showing that evolution with the generative

representation was better able to create multi-legged tables. Second we claim that generative representations are more conducive to changes thereby improving the ability of the search algorithm to move about in the design space. This is demonstrated by the greater variety of styles of tables produced with the generative representation.

The rest of the paper is organized as follows. First we present our arguments for the advantages of generative representations followed by a review of different automated design systems. Next we describe the parts of our evolutionary design system and then the results of our experiments in evolving table designs. Finally we close with a discussion of our findings and a summary of this paper.

2 Argument for Generative Representations

As the complexity and number of parts in a design grows, the functional scalability of non-generative representations is limited by their weakness in handling the increasing number of design dependencies and the exponential growth in the size of the search space. In the first case, as designs become more complex dependencies develop between parts of a design such that changing a property of one part requires the simultaneous change in another part of the design. For example, if the length of a table leg is changed, then all of the other legs must be changed or the table will become unbalanced. Second, as a design grows in the number of parts the expected distance (in number of parts) between a starting design and the desired optimized design increases. Conversely, changing a single part makes proportionately smaller and smaller moves towards the desired design. One consequence of this is that as designs increase in the number of parts, search algorithms require more steps to find a good solution. Increasing the size of variation (by changing more parts at a time) is not a solution because as the amount of variation is increased, the probability of the variation being advantageous decreases. Non-generative representations are not well suited to handling these increases in size and complexity because their language for representing designs is static.

The advantage of generative representations comes from their ability to reuse previously discovered building blocks. First, reuse of elements of an encoded design allows a generative representation to capture design dependencies by giving it the ability to make coordinated changes in several parts of a design simultaneously. For example, if all the legs of a table design are a reuse of the same component, then changing the length of that component will change the length of all table-legs simultaneously. Secondly, navigation of large design spaces is improved through the ability to manipulate assemblies of components as units. For example, if adding/removing an assembly of m parts would make a design better, this would require the manipulation of m elements of a design encoded with a non-generative representation. With a generative representation the ability to reuse pre-

viously discovered assemblies of parts enables the addition/deletion of groups of parts through the addition/deletion/modification of symbols which represent groups of parts. Here the ability to hierarchically create and reuse building blocks acts as a scaling of knowledge through the scaling of the unit of variation.

3 Review of Automated Design Representations

To assist in our review of design representations, we first define some of their properties. Previously we used the metaphor of design representations as a kind of computer programming language to define the following features of design representations (Hornby & Pollack, 2002):

- **Combination**: The ability to hierarchically create more powerful expressions from simpler ones. While the subroutines of GLib (Angeline & Pollack, 1994) and genetic programming (GP) (Koza, 1992) 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 the `for`-loop in C/C++ programs.
- **Abstraction**: This consists of the ability to encapsulate a group of expressions in the language and label them, enabling them to be manipulated/referenced as a unit, and the ability to pass parameters to procedures. An example of abstraction is the automatically defined functions (ADFs) of GP.

An open-ended representation is generative if it has reuse (which can be through either iteration, procedure labels, or both), otherwise it is non-generative. In the rest of this section we review different design representations for both non-generative and generative representations.

The two classes of non-generative representations are direct and indirect. Direct, non-generative representations typically use a fixed-size data structure for specifying the existence of material at a given location, such as with two-dimensional arrays (Kane & Schoenauer, 1996) (Baron *et al.* , 1999). Indirect, non-generative representations tend to be variable length assembly procedures which specify the assembly of an object (Bentley, 1996) (Funes & Pollack, 1998). An additional layer of indirection is used in those representations in which an object is encoded as a derivation tree for a grammar, with the resulting assembly procedure specifying the assembly of the artifact (Roston, 1994).

One type of grammar for design is Stiny's shape grammars (Stiny, 1980). The grammatical rules of a shape grammar specify a transformation from one shape to another. Optimization with shape grammars consists of producing a derivation tree for the grammar, with a typical approach using simulated annealing to iteratively modify a design (Shea *et al.* , 1997) (Agarwal & Cagan, 1998). While automated design using shape grammars is an example of an indirect representation (since no element of the derivation tree is used more than once) it could be extended to a generative representation by allowing new rules to be added to the grammar through the combination of existing rules.

Most implicit, generative representations consist of a starting shape and a set of rules for iteratively transforming the design. One of the earliest such examples is Frazer's work using shape transformation rules to rotate/stretch/grow/shear a starting shape (Frazer, 1995). Similar to Frazer's work is that of de Garis' augmented cellular automata (CA) (de Garis, 1992), in which each cell in the CA maintains the number of neighbor cells in the ON state in each of the four directions, North, East, West and South. More standard CAs are used in (Bentley & Kumar, 1999) for creating two-dimensional tessellating tile patterns and for creating patterns of cells in an isospatial grid (Bonabeau *et al.* , 2000). Rather than working in a grid, Shape Feature Generating Process (SFGP) grows designs by optimizing rules for the division of dots (metaphors for a cell) on the surface of a shape (Taura *et al.* , 1998) (Taura & Nagasaka, 1999). After development is complete, the final shape is formed by creating an outer surface using the density of dots to determine the distance from the initial shape. A more biologically based model is Eggenberger's method of growing three-dimensional shapes from an artificial genome using an artificial chemistry (Eggenberger, 1997). Designs are encoded in a linear string which consists of regulatory genes, for switching other genes in the genome, and structural genes, which encode for specific chemicals.

Explicit, generative representations consist of an algorithm for constructing a design and are typically implemented as a type of grammar. With Todd and Latham's Mutator, structures are defined by an expression in a geometrical construction language that specifies the shape, shape transformations, number of repetitions of a shape and angles between shapes. Initially only the evolution of parameters was possible (Todd & Latham, 1992), and then in later work the ability to change the grammar was added (Todd & Latham, 1999). Broughton, Coates and Jackson (Broughton *et al.* , 1997) (Coates *et al.* , 1999) use a Lindenmayer system (L-system) as the representation for evolving shapes in an isospatial grid and the Emergent Design group has used L-systems with attractors/repellers for evolving curved surfaces in a three-dimensional space (Testa *et al.* , 2000) (Hemberg *et al.* , 2001). Rosenman (Rosenman, 1996) (Rosenman, 1997) describes a hierarchical grammar for building two-dimensional, grid-based, floor plans which uses multiple evolutionary runs to evolve different levels of the design. Instead of constructing a final design shape from a number of simpler shapes, the work of Husbands et. al. (Husbands *et al.* , 1996) and Nishino et. al. (Nishino *et al.* , 2001) combines su-

Table 1
Properties of the different design representations.

	Combination	Control Flow		Abstraction	
System		Iter.	Cond.	Labels	Param.
Direct Non-generative					
(Baron *et al.* , 1999)	no	no	no	no	no
(Kane & Schoenauer, 1995)	no	no	no	no	no
Shape grammar systems	no	no	no	no	no
Indirect Non-generative					
(Bentley, 1996)	yes	no	no	no	no
(Bentley & Kumar, 1999), *explicit*	yes	no	no	no	no
(Funes & Pollack, 1998)	yes	no	no	no	no
Genetic Design (Roston, 1994)	yes	no	no	no	no
GENRE, non-generative (section 4.3)	yes	no	no	no	no
Implicit Generative					
(Bentley & Kumar, 1999), *implicit*	no	yes	yes	no	no
(Bonabeau *et al.* , 2000)	no	yes	yes	no	no
(de Garis, 1992)	no	yes	yes	no	no
(Eggenberger, 1997)	no	yes	yes	no	yes
(Frazer, 1995)	no	yes	yes	no	no
(Taura *et al.* , 1998)	no	yes	yes	no	no
Explicit Generative					
(Broughton *et al.* , 1997)	yes	no	no	yes	no
Emergent Design Group	yes	no	no	yes	no
GENRE, generative (section 4.2)	yes	yes	yes	yes	yes
Mutator (Todd & Latham, 1992)	yes	yes	no	no	no
(Rosenman, 1997)	yes	no	no	yes	no
Superquadrics	yes	no	no	yes	no

perquadric modeling primitives (Barr, 1981) with constructive solid geometry as a kind of genetic program for transforming a starting shape.

