Generative Representations for Computer-Automated Evolutionary Design

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

Abstract

With the increasing computational power of computers, software design systems are progressing from being tools for architects and designers to express their ideas to tools capable of creating designs under human guidance. One of the main limitations for these computer-automated design systems is the representation with which they encode designs. If the representation cannot encode a certain design, then the design system cannot produce it. To be able to produce new types of designs, and not just optimize pre-defined parameterizations, evolutionary design systems must use generative representations. Generative representations are assembly procedures, or algorithms, for constructing a design thereby allowing for truly novel design solutions to be encoded. In addition, by enabling modularity, regularity and hierarchy, the level of sophistication that can be evolved is increased. We demonstrate the advantages of generative representations on two different design domains: the evolution of spacecraft antennas and the evolution of 3D objects.

1 Introduction

As computers become more powerful, software design tools are becoming increasingly more powerful tools for architects and designers to express their ideas. In addition, the use of artificial intelligence techniques, such as evolutionary algorithms, has enabled these software packages to assist in the design process themselves. Already evolutionary design systems (EDS) have been used for the design of antennas, flywheels, load cells, trusses, robots and other structures [1, 2]. While these systems have been successful at producing simple, albeit novel artifacts, concerns with them are how to enable them to produce truly novel solutions and how to increase the sophistication of what they can produce. Here we argue that achieving sophisticated, novel designs is enabled by the use of open-ended, generative representations.

Breaking down an EDS into its separate modules yields the representation for encoding designs, the evolutionary algorithm for exploring the space of designs that can be represented, and the fitness function for scoring the quality of a particular design. While the evolutionary algorithm and fitness function influence the designs pro-

duced by the design system, results are limited to those that can be expressed by the chosen representation. For example, in optimizing the dimensions of a parameterized design the EDS can only produce designs that fall in the pre-specified parameter space and no modification of the evolutionary algorithm or choice of fitness function can enable the design program to produce designs outside that parameter space. Consequently, to improve the sophistication of what can evolved better representations need to be used.

Representations for computer-automated design can be divided into *parameterizations* and open-ended, *generative* representations. With the first class, parameterizations, the topology of the design is pre-specified and the search algorithm is limited to performing numerical optimization on the set of parameters. In contrast, a generative representation is an assembly procedure, or algorithm, for creating a design and the EDS is able explore a space of design topologies thereby allowing novel types of designs to be discovered.

With the ability to explore the space of design topologies comes the challenge of being able to produce sophisticated designs, such as those produced by the best human engineers. In engineering and software development complex artifacts are achieved by exploiting the principles of modularity, regularity, and hierarchy [5] [8], and these characteristics can also be seen in the artifacts of the natural world. Again, the ability of an EDS to achieve designs with modularity, regularity and hierarchy is limited to such systems in which the representation is capable of encoding designs with these characteristics. Thus an important distinction between classes of generative representations is in their ability to hierarchically form and combine modules as well as reuse them.

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

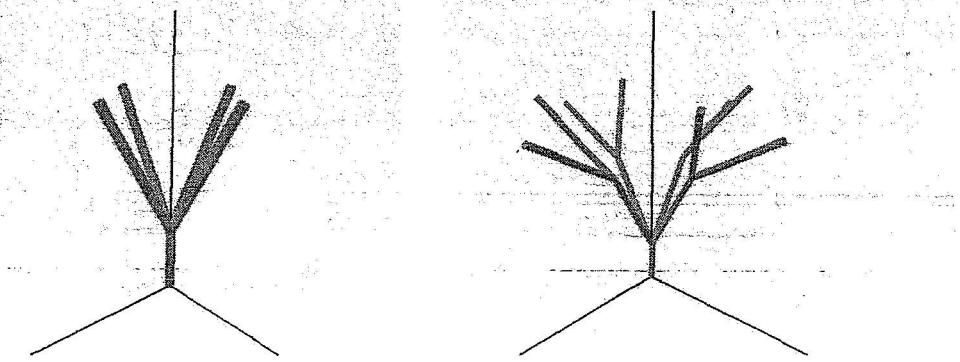
In the rest of this paper we describe work done using two different generative representations. The first example is of a simple generative representation for the evolution of a spacecraft antenna. In the second example we describe a more complex generative representation – with the ability to hierarchically create, combine and reuse modules – and demonstrate its ability to produce sophisticated, modular designs on a table design problem.

