Improving Evolvability through Generative Representations

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 of computer automated design systems is the representation used for encoding designs. Using computer programs as an analogy, representations can be thought of as having the properties of *combination*, *control-flow* and *abstraction*. *Generative representations* are those which have the ability to reuse elements in an encoding through either iteration, a form of control-flow, or abstraction. Here we argue that generative representations improve the evolvability of designs by capturing design dependencies in a way that makes them easier to change, and we support this with examples from two design substrates.

1 Introduction

Computer automated design systems have been used to design a variety of different types of artifacts such as antennas [1], flywheels, load cells [2], trusses [3], and robots [4]. 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 the larger design spaces associated with more complex artifacts. In engineering and software development, complex artifacts are achieved by exploiting the principles of regularity, modularity, hierarchy and reuse.

While the optimization algorithm can affect the degree of modularity, regularity and hierarchy in a design, their manifestation is limited by the representation used to encode an artifact. For example, with the parameterization of a table shown in figure 1, no modification to the search algorithm can affect the degree of reuse in an evolved design, nor is the hierarchical construction of building blocks possible.

There are many different ways of converting the encoding of a design (the *genotype*) to the actual design (the *phenotype*) but since it is always performed through a computational process a representation can be thought of as a computer program. With this analogy, features of programming languages can be used to understand and classify different representations. From [5], programming languages have features of:

- **Combination**: Languages create the framework for the hierarchical construction of more powerful expressions from simpler ones, down to atomic primitives.

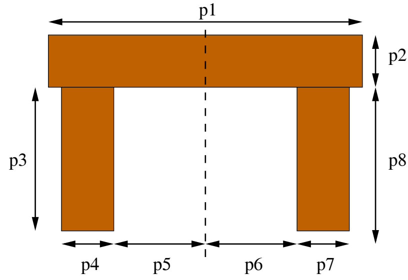


Fig. 1. A parameterization of a table.

- **Control-flow**: All programming languages have some form of control of execution, which permits the conditional and repetitive use of structures.
- **Abstraction**: Both the ability to label compound elements (to manipulate them as units) and the ability to pass parameters to procedures are forms of abstraction.

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

The meanings of combination, control-flow and abstraction translate almost directly from properties of programming languages to properties of design representations. Combination refers to the ability to create more complex expressions from the basic set of commands in the language. It is not fully enabled by mere adjacency, such as proximity in strings utilized by typical GA representations, instead the subroutines of GLib [6] and genetic programming (GP) [7] allow explicit combinations of expressions. 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 (CA). Iteration is a looping ability, such as the repeat structure in cellular encoding [8], or embedded in the fundamental behavior of CA's. Abstraction is the ability to encapsulate part of the genotype and label it such that it can be used like a procedure, such as with automatically defined functions (ADFs) in GP or automatically defined sub-networks (ADSNs) in cellular encoding. Abstraction can be seen when subfunctions can take parameters, as with ADFs.

Using these properties 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 such as dimensions of a pre-defined structure as in the table in figure 1, and have no properties of combination, control-flow or abstraction. Since one of the goals of automated design systems is to achieve truly novel artifacts, we focus on open-ended representations, those in which the topology of a design is changeable, because it is difficult for a parameterization to achieve a type of design that was not conceived of by its creators.

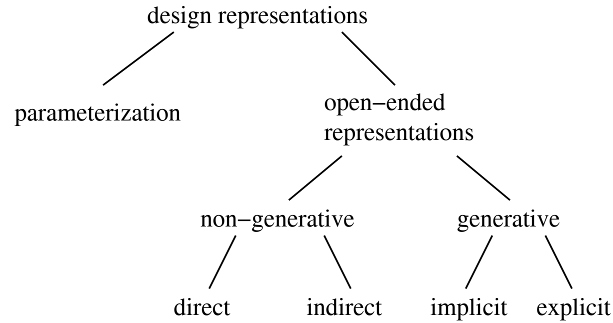


Fig. 2. Classes of design representations.

A fundamental distinction between open-ended representations is whether it is *non-generative* or *generative*. With a non-generative representation each representational element of an encoded design can map at most once to 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 essentially the same as the actual design, and with an indirect representation there is a translation, or construction process, in going from the encoding to the actual design. A **generative representation** is one in which an encoded design can reuse elements of its encoding in the translation to an actual design through either abstraction or iteration (a form of control-flow). The two subcategories of generative representations are *implicit* and *explicit*. Implicit generative representations consist of a set of rules that implicitly specify an artifact, such as through an iterative construction process similar to a cellular automata (CA), and explicit generative representations are a procedural approach in which a design is explicitly represented by an algorithm for constructing it. This hierarchy of design representations is shown in figure 2.