The evolutionary design systems described in this section are listed in table 1. In this table the representations are grouped by category and for each representation it is stated whether or not it has the property of combination, iteration, conditionals, labels or parameters.

4 Evolutionary Design System

The computer-automated design system used to create designs is called *GENRE* and it consists of the design constructor, the compiler for the generative representation, the fitness function for evaluating designs and the evolutionary algorithm. The evolutionary algorithm evolves encoded designs using the fitness function to score designs. To allow for comparing non-generative and generative representations, GENRE has both a non-generative and a generative representation for encoding designs. The non-generative representation encodes designs indirectly using a sequence of construction commands, called an assembly procedure, for building a design with the design constructor. The generative representation is based on Lindenmayer systems (Lindenmayer, 1968), which are compiled into an assembly procedure that is used to build the design. By using an indirect, non-generative representation and an explicit, generative representation, these two representations can be applied to different design substrates by changing only the set of construction commands and the design constructor. The following subsections describe each of these parts.

4.1 Design Constructor

The design constructor starts with a single cube with which it creates a more complex object by executing the instructions of an assembly procedure. Commands in the command set act on the local state – which consists of the current position and orientation – and are listed in table 2. The commands '`[`' and '`]`' push and pop the current state to and from a stack. `Forward` adds cubes in the positive X direction of the local state and `back` adds cubes in the negative X direction. In addition to adding cubes `forward` and `back` also change the current position. The commands `left/right/up/down/clockwise/counter-clockwise` rotate the current heading about the appropriate axis in units of $90°$.

The images in figure 1 show intermediate stages in the construction of an object from the assembly procedure: `forward(2) right(1) forward(1) up(1) forward(3)`. Initially there is a single cube in the design space, figure 1.a. After executing the command `forward(2)`, two cubes are added to the first, figure 1.b. The image in figure 1.c shows the design after executing `right(1) forward(1)`, which turns the current orientation $90°$ to the right and then adds a

Table 2
Design language for constructing tables.

Command	Description
[ ]	Push/pop state to stack.
forward(n)	Move and add cubes in the local, positive X direction n units.
back(n)	Move and add cubes in the local, negative X direction n units.
clockwise(n)	Rotate local heading $n \times 90°$ about the X axis.
counter-clockwise(n)	Rotate local heading $n \times -90°$ about the X axis.
left(n)	Rotate local heading $n \times 90°$ about the Y axis.
right(n)	Rotate local heading $n \times -90°$ about the Y axis.
up(n)	Rotate local heading $n \times 90°$ about the Z axis.
down(n)	Rotate local heading $n \times -90°$ about the Z axis.

cube in the current forward direction. After executing `up(1) forward(3)`, the final object is shown in figure 1.d. In building an object, if the constructor is asked to place a cube where one already exists, it ignores the existing cube but updates its location as if it had placed this cube and then continues executing the assembly procedure.

4.2 Generative Representation

The generative representation for each design is based on a grammatical rewriting system called Lindenmayer Systems (L-systems) (Lindenmayer, 1968). The class of L-systems used as the encoding for designs in this work is parametric L-systems. Production rules for a parametric L-system consist of a rule-head, which is the symbol to be replaced, followed by a number of condition-successor pairs. The condition is a boolean expression on the parameters to the production-rule, and the successor consists of a sequence of characters that replace the rule-head. Rule-head symbols are re-written by testing each of their conditions sequentially, and replacing the rule-head symbol with the successor of the first condition that succeeds. Thus the parametric L-system has the properties of combination, abstraction, naming of compound procedures, formal parameters to the procedures, and conditionals. To handle iteration, a looping ability is added that replicates the symbols enclosed with parenthesis similar to `for` loops in computer programs: $\{\ block\ \}(n)$ repeats the enclosed block of symbols n times. For example $\{abc\}(3)$ translates to, $abcabcabc$.

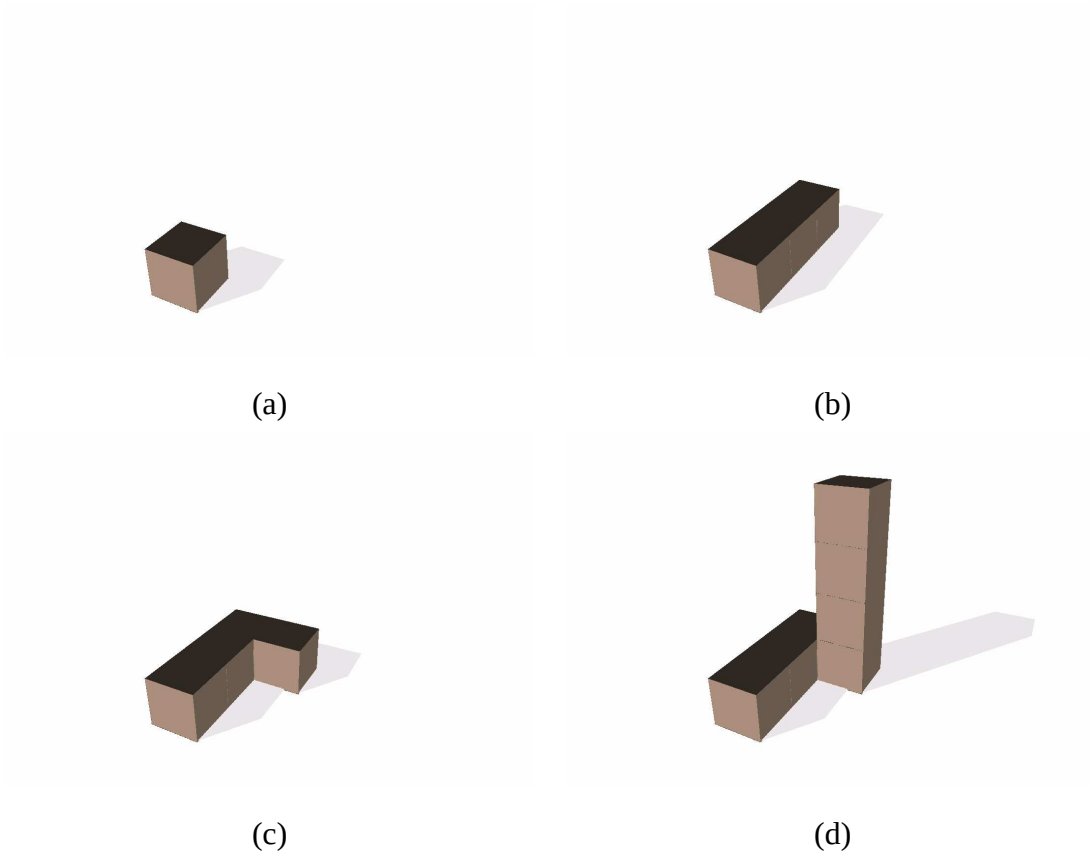


Fig. 1. Steps in building an object.