2 Evolution of a NASA Spacecraft Antenna

The first problem domain on which we demonstrate the use of generative representations is the design of an antenna for NASA's Space Technology 5 (ST5) mission. The ST5 mission consists of three small spacecraft that will orbit the Earth and measure the magnetosphere. Initially the three ST5 spacecraft were to orbit at close separations in a highly elliptical geosynchronous transfer orbit approximately 35,000 km above Earth and the requirements for the communications antenna were for a gain pattern of $\geq$0 dBic from 40° - 80° from zenith. Because of a change in launch vehicle and the new, lower orbit this necessitated the addition of a new requirement on the gain pattern of $\geq$-5 dBic from 0° 40° from zenith. Consequently, we evolved first an antenna to meet the initial gain pattern requirements and then evolved a second antenna to meet the additional gain pattern requirements. In addition, the antenna must have a voltage standing wave ratio (VSWR) of under 1.2 at the transmit frequency (8470GHz) and under 1.5 at the receive frequency (7209.125GHz) both at an input impedance of 50Ω, and is restricted in shape to a mass of under 165g, and must fit in a cylinder of height and diameter of 15.24cm.

2.1 Generative Representation for Antennas

The generative representation we used for evolving antennas consists of antenna-constructing programs that are composed of commands from a simple programming language we devised for building objects out of line segments [4]. The language is composed of commands that specify wire segments and perform coordinate system rotations. An antenna design is created by starting with an initial feedwire and creating wires specified by executing the evolved antenna-constructing program. The command `forward(length, radius)` adds a wire with the given length and radius. The command `rotate-x(angle)` changes the coordinate system orientation by rotating it the specified amount about the x-axis. Similar commands are defined for the y and z axes.



(a) (b)

Figure 1: Example antennas: (a) non-branching arms; (b) branching arms.

For example, in executing the program `rotate-z(0.5236) forward(1.0,0.000406)`, the `rotate-z()` operator causes the the current orientation to rotate 0.5236 radians (30°) about the Z axis. The `forward()` operator adds a wire of length 1.0 cm and radius 0.000406 cm (which corresponds to a 20 gauge wire) in the current forward direction. Branches in the representation cause a branch in the flow of execution and create different branches in the constructed antenna. The following is an encoding of an antenna with branching in the arms, here brackets are used to separate the subtrees: `rotate-z(0.5236) [ forward(1.0,0.032) [`

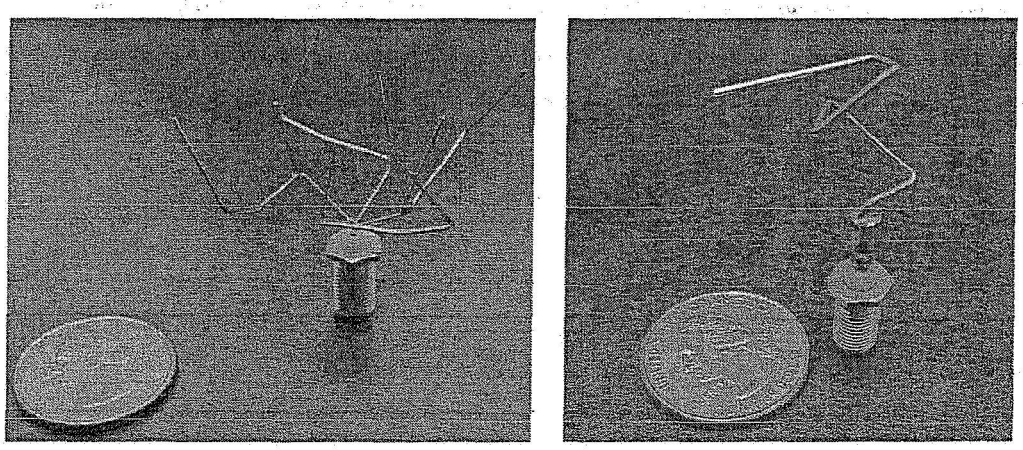
```
rotate-z(0.5236) [ forward(1.0,0.032) ] rotate-x(0.5236) [ forward(1.0,0.032)
] ] ]
```