Both direct and indirect non-generative representations are limited in their ability to scale to complex structures because with the increase in the size of the genotype there is an exponential growth in the size of the design space and because the increasing number of dependencies in a design makes it more difficult to make changes. In the first case, 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 a proportionately smaller and smaller move toward 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. The second case is similar: 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

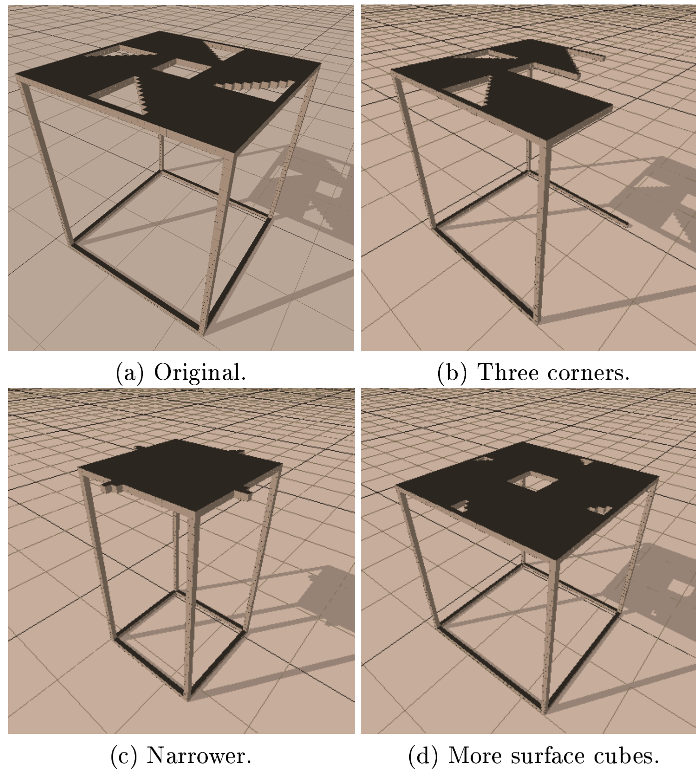
of a table leg is changed, then all of the other table legs must be changed or the table will become unbalanced. Non-generative representations are not well suited to handling these increases in size and complexity because their language for representing designs is static.

Unlike a non-generative representation, a generative representation's ability to reuse elements of an encoded design improves the ability of search to navigate large design spaces and improves scalability by capturing design dependencies through the discovery of useful building blocks. First, 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 the design encoding with a non-generative representation. With a generative representation, abstraction allows for these assemblies to be inserted/deleted through the change of a single symbol, and iteration allows for the addition/deletion of multiple copies of groups of parts through changing the iteration counter. Secondly, 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. With a generative representation the ability to 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.

2 Examples of Evolution with a Generative Representation

That evolution with a generative representation improves the evolvability of designs can be intuitively understood by looking at some examples. Figure 3 contains different tables that can be produced with a single change to an encoded design. The original table is shown in figure 3.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 non-generative representation 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 a generative representation.

That the evolutionary design system is taking advantage of the ability to make coordinated changes with a generative representation is demonstrated by individuals taken from different generations of the evolutionary process. The se-



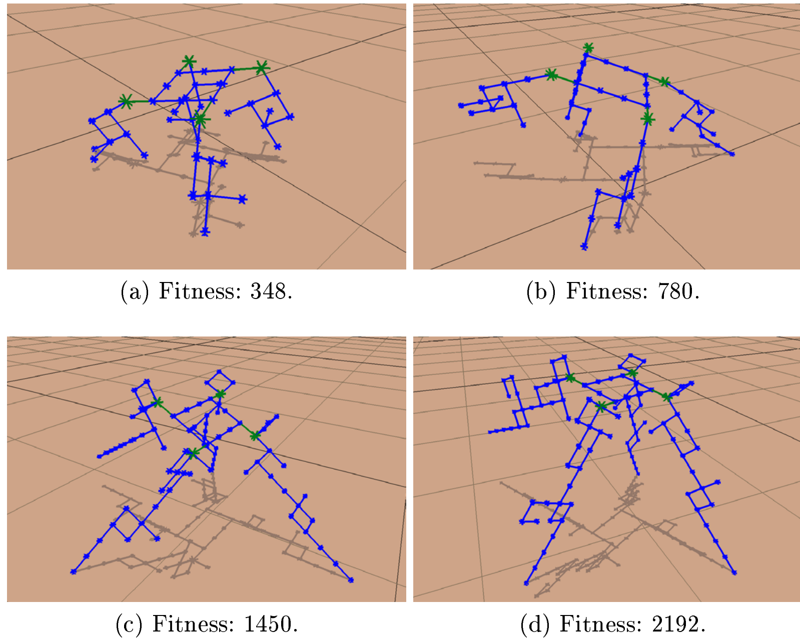
(a) Original. (b) Three corners.