Compiling an L-system into an assembly procedure consists of starting with a single symbol and then iteratively applying the production rules in parallel to all commands in an assembly procedure. The following is an example design encoded with the generative representation and the construction language of table 2. It consists of two productions with each production containing one condition-successor pair:

$$P0(n_0): \; n_0 > 1.0 \rightarrow [\; P1(n_0 * 1.5) \;]\; up(1)\; forward(3)\; down(1)\; P0(n_0 - 1)$$

$$P1(n_0): \; n_0 > 1.0 \rightarrow \{\; [\; forward(n_0) \;]\; left(1) \;\}(4)$$

Compiling this design encoding starting with the symbol P0(4) produces the following sequence of strings,

1. *P0(4)*
2. *[P1(6)] up(1) forward(3) down(1) P0(3)*
3. *[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [P1(4.5)] up(1) forward(3) down(1) P0(2)*
4. *[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(4.5)] left(1) }(4)] up(1) forward(3) down(1) [P1(3)] up(1) forward(3) down(1) P0(1)*
5. *[{ [forward(6)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(4.5)] left(1) }(4)] up(1) forward(3) down(1) [{ [forward(3)] left(1) }(4)] up(1) forward(3) down(1)*
6. *[[forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1) [forward(6)] left(1)] up(1) forward(3) down(1) [[forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1) [forward(4.5)] left(1)] up(1) forward(3) down(1) [[forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1) [forward(3)] left(1)] up(1) forward(3) down(1) forward(3)*

This final string is a sequence of construction commands which is then used by the design constructor to build an object.

4.3 Non-generative Representation

To show the advantages of a generative representation it must be compared against a non-generative representation. For the non-generative representation each individual in the population is an assembly procedure which specifies how to construct the design. We implement this assembly procedure as a degenerate, parametric L-system which has only a single rule and no iterative loops or abstraction. Implementing the non-generative representation in the same way as the generative representation allows us to use the same evolutionary algorithm and variation operators so that the only difference between evolutionary runs with the two systems is the ability to hierarchically reuse elements of encoded designs.

4.4 Evolutionary Algorithm

An evolutionary algorithm is a population-based, search algorithm used in optimization. Search operates by creating an initial population of candidate designs, called individuals, and then iteratively selecting better individuals to reproduce and make new designs until the search is done. The evolutionary algorithm and variation operators used by GENRE are described in detail in (Hornby, 2003), here we give an overview of the system.

The evolutionary algorithm used to evolve designs is the canonical generational EA with specialized variation operators. Each individual in the initial population is an L-system with a random set of production rules. After all individuals have been evaluated, better individuals are selected as parents to create a new population. New individuals are created through applying mutation or recombination (chosen with equal probability) to individuals selected as parents. Mutation takes a single individual as a parent, makes a copy of it and then makes a small random change to this child copy. Some of the changes that can occur are inserting a small sequence of random commands, deleting a small sequence of commands, changing the parameters to a command, changing the parameters of a conditional, and encapsulating a sequence of commands into its own production rule. Recombination takes two individuals as parents, makes a copy of the first individual and then inserts a small part of the second parent into this child. Examples of some of the insertions that can be done are replacing a subsequence of commands in the child with a subsequence of commands from the second parent, replacing one of the child's successors with one from the second parent, and replacing a complete condition-successor pair in the child with one from the second parent. This process of evaluation, selection, and reproduction is then repeated for a fixed number of generations.

To reduce the frequency of variations that do not result in a change in the resulting design, data is kept for compiled L-systems on which production rules and successors were used, as well as the value range for each parameter. This compilation history is used so that variation operators are then applied only to those production rules that were used in compiling and so that mutated of condition values stay within the value range of the parameter being compared against.

4.5 Fitness Function

Once an assembly procedure is executed the resulting structure is evaluated by a pre-specified *fitness function*. First, the design simulator determines whether or not the object is balanced or will fall over. Objects that are not balanced are given a fitness of zero, otherwise an object is given a score based on several easy to compute attributes of a table. These attributes are the height at which it holds objects, the amount of surface area available, how stable it is, and the amount of material it is made of. Height is simply the number of cubes above the ground and surface area is the number of cubes at the maximum height. Rather than measuring actual structural integrity, a rough measure of stability is calculated by using the volume of the table from summing the area at each layer of the table. The measurement of amount of material used includes only those cubes not on the surface and is necessary since maximizing height, surface area and stability typically result in table designs that are solid volumes.

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

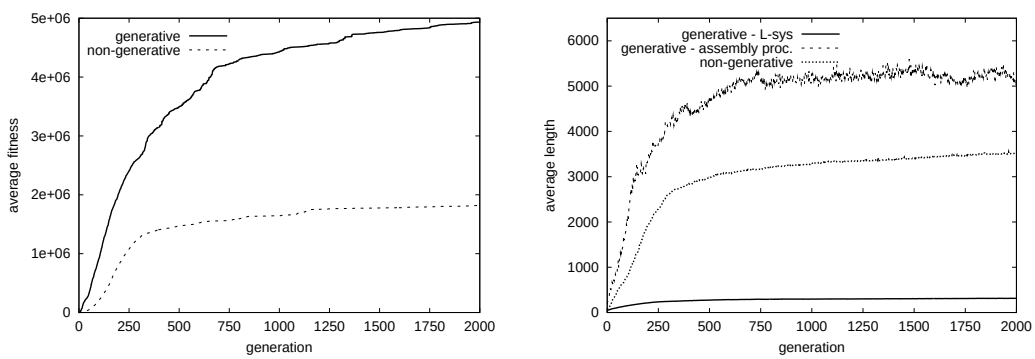
$$f_{surface} = \text{the number of cubes at } Y_{max}.$$
$$f_{stability} = \sum_{y=0}^{Y_{max}} f_{area}(y)$$
$$f_{area}(y) = \text{area in the convex hull at height y.}$$
$$f_{excess} = \text{number of cubes not on the surface.}$$

For these experiments we combine these measures into a single function,

$$\text{fitness} = f_{height} \times f_{surface} \times f_{stability} / f_{excess} \tag{1}$$

5 Results

To compare the generative representation against the non-generative representation the evolutionary algorithm was configured to run for two thousand generations using a population size of two hundred. In this section we present results comparing fitness and evolvability of designs produced with both representations and also show tables constructed from evolved designs.



(a) (b)

Fig. 2. Graphs comparing (a) average fitness and (b) average length of the design encodings and assembly procedure.

The graph in figure 2.a contains a comparison of the fitness of the best individuals evolved with the non-generative representation against the best individuals evolved with the generative representation, averaged over fifty trials. With the non-generative representation, fitness improved rapidly over the first 300 generations, then quickly leveled off, improving by less than 25% over the last 1700 generations. Fitness increased faster with the generative representation, and the rate of increase in fitness did not slow as quickly as with the non-generative representation. The final results are an average best fitness of 1826158 with the non-generative representation and 4938144 with the generative representation.