To produce antennas for the initial ST5 mission requirements we constrained our evolutionary design system to a monopole wire antenna with four identical arms, with each arm rotated 90° from its neighbors. The EA thus evolves a program that specifies the design for one arm, and evaluates these individuals by building a complete antenna using four copies of the evolved arm. Graphical images of the two antennas produced by the sample program in the previous paragraph are shown Figure 1.

For the revised mission specifications the four-arm design is unacceptable because it has a null at zenith so we changed our system to produce a single arm. In addition, because of the difficulties we experienced in fabricating branching antennas to the required precision, here we constrained our antenna designs to non-branching ones.

2.2 Evolved Antennas

To achieve a requirements-compliant antenna for the ST5 mission an evolutionary algorithm was used to automatically breed antennas in a simulated computer environment. This evolutionary algorithm works by starting with a population of randomly generated antenna-constructing programs using that result in poor-performing random antenna designs. Using biased selection of higher-performing antennas, the antenna-constructing programs are bred using recombination and mutation operators inspired by natural evolution. These operators create new antenna designs from old ones by swapping parts of antenna-constructing programs and randomly changing the commands. From all of these runs the best antenna design found was fabricated and tested (Fig. 2).



(a) (b)

Figure 2: Photographs of prototype evolved antennas: (a) the best evolved antenna for the initial gain pattern requirement, ST5-3-10; (b) the best evolved antenna for the revised specifications, ST5-33.142.7.

Compliancy with mission requirements was confirmed by testing the prototype antenna in an anechoic test chamber at NASA Goddard Space Flight Center. In comparison with traditional design techniques, the evolved antenna has a number of advantages in regard to power consumption, fabrication time, complexity, and performance. Originally the ST5 mission managers had hired a contractor to design and produce an antenna for this mission. Using conventional design practices the contractor produced a quadrifilar helix antenna (QHA). In figure 3 we show performance comparisons of our evolved antennas with the conventionally designed QHA on an

ST5 mock-up. Since two antennas are used on each spacecraft – one on the top and one on the bottom – it is important to measure the overall gain pattern with two antennas mounted on the spacecraft. With two QHAs, 38% efficiency was achieved, using a QHA with an evolved antenna resulted in 80% efficiency, and using two evolved antennas resulted in 93% efficiency. Lower power requirements result from achieving high gain across a wider range of elevation angles, thus allowing a broader range of angles over which maximum data throughput can be achieved. Since the evolved antenna does not require a phasing circuit, less design and fabrication work is required, and having fewer parts may result in greater reliability. In terms of overall work, the evolved antenna required approximately three person-months to design and fabricate whereas the conventional antenna required approximately five months. Lastly, the evolved antenna has more uniform coverage in that it has a uniform pattern with only small ripples in the elevations of greatest interest (40° – 80°). This allows for reliable performance as the elevation angle relative to the ground changes. Finally, the evolved antenna represents the first antenna to be fielded with an evolved shape and, if deployed successfully when launched in 2006, the first evolved object to fly in space [6, 7].