(c) Narrower. (d) More surface cubes.

Fig. 3. Mutations of a table.

quence of images in figure 4, which are of the best individual in the population taken from different generations, show two changes occurring. First, the rectangle that forms the body of the genobot goes from two-by-two (figure 4.a), to three-by-three (figure 4.b), before settling on two-by-three (figures 4.c-d). These changes are possible with a single change on a generative representation but cannot be done with a single change on a non-generative representation. The second change is the evolution of the genobot's legs. That all four legs are the same in all four images strongly suggests that the same module in the encoding is being used to create them. As with the body, changing all four legs simultaneously can be done easily with the generative representation by changing the one module that constructs them, but would require simultaneously making the same change to all four occurrences of the leg assembly procedure in the non-generative representation.

One other advantage of using a generative representation is that by encoding an object through a set of reusable rules for its construction it is possible to encode a class of designs. By evaluating an individual with different parameters



(a) Fitness: 348.

(b) Fitness: 780.

(c) Fitness: 1450.

(d) Fitness: 2192.

Fig. 4. Evolution of a four-legged walking genobot.

to its starting command, families of designs can be evolved, such as the tables in figure 5 [9].

3 Conclusion

Here we defined three properties of design representations and have argued that for computer-automated design systems to scale in complexity they must use generative representations: representations which allow for the hierarchical construction of reusable organizational units. To support this claim we presented examples from our work in evolving tables and robots and showed that designs encoded with a generative representation did capture some design dependencies and enabled moving through the design landscape in more meaningful ways than would have been possible with a non-generative representation.

While this comparison has shown how important reuse is for an evolutionary design system to scale to complex designs it has not discussed how to achieve it. The representation used here is a modification of Lindenmayer Systems and is described in previous work [10, 11]. But there are many ways of allowing reuse in the representation – such as variations of cellular automata, models of developmental biology, as well as actual computer programs – each with its own strengths and weaknesses. For now it is premature to say which direction is best, but as representations become increasingly more powerful in hierarchically

Fig. 5. Two tables from a family of designs.

encoding organizational units so too will computer-automated design systems improve in their ability to produce ever more complex and interesting designs.

References

1. Linden, D.S.: Innovative antenna design using genetic algorithms. In Bentley, P.J., Corne, D.W., eds.: Creative Evolutionary Systems. Morgan Kaufmann, San Francisco (2001) 487–510
2. Robinson, G., El-Beltagy, M., Keane, A.: Optimization in mechanical design. In Bentley, P.J., ed.: Evolutionary Design by Computers. Morgan Kaufmann, San Francisco (1999) 147–165
3. Michalewicz, Z., Dasgupta, D., Riche, R.G.L., Schoenauer, M.: Evolutionary algorithms for constrained engineering problems. Computers and Industrial Engineering Journal **30** (1996) 851–870
4. Lipson, H., Pollack, J.B.: Automatic design and manufacture of robotic lifeforms. Nature **406** (2000) 974–978
5. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. Second edn. McGraw-Hill (1996)
6. Angeline, P., Pollack, J.B.: Coevolving high-level representations. In Langton, C., ed.: Proceedings of the Third Workshop on Artificial Life, Reading, MA, Addison-Wesley (1994)
7. Koza, J.R.: Genetic Programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, Mass. (1992)
8. Gruau, F.: Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm. PhD thesis, Ecole Normale Supérieure de Lyon (1994)
9. Hornby, G.S.: Generative representations for evolving families of designs. In et al., E.C.P., ed.: Proc. of the Genetic and Evolutionary Computation Conference. LNCS 2724, Berlin, Springer-Verlag (2003) 1678–1689
10. Hornby, G.S., Pollack, J.B.: Creating high-level components with a generative representation for body-brain evolution. Artificial Life **8** (2002) 223–246
11. Hornby, G.S.: Generative Representations for Evolutionary Design Automation. PhD thesis, Michtom School of Computer Science, Brandeis University, Waltham, MA (2003)