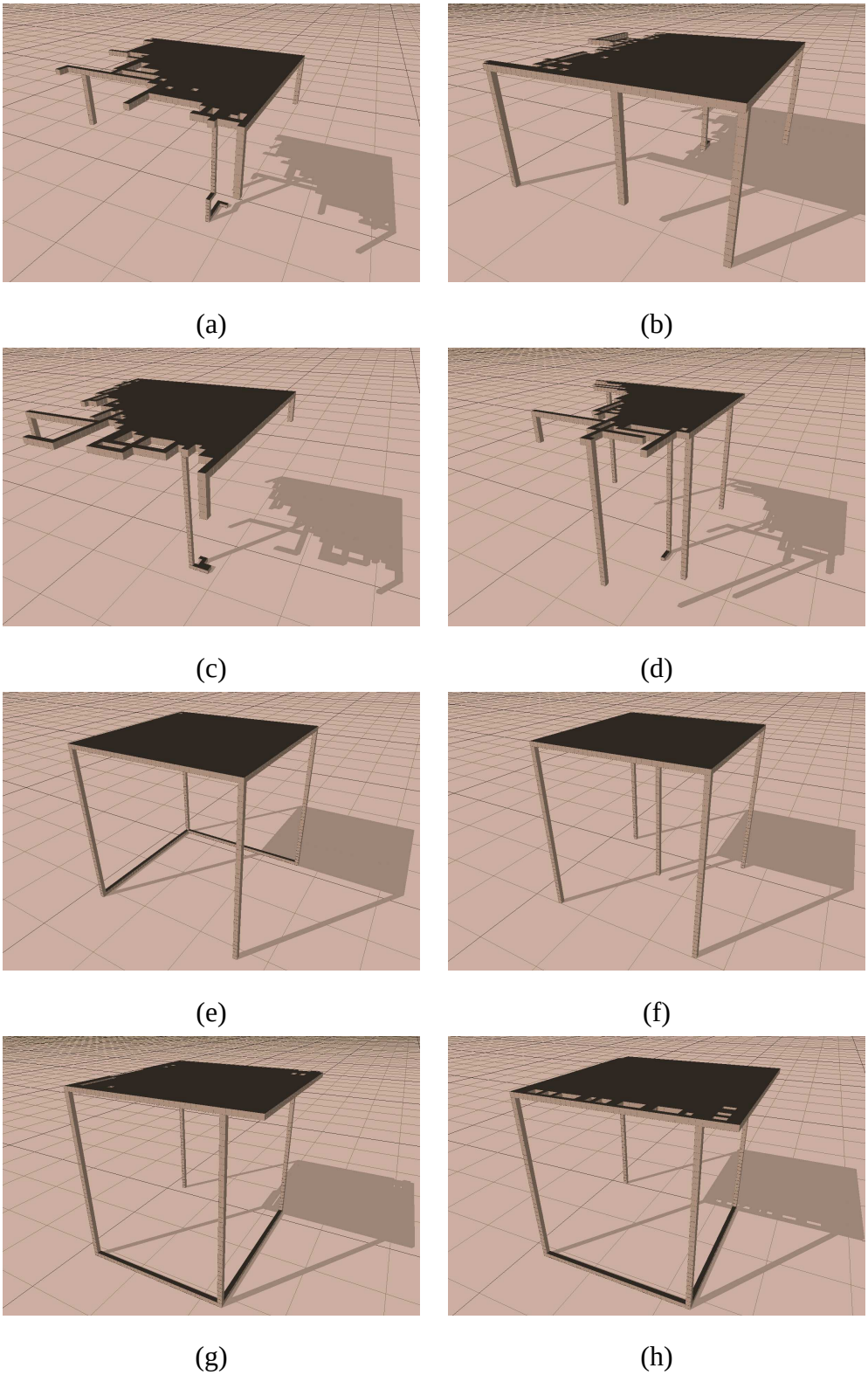
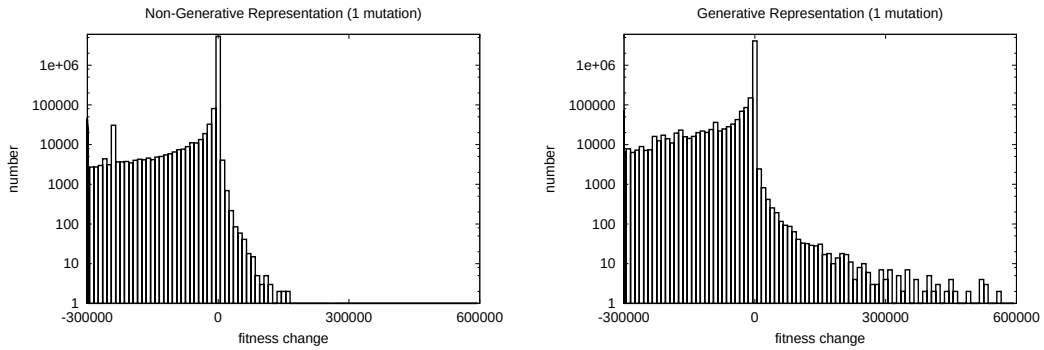
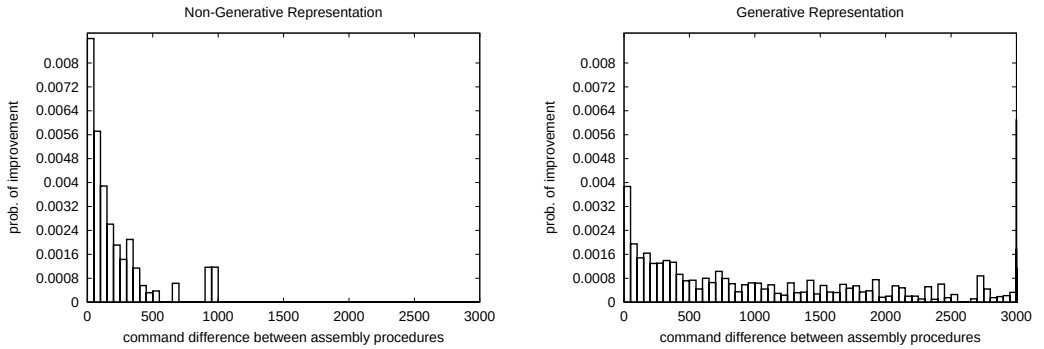


Fig. 3. The four best tables evolved with: (a)-(d), the non-generative representation; and (e)-(h), the generative representation.



(a) (b)

Fig. 4. Graph of the number of offspring (y-axis, log scale) that had a given fitness differential (x-axis) from their parent.



(a) (b)

Fig. 5. Probability of success (child is more fit than parent) comparison between non-generative and generative representations, for ranges 1-50, 51-100, 101-150, ... 1951-2000

In addition to having higher fitness, tables evolved with the generative representation show some symmetries and regularities whereas tables evolved with the non-generative representation tended to be irregular, figure 3. These regularities are most likely a result of the generative representation's ability to reuse elements. The average amount of reuse with the generative representation can be calculated from the average length of the encoded design and the average length of the assembly procedure it compiles to. From the graph in figure 2.b it can be seen that these values are approximately 310 and 5100 respectively, which leads to an average reuse of just over sixteen elements.

To determine if the generative representation produces encodings that are more conducive to evolution we compare the success rate of the mutation operator. The graphs in figure 4 plot the number of offspring that fall at a given fitness differential from the parent design. These graphs show that the vast majority of mutations to a design produced little or no change in fitness. While most of the remaining mutations produced a negative change in fitness with both representations, there are