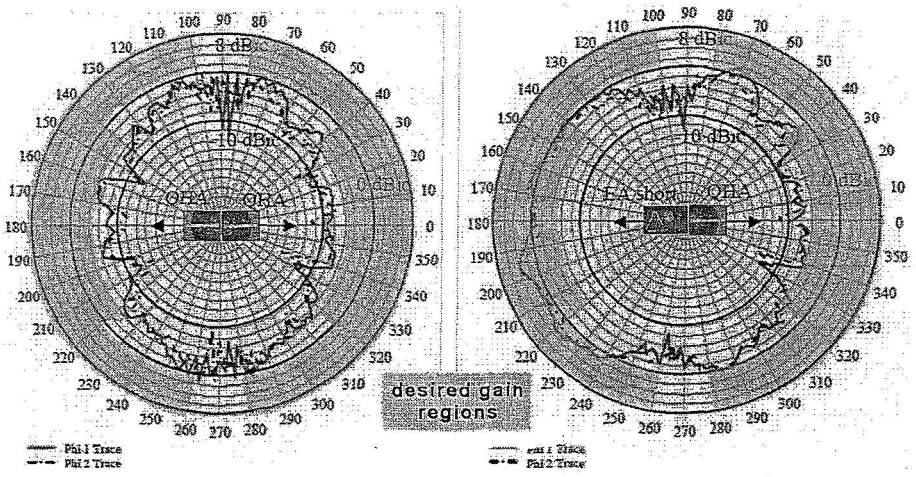


Figure 3: Measured patterns on ST-5 mock-up of two QHAs and a ST5-104.33 with a QHA. Phi 1 = 0 deg., Phi 2 = 90 deg.

3 Increasing Evolved Sophistication

For computer-automated design systems to scale to complex designs they must be able to produce designs that exhibit the characteristics of modularity, regularity and hierarchy – characteristics that are found both in man-made and natural designs. We claim that these characteristics are enabled by implementing representations with the attributes of combination, control-flow and abstraction [3]. In this section we describe a generic, generative representation for producing more sophisticated designs with the characteristics of modularity, regularity and hierarchy.

3.1 Generative Representation

The generative representation we use for enabling modularity, regularity and hierarchy is a kind of computer language within which design-constructing programs are written. Modularity is enabled through subprocedure-like elements (abstraction), regularity is enabled through iterative loops and subprocedures, and hierarchy is enabled by being able to nest subprocedures and iterative loops inside themselves.

The language consists of a framework for design construction rules and a set of these rules defines a program for a design. Designs are created by compiling a design program into an assembly procedure of construction commands and then executing this assembly procedure in the module which constructs designs. The rules for constructing a design consist of a rule head followed by a number of condition-body pairs. For example in the following rule,

$$A(n0, n1) : n1 > 5 \rightarrow B(n1{+}1)cD(n1{+}0.5, n0{-}2)$$

the rule head is A($n0$, $n1$), the condition is $n1 > 5$ and the body is B($n1$+1) c D($n1$+0.5, $n0$-2). A complete encoding of a design consists of a starting command and a sequence of rules. For example a design could be encoded as,

$$P0(4)$$

$$P0(n0): \quad n0 > 1.0 \rightarrow [\ P1(n0 * 1.5)\ ]\ a(1)\ b(3)\ c(1)\ P0(n0 - 1)$$

$$P1(n0): \quad n0 > 1.0 \rightarrow \{\ [\ b(n0)\ ]\ d(1)\ \}(4)$$

Through an iterative sequence of replacing rule heads with the appropriate body this program compiles as follows,

1. *P0(4)*
2. *[P1(6)] a(1) b(3) c(1) P0(3)*
3. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [P1(4.5)] a(1) b(3) c(1) P0(2)*
4. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [P1(3)] a(1) b(3) c(1) P0(1)*
5. *[{ [b(6)] d(1) }(4)] a(1) b(3) c(1) [{ [b(4.5)] d(1) }(4)] a(1) b(3) c(1) [{ [b(3)] d(1) }(4)] a(1) b(3) c(1)*
6. *[[b(6)] d(1) [b(6)] d(1) [b(6)] d(1) [b(6)] d(1)] a(1) b(3) c(1) [[b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1) [b(4.5)] d(1)] a(1) b(3) c(1) [[b(3)] d(1) [b(3)] d(1) [b(3)] d(1) [b(3)] d(1)] a(1) b(3) c(1) b(3)*