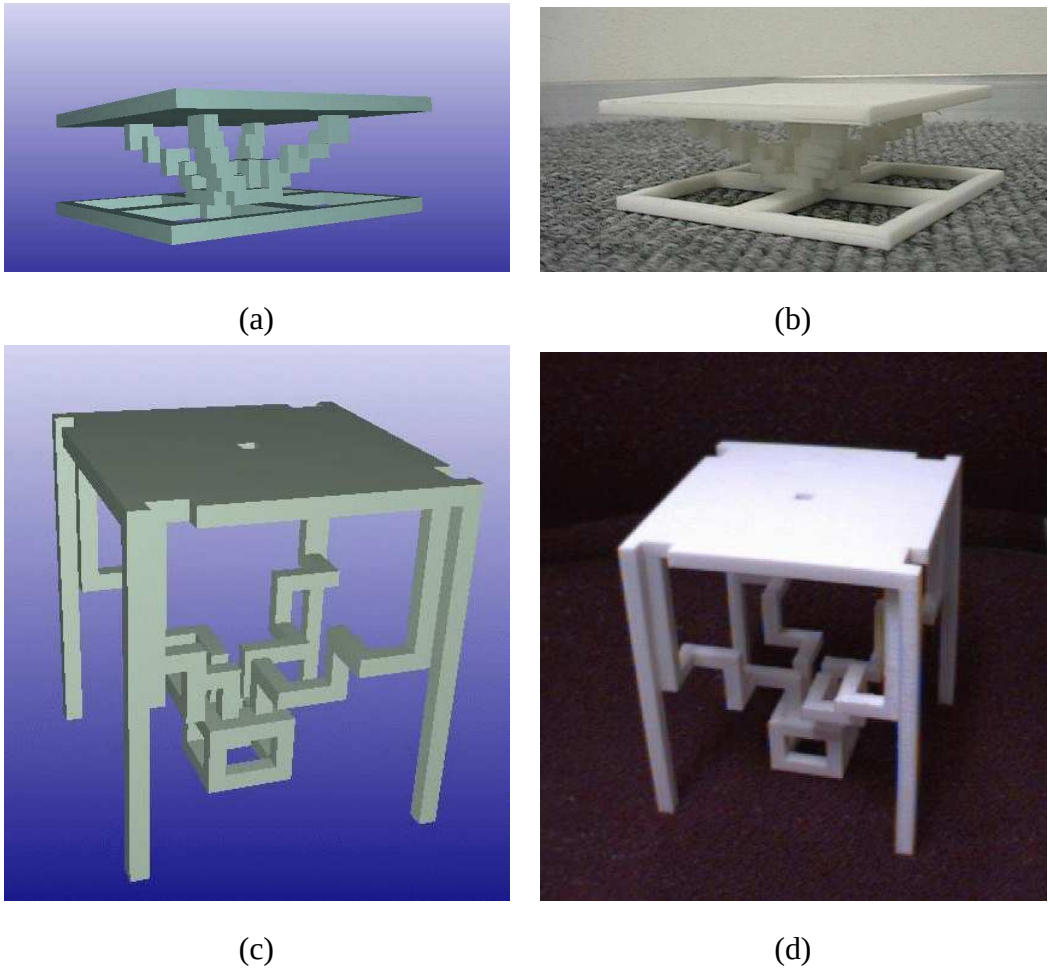


Fig. 6. Evolved tables shown both in simulation (left) and reality (right).

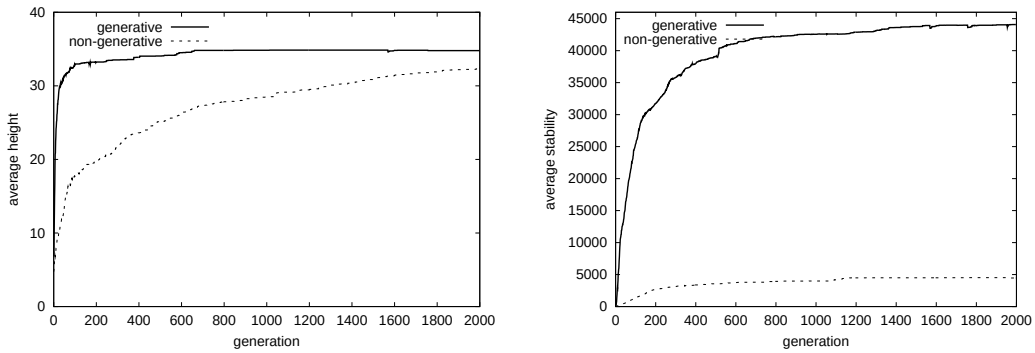
more positive changes to fitness with the generative representation, especially large positive changes. A plot of the success rate under mutation (a child has higher fitness than its parent) is shown in the graphs in figure 5. On this graph it can be seen that mutations that produce a large change in the assembly procedure have a greater rate of success with the generative representation than with the non-generative representation. These two sets of graphs show that designs encoded with the generative representation are more evolvable than those encoded with the non-generative representation.

Previous work has shown the successful transfer from design to reality of static objects (Funes & Pollack, 1998) and robots (Lipson & Pollack, 2000). Similarly, designs produced with this system have also been successfully transferred to the real world using rapid-prototyping equipment, figure 6.

6 Discussion

In section 2 it was argued that reusing elements of an encoded design for multiple parts in the actual design improves evolvability by capturing design dependencies and by improving the ability to make large changes in design space. This section consists of two parts that give evidence supporting both of these claims.

6.1 Design Dependencies



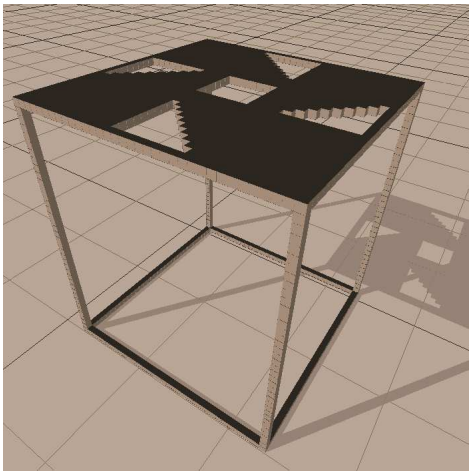
(a) (b)

Fig. 7. Graphs of (a) average stability and (b) average height over the course of evolution.

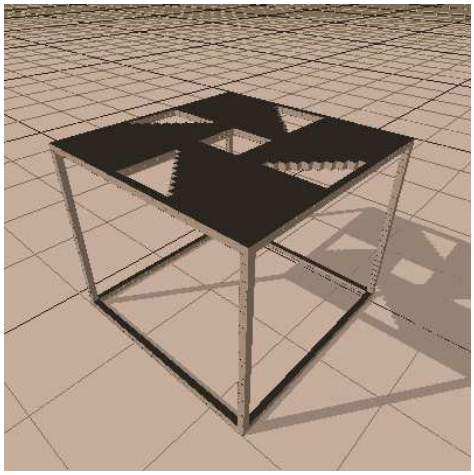
One way to determine which representation better captures design dependencies is to compare the designs that are evolved with them. With table designs, the main dependency is with the table legs: in maximizing the height of a table, the optimal length of each leg is dependent on the lengths of the other table legs. First, it can be seen from the graph in figure 7.a that evolution with the generative representation is both faster at producing high tables and also produces higher overall tables than evolution with the non-generative representation. Next, the low stability score of tables evolved with the non-generative representation (figure 7.b) along with the images in figure 3 suggest that these tables typically have only a single leg stretching from table top to the ground whereas the high stability score and images in figure 3 suggest that tables evolved with the generative representation have multiple table legs. That evolution with the generative representation is better able to produce multi-legged tables suports our argument that generative representations are better able to capture and manipulate design dependencies.

6.2 Coverage of the Search Space

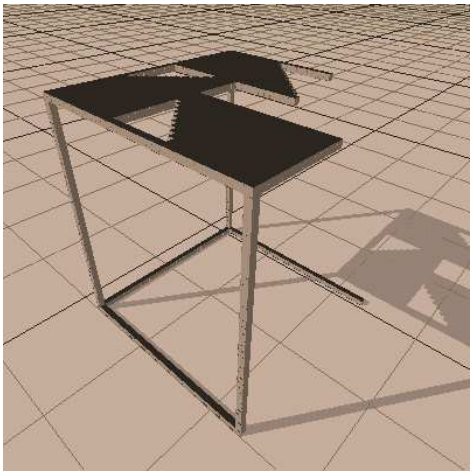
Better capturing dependencies and reusable building blocks which can be easily added/removed enables variation operators to move through the design space in



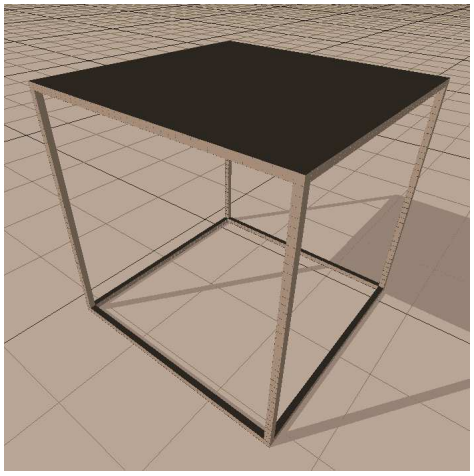
(a) Original.



(b) Shorter.



(c) Three corners



(d) More surface cubes.

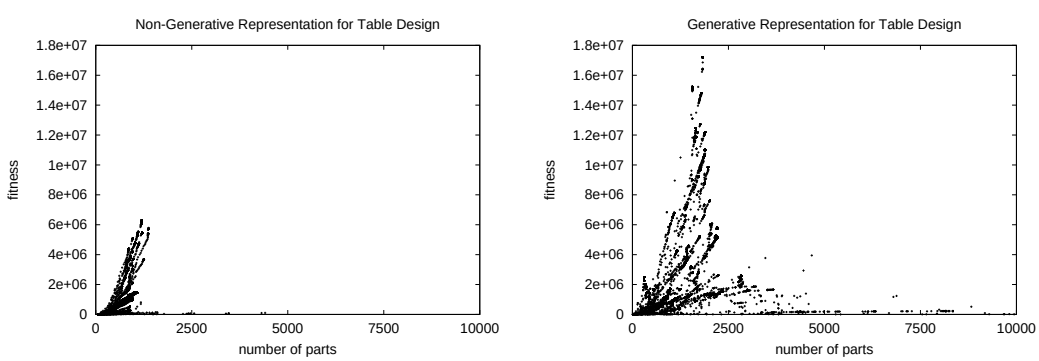
Fig. 8. Table mutants.

more meaningful ways. For an example we use the table in figure 8.a, for which the generative representation is listed in appendix A. The height of the table legs is encoded in the first successor of *P8* and changing the values of either *back()* command will change the height of the table, figure 8.b. Similarly, changing the iteration counter in the second successor of production rule *P0* from }*(4)* to }*(3)* changes the number of corners and legs on the table from four to three, figure 8.c. Finally, the number of cubes on the surface of the table can be changed by changing the parameter values of any of the last three *back()* commands in the second successor of *P6*. In this case, further evolution changed the production rules $P6$ and $P8$ to,

P6(n_0, n_1)	(n_1>1) →	[back(5) up(5) back(n_0) left(1) back(5) back(5) back(5) back(4)]
	(n_0>2) →	[back(5) up(5) back(n_0) back(5) left(1) down(5) up(5) back(5) back(5) back(5) back(4)]
	(n_1>0) →	[back(5) up(5) left(1) back(n_0) back(5) down(5) up(1) back(5) back(5) back(5) back(4)]
P8(n_0, n_1)	(n_0>0) →	P8($n_0/5$,n_1+1) [back(4) back(4) P8(n_1-2,n_0-5)]
	(n_1>-2) →	[P8($n_0/4$,n_1+1) back(5) back(4) P8(2-5,3-5) back(4) back(5) P6(n_1-n_0,n_0+n_1)]
	(n_0>-1) →	counter-clockwise(1) down(3) down(n_0)

with the resulting table shown in figure 8.d. All three of these example show how through the process of evolution the generative representation has evolved an encoding with which it is easy for the mutation operator to make large, meaningful movements in the design space.

With a non-generative representation, all of those changes to the table in figure 8.a would require the simultaneous change of multiple symbols in the encoding. Some of these changes must be done simultaneously for the resulting design to be viable – changing the height of only one leg of the table can result in a significant loss of fitness – and so these changes are not evolvable with a non-generative representation. Others, such as the number of cubes on the surface, are viable with a series of single-cube changes. Yet, in general this could result in a significantly slower search speed in comparison with a single change to a table encoded with a generative representation.



(a) (b)

Fig. 9. Plots of number of parts versus fitness of the best individual from each trial: (a) non-generative representation; and (b) generative representation.

That this improved ability make large, meaningful changes to a design is resulting in an increase in this size of the design space that is explored can be seen by

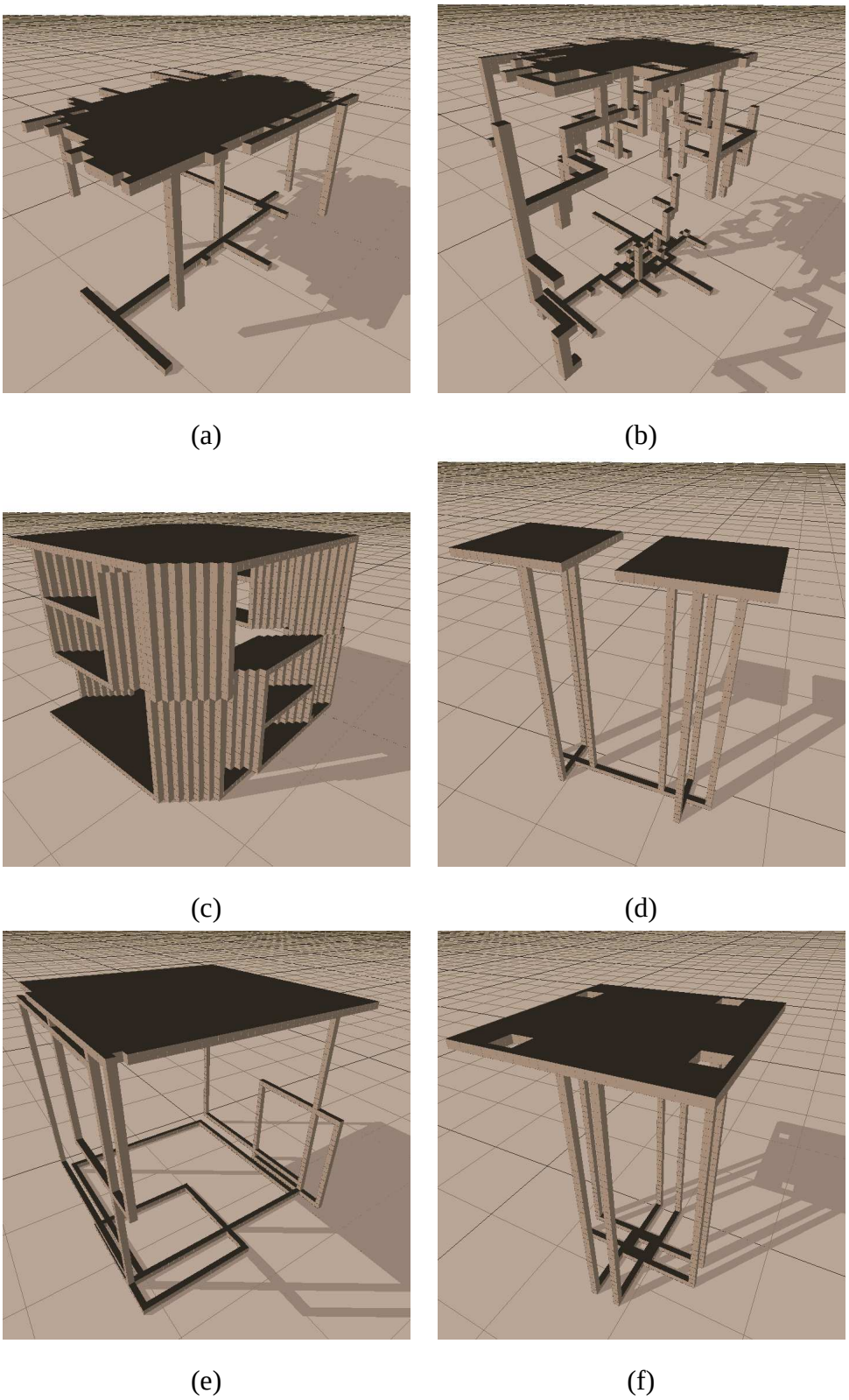


Fig. 10. Other tables evolved using: (a)-(b) the non-generative representation; and (c)-(f) the generative representation.

the designs evolved with the two representations. The graphs in figure 9 are plots of the number of parts in the design against their fitness with the non-generative and generative representations. This graph shows that search with the generative representation tested designs with a wider range of parts and fitness than with the non-generative representation. Using variations of the fitness function described in section 4.5, examples of the different types of tables that have been evolved with the non-generative and generative representations are shown in figure 10. In general, tables evolved with the non-generative representation are similarly irregular as the ones in figure 3 whereas tables evolved with the the generative representation are more varied in how the table-top is supported above the ground. These designs show that search with the generative representation is exploring a larger, more interesting part of the table design space than is search with the non-generative representation.