To create designs with this type of generative representation, the non-rule-head symbols are interpreted as construction commands in a design construction language. For example, three-dimensional objects can be constructed by creating a language for adding cubes in a three-dimensional grid: `back`(n), move in the negative X direction n units; `clockwise`(n), rotate heading $n \times 90°$ about the X axis; `counter-clockwise`(n), rotate heading $n \times -90°$; about the X axis; `down`(n), rotate heading $n \times -90°$ about the Z axis; `forward`(n), move in the positive X direction n units; `left`(n), rotate heading $n \times 90°$ about the Y axis; `right`(n), rotate heading $n \times -90°$ about the Y axis; `up`(n), rotate heading $n \times 90°$ about the Z axis; `]`, pop the top state off the stack and makes it the current state; and `[`, push the current state to the stack.

With this design-construction language a design starts with a single cube in a three-dimensional grid and new

cubes are added with the commands `forward()` and `back()`. The current state, consisting of location and orientation, is maintained and the commands `clockwise()`, `counter-clockwise()`, `down()`, `left()`, `right()`, and `up()` change the orientation. A branching in design construction is achieved through the use of the commands `[` and `]`, which push (save) and pop (restore) the current state onto a stack. Using the key: a = up, b = forward, c = down, and d = left; the above example becomes,

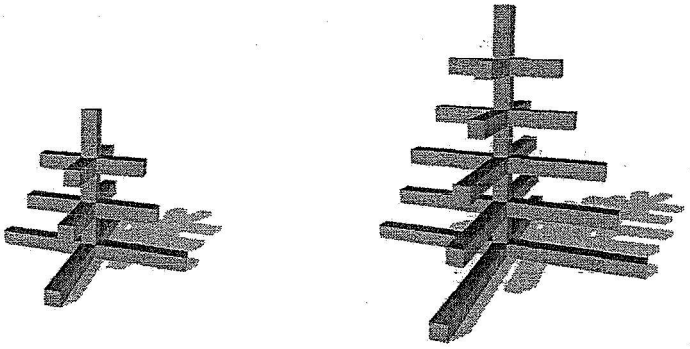
$$
\begin{aligned}
&P0(4)\\
&P0(n0): \quad n0 > 1.0 \rightarrow [\, P1(n0 * 1.5)\,]\ up(1)\ forward(3)\\
&\qquad\qquad down(1)\ P0(n0 - 1)
\end{aligned}
$$

~~$P1(n0): \quad n0 > 1.0 \rightarrow \{[\, forward(n0)\,]\ left(1)\,\}(4)$~~

and compiles into the sequence:

[[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)

Executing this assembly procedure produces the structure shown in figure 4.a. Interestingly, the rules of this design program encode for a family of designs and by using a different starting command different designs can be created. The design in figure 4.b is created by using the starting command $P0(6)$ instead of $P0(4)$.



(a) (b)

Figure 4: Two tree structures produced from the same set of rules with different starting commands.

3.2 Evolution of Tables

To demonstrate the advantages of a generative representation of with modularity, regularity and hierarchy enabled we compare it against a generative representation without subprocedures and iterative loops. The design problem on which we compare the two representations is that of producing tables. The fitness function to score tables is a function of its height, surface structure, stability and number of excess cubes used. Height, f_{height}, is the number of cubes above the ground. Surface structure, $f_{surface}$, is the number of cubes at the maximum height. Stability,

$f_{stability}$, 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, f_{excess}, is used to reward designs that use fewer bricks. To produce a single fitness score for a design these five criteria are combined together:

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

We compare the advanced generative representation against a simple generative representation without modularity, regularity and hierarchy enabled. For each trial the evolutionary algorithm was configured to run for two thousand generations with a population of two hundred individuals. The graph in figure 5 contains the results of these experiments. Evolution with the advanced generative representation increased in fitness faster than with the simple generative representation and had a higher final average fitness of approximately five hundred thousand versus a final average of just under two hundred thousand with the non-generative representation. In addition, the greater leveling off of the fitness curve with the simple generative representation suggests that it does not handle increased design complexity as well as the advanced generative representation.