7 Conclusion

The complexity of designs achievable with computer-automated design systems is strongly limited by their representation. Here we defined several classes of design representations and then argued that the generative representations are a better method for encoding designs than non-generative representations. To support these arguments we compared a generative and non-generative representation on a table design problem and found that designs evolved with the generative representation have higher fitness and a more regular structure. In addition, we argued that generative representations are better able to capture and manipulate design dependencies and supported this by showing that evolution tended to evolve multi-legged tales with the generative representation and single-legged tables with the non-generative representation. Finally, we claimed that a generative representation enables meaningful, large scale design changes and gave examples of this with an evolved table design as well as showed that evolution with the generative representation produced a wider range of table styles. The next step in automated design is in producing design representations that can hierarchically create and reuse assemblies of parts in ever more powerful ways.

Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Administration (DARPA) Grant, DASG60-99-1-0004.

References

Agarwal, M., & Cagan, J. 1998. The Language of Coffee Makers. *Environment and Planning B: Planning and Design*, **25**(2), 205–226.

Angeline, P., & Pollack, J. B. 1994. Coevolving High-Level Representations. *Pages 55–71 of:* Langton, C. (ed), *Proceedings of the Third Workshop on Artificial Life*. Reading, MA: Addison-Wesley.

Baron, P., Tuson, A., & Fisher, R. 1999. A Voxel-Based Representation for Evolutionary Shape Optimisation. *Journal of Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Special Issue on Evolutionary Design*, **13**(3), 145–156.

Barr, A. 1981. Superquadrics and angle preserving transformations. *IEEE Computer Graphics and Applications*, **1**(1), 11–23.

Bentley, P., & Kumar, S. 1999. Three Ways to Grow Designs: A Comparison of Embryogenies of an Evolutionary Design Problem. *Pages 35–43 of:* Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (eds), *Genetic and Evolutionary Computation Conference*. Morgan Kaufmann.

Bentley, P. J. 1996. *Generic Evolutionary Design of Solid Objects Using a Genetic Algorithm*. Ph.D. thesis, Dept. of Engineering, University of Huddersfi eld.

Bentley, P. J. (ed). 1999. *Evolutionary Design by Computers*. San Francisco: Morgan Kaufmann.

Bentley, P. J., & Corne, D. W. (eds). 2001. *Creative Evolutionary Systems*. San Francisco: Morgan Kaufmann.

Bonabeau, E., Gurin, S., Snyers, D., Kuntz, P., & Theraulaz, G. 2000. Three-dimensional architectures grown by simple 'stigmergic' agents. *BioSystems*, **56**(1), 13–32.

Broughton, T., Tan, A., & Coates, P. S. 1997. The Use of Genetic Programming in Exploring 3d Design Worlds. *Pages 885–917 of:* Junge, R. (ed), *CAAD Futures 1997*. Kluwer Academic.

Coates, P., Broughton, T., & Jackson, H. 1999. Exploring Three-Dimensional Design Worlds using Lindenmayer Systems and Genetic Programming. *Chap. 14 of:* Bentley, P. J. (ed), *Evolutionary Design by Computers*. San Francisco: Morgan Kaufmann.

de Garis, H. 1992. Artifi cial Embryology : The Genetic Programming of an Artifi cial Embryo. *Chap. Ch. 14 of:* Soucek, Branko, & the IRIS Group (eds), *Dynamic, Genetic and Chaotic Programming*. Wiley.

Drexler, K. E. 1989. Biological and Nanomechanical Systems. *Pages 501–519 of:* Langton, C.G. (ed), *Artificial Life*. Addison Wesley.

Eggenberger, P. 1997. Evolving Morphologies of Simulated 3d Organisms Based on Differential Gene Expression. *Pages 440–448 of:* Husbands, P., & Harvey, I. (eds), *Proc. of the 4rth European Conf. on Artificial Life*. Cambridge: MIT Press.

Frazer, J. 1995. *An Evolutionary Architecture*. Architectural Association Publica-

tions.

Funes, P., & Pollack, J. B. 1998. Evolutionary Body Building: Adaptive physical designs for robots. *Artificial Life*, **4**(4), 337–357.

Hemberg, M., O'Reilly, U.-M., & Nordin, P. 2001. GENR8: A Design Tool for Surface Generation. *In: Late Breaking paper at the Genetic and Evolutionary Computation Conference*. AAAI.

Hornby, G. S. 2003. *Generative Representations for Evolutionary Design Automation*. Ph.D. thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA.

Hornby, G. S., & Pollack, J. B. 2001. The Advantages of Generative Grammatical Encodings for Physical Design. *Pages 600–607 of: Congress on Evolutionary Computation*. IEEE Press.

Hornby, G. S., & Pollack, J. B. 2002. Creating High-level Components with a Generative Representation for Body-Brain Evolution. *Artificial Life*, **8**(3), 223–246.

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

Husbands, P., Germy, G., McIlhagga, M., & Ives, R. 1996. Two Applications of Genetic Algorithms to Component Design. *Pages 50–61 of:* Fogarty, T. (ed), *Evolutionary Computing. LNCS 1143*. Springer-Verlag.

Kane, C., & Schoenauer, M. 1995. Genetic Operators for Two-Dimentional Shape Optimization. *Pages 355–369 of:* Alliot, J.-M., Lutton, E., Ronald, E., Schoenauer, M., & Snyers, D. (eds), *Artificiale Evolution - EA95*. Springer-Verlag.

Kane, C., & Schoenauer, M. 1996. Topological Optimum Design. *Control and Cybernetics*, **25**(5), 1059–1088.

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

Lindenmayer, A. 1968. Mathematical Models for Cellular Interaction in Development. Parts I and II. *Journal of Theoretical Biology*, **18**, 280–299 and 300–315.

Lipson, H., & Pollack, J. B. 2000. Automatic Design and Manufacture of Robotic Lifeforms. *Nature*, **406**, 974–978.

Lohn, J. D., Hornby, G. S., & Linden, D. S. in press. An Evolved Antenna for Deployment on NASA's Space Technology 5 Mission. *Chap. 18 of:* O'Reilly, U.-M., Riolo, R. L., Yu, T., & Worzel, B. (eds), *Genetic Programming Theory and Practice II*. Kluwer.

Meyer, B. 1988. *Object-oriented Software Construction*. New York: Prentice Hall.

Michalewicz, Z., Dasgupta, D., Riche, R. G. Le, & Schoenauer, M. 1996. Evolutionary Algorithms for Constrained Engineering Problems. *Computers and Industrial Engineering Journal*, **30**(2), 851–870.

Nishino, H., Takagi, H., Cho, S.-B., & Utsumiya, K. 2001. A 3D Modeling System for Creative Design. *Pages 479–486 of: 15th Intl. Conf. on Information Networking*. Beppu, Japan: IEEE.

Pollack, J. B., Lipson, H., Hornby, G., & Funes, P. 2001. Three Generations of Automatically Designed Robots. *Artificial Life*, **7**(3), 215–223.

Robinson, G., El-Beltagy, M., & Keane, A. 1999. Optimization in Mechanical

Design. *Chap. 6, pages 147–165 of:* Bentley, P. J. (ed), *Evolutionary Design by Computers*. San Francisco: Morgan Kaufmann.

Rosenman, M. 1996. A growth model for form generation using a hierarchical evolutionary approach. *Microcomputer in Civil Engineering*, **11**, 161–172.

Rosenman, M. A. 1997. The generation of form using an evolutionary approach. *Pages 69–85 of:* Dasgupta, D., & Michalewicz, Z. (eds), *Evolutionary Algorithms in Engineering Applications*. Southampton: Springer-Verlag.

Roston, G. P. 1994 (December). *A Genetic Methodology for Configuration Design*. Ph.D. thesis, Dept. of Mechanical Engineering, Carnegie Mellon University.

Shea, K., Cagan, J., & Fenves, S. J. 1997. A Shape Annealing Approach to Optimal Truss Design With Dynamic Grouping of Members. *Journal of Mechanical Design*, **119**(September), 388–394.

Stiny, G. 1980. Introduction to Shape and Shape Grammars. *Environment and Planning B: Planning and Design*, **7**, 343–351.

Taura, T., & Nagasaka, I. 1999. Adative-growth-type 3D Geometric Representation for Spatial Design. *Journal of Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **13**(3), 171–184.

Taura, T., Nagasaka, I., & Yamagishi, A. 1998. An application of evolutionary programming to shape design. *Computer-Aided Design*, **30**(1), 29–35.

Testa, P., O'Reilly, U.-M., Kangas, M., & Kilian, A. 2000. MoSS: Morphogenetic Surface Structure - A Software Tool for Design Exploration. *Pages 71–80 of: Proceedings of Greenwich 2000: Digital Creativity Symposium*. University of Greenwich.

Todd, S., & Latham, W. 1992. *Evolutionary Art and Computers*. Academic Press.

Todd, S., & Latham, W. 1999. The Mutation and Growth of Art by Computers. *Chap. 9, pages 221–250 of:* Bentley, P. J. (ed), *Evolutionary Design by Computers*. San Francisco: Morgan Kaufmann.

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

A Generative Representation for an Evolved Table

The following L-system is the generative representation for the table in fi gure 8.a:

P0(n_0, n_1) (n_1 >10) → P11(n_0/4,2) down(1) {P17(3,n_1/2) P12(n_1+n_0,n_0+n_1) P3(1,n_1-n_0) }(4)

(n_1>3) → P11(n_0/4,2) down(1) {P17(3,n_1/2) P18(n_1+n_0,n_0+n_1) P3(1,n_1-n_0) }(4)

(n_1>0) → [P16(2,n_0+2)]

P2(n_0, n_1) (n_0>5) → P7(n_1/2,n_0+2) back(1)

(n_0>5) → P7(n_1/2,n_0+2) back(1)

(n_0>0) → [left(4) P12(3,4)]

P3(n_0, n_1) (n_1>2) → P16(4,n_0-n_1) P16(4,n_0-n_1) P16(4,n_0-2) P16(4,n_0-n_1)

(n_1>2) → P16(4,n_0-n_1) P16(4,n_0-n_1) P16(4,n_0-n_1) P16(4,n_0-n_1)

(n_0>0) → down(1) {clockwise(n_1) forward(3) }(5)

P6(n_0, n_1) (n_1>1) → [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]

(n_0>1) → [back(5) up(1) back(n_0) left(1) back(n_0) down(5) up(1) back(5) left(n_1) back(5) down(1)]

(n_1>0) → [back(5) left(1) back(5) down(1) up(1) back(5) back(5) back(5)]

P7(n_0, n_1) (n_0>-1) → [clockwise(1) clockwise(1) left(1) clockwise(1) clockwise(5) right(1) clockwise(1)] down(1)

(n_1>1) → [clockwise(1) clockwise(1) left(1) clockwise(1) clockwise(5) clockwise(1) clockwise(1)] down(1)

(n_0>0) → [clockwise(1) left(1)] right(2) up(2) P18(n_1-5,n_1+4)

P8(n_0, n_1) (n_0>0) → P8(n_0/4,n_1+1) [back(4) back(4) P8(n_1-2,n_0-5)]

(n_1>-2) → [P8(n_0/4,n_1+1) back(5) P8(n_1-5,n_0-5) back(4) back(4) back(4) P6(n_1-n_0,n_0+n_1)]

(n_0>0) → [back(4) P8(n_1-2,n_0-5)] P6(n_1-n_0,n_0+n_1)

P9(n_0, n_1) (n_1>3) → P7(3-3,n_0+n_1) clockwise(1) P8(n_1-n_0,n_1+1)

(n_1>2) → P7(3-3,n_0+n_1) clockwise(1) P8(n_1-n_0,n_1+1)

(n_1>0) → forward(1) P16(n_1-1,n_0-n_1)

P11(n_0, n_1) (n_1>4) → [P19(4,5)]

(n_1>-10) → right(1)

(n_1>0) →

P12(n_0, n_1)	(n_1>4) →	back(1) up(1)
	(n_1>4) →	back(1) up(1)
	(n_1>0) →	counter-clockwise(4)
P14(n_0, n_1)	(n_1>3) →	P11(n_1,n_0/n_1)
	(n_0>10) →	P2($n_1/3$,n_1+n_0) P9(n_1,n_0/n_1)
	(n_1>0) →	P9(n_1,n_0/n_1)
P16(n_0, n_1)	(n_1>22) →	
	(n_1>5) →	
	(n_1>0) →	[P19($n_0/2$,n_1-n_0) back(1) forward(3)] clockwise(2) P3(5,n_1-5)
P17(n_0, n_1)	(n_1>3) →	up(n_1) back(2) back(4) back(n_0) back(3) back(3) back(5)
	(n_1>3) →	up(n_1) back(4) back(2) back(2) back(2) back(3) back(5)
	(n_1>0) →	up(2) back(2)
P18(n_0, n_1)	(n_1>3) →	back(n_1) P14(n_1-3,3) left(1) left(1) right(1) right(1)
	(n_1>3) →	back(n_1) P14(n_1-2,3) left(1) [counter-clockwise(1)] right(1)
	(n_0>0) →	[P13(5,$n_1/4$) right(1)] right(1)

This L-system is started with P0(4, 10) and run for 17 iterations, after which it produces an assembly procedure comprising of several thousand commands. The sequence of commands starts as follows:

right(1) down(1) up(5) back(2) back(4) back(3) back(3) back(3) back(5) back(14) [left(4) counter-clockwise(4)] [clockwise(1) clockwise(1) left(1) clockwise(1) clockwise(5) right(1) clockwise(1)] down(1) clockwise(1) [back(4) back(4)] [back(4) back(4) [back(4) back(4)]] [back(4) back(4) [back(4) back(4)] [back(4) back(4) [back(5) back(4) back(4) back(4)]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4) [[back(5) back(4) back(4) back(4)] back(5) [back(4) back(4)] back(4) back(4) back(4) [back(5) up(1) back(13.7) left(1) back(13.7) down(5) up(1) back(5) left(-0.3) back(5) down(1)]]]] [back(4) back(4) [back(4) back(4)] [back(4) back(4)] [back(4) back(4)] . . .

The entire assembly procedure is listed in appendix A of (Hornby, 2003).