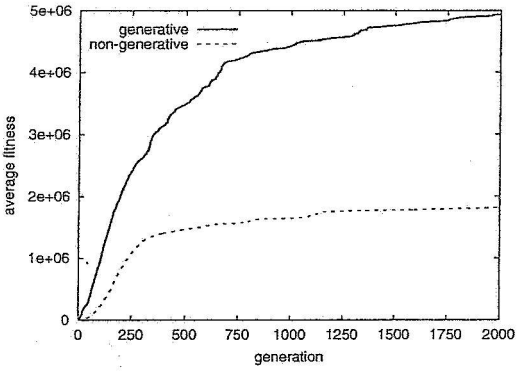


Figure 5: Fitness comparison between the simple generative and advanced generative representations on evolving tables.

Images of tables evolved with the two generative representations show the different styles achieved with them. Examples of the best table evolved with each representation, along with additional tables evolved with the advanced generative representation, are shown in figure 6. The number of parts in these tables range from under a thousand to 5921 for the table in figure 6.d. In general, tables evolved with the simple generative representation are irregular and evolution with this representation tends to produce designs in which tables are supported by only one leg. The likely reason for this is that it is not possible to change the length of multiple table-legs simultaneously with the simple generative representation, so the best designs (those with the highest fitness) had only one leg that raised the surface to the maximum height. In contrast, tables evolved with the advanced generative representation have a reuse of parts and assemblies of parts and are supported with multiple-legs.

3.3 Advantages of the Advanced Generative Representation

That adding elements of programming languages such as subprocedures and iterative loops increases the sophistication of what can be evolved can be intuitively understood by examining a design evolved using the advanced

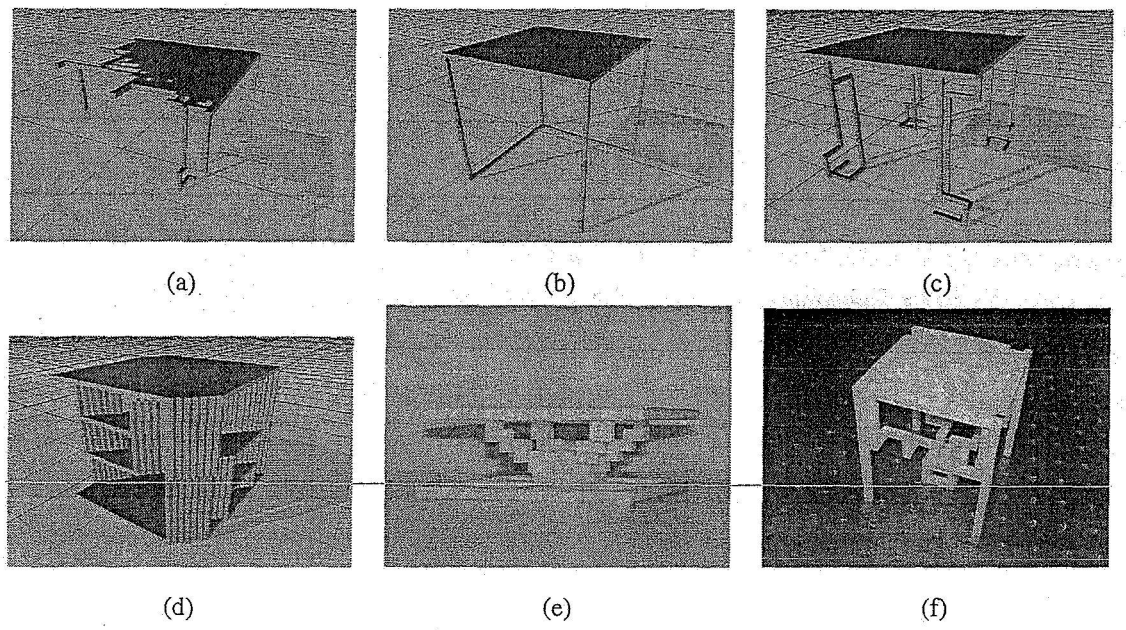


Figure 6: Evolved tables: (a) the best table evolved using the simple generative representation; (b) the best table evolved using the advanced generative representation; (c)-(f) are other tables evolved using the advanced generative representation with variations of the original fitness function.

generative representation. Figure 7 contains examples of different tables that can be produced with a single change to an encoded design. The original table is shown in figure 7.a and one change to its generative encoding can produce a table with: (b), three legs instead of four; (c), a narrower frame; or (d), more cubes on the surface. With a representation without reusable modules these changes 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, such as changing the height of the table legs, 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-voxel changes. Yet, in the general case this would result in a significantly slower search speed in comparison with a single change to a table encoded with the advanced generative representation.

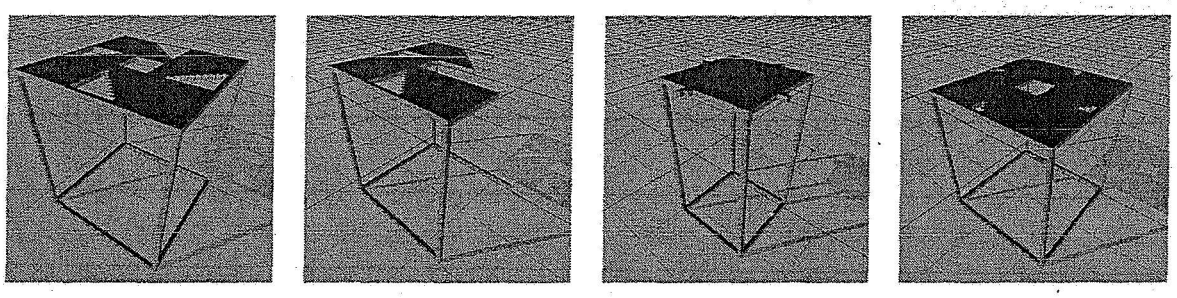


Figure 7: Mutations of a table.

4 Conclusion

The designs that we can achieve are limited only by our imagination and the tools with which we work. Similarly, the designs that evolutionary design systems can achieve are limited only by the representations with which they operate. Here we demonstrated two different generative representations on two different design problems. With the simple generative representation we showed how novel antenna designs could be produced, one of which will be flown on NASA's Space Technology 5 mission. To achieve designs of greater sophistication we claimed that the characteristics of modularity, regularity and hierarchy must be enabled in the representation by adding features of programming languages such as abstraction and iteration. We demonstrated the advantages of a generative representation with this ability on a table design problem and showed that a meaningful modularity of the problem had been evolved. As future work produces increasingly more powerful representations for hierarchically encoding organizational units so too will evolutionary design systems improve in their ability to produce ever more complex and interesting designs.

References

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

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

[3] G. S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In H.-G. B. et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference, GECCO-2005*, pages 1729–1736, New York, 2005. ACM Press.

[4] G. S. Hornby, H. Lipson, and J. B. Pollack. Generative representations for the automatic design of modular physical robots. *IEEE Transactions on Robotics and Automation*, 19(4):703–719, 2003.

[5] 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.

[6] J. Lohn, G. Hornby, and D. Linden. Evolutionary antenna design for a NASA spacecraft. In U.-M. O'Reilly, T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 18, pages 301–315. Springer, Ann Arbor, 2004.

[7] J. D. Lohn, G. S. Hornby, and D. S. Linden. Rapid re-evolution of an X-band antenna for NASA's space technology 5 mission. In T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 5, pages 65–78. Springer, Ann Arbor, 2005.

[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.