Environment and Planning B: Planning and Design 2012, volume 39, pages 713 – 731

doi:10.1068/b38037

String-rewriting grammars for evolutionary architectural design

James McDermott
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA; e-mail: jmmcd@csail.mit.edu

John Mark Swafford
Natural Computing Research and Applications Group, University College Dublin, Belfield, Dublin 4, Ireland; e-mail: johnmarksuave@gmail.com

Martin Hemberg
Department of Ophthalmology, Children's Hospital Boston, Harvard Medical School, Boston, MA 02115, USA; e-mail: martin.hemberg@childrens.harvard.edu

Jonathan Byrne, Erik Hemberg
Natural Computing Research and Applications Group, University College Dublin, Belfield, Dublin 4, Ireland; e-mail: jonathanbyrn@gmail.com, erik.hemberg@ucd.ie

Michael Fenton, Ciaran McNally
School of Civil, Structural and Environmental Engineering, University College Dublin, Belfield, Dublin 4, Ireland; e-mail: michaelfenton1@gmail.com, ciaran.mcnally@ucd.ie

Elizabeth Shotton
School of Architecture, University College Dublin, Belfield, Dublin 4, Ireland; e-mail: elizabeth.shotton@ucd.ie

Michael O'Neill
Natural Computing Research and Applications Group, University College Dublin, Belfield, Dublin 4, Ireland; e-mail: m.oneill@ucd.ie
Received 11 February 2011; in revised form 14 September 2011

Abstract. Evolutionary methods afford a productive and creative alternative design workflow. Crucial to success is the choice of formal representation of the problem. String-rewriting context-free grammars (CFGs) are one common option in evolutionary computation, but their suitability for design is not obvious. Here, a CFG-based evolutionary algorithm for design is presented. The process of meta-design is described, in which the CFG is created and then refined to produce an improved design language. CFGs are contrasted with another grammatical formalism better known in architectural design: Stiny's shape grammars. The advantages and disadvantages of the two types of grammars for design tasks are discussed.

Keywords: evolutionary design, grammatical evolution, design languages, context-free grammar, shape grammar

1 Introduction

The ideas of evolutionary computation (EC) have proved to be useful in many design domains and many aspects of the design process. Examples include the optimization of buildings' structural properties (Parmee, 2001); the interactive exploration of design spaces (Hornby and Pollack, 2001); and the creation of novel, unexpected solutions to difficult engineering problems (Manos et al, 2007). At its best, evolutionary design is an iterative, creative process which affords the designer a productive alternative workflow. The evolutionary workflow can

have the advantage of leading the designer to ideas and solutions which are creative, novel, or surprising (Gero, 1996). The characteristic concepts of evolutionary computation—a population of designs, selection of the best, recombination and mutation of existing designs, and iterative improvement—seem well suited to the creative process (Bentley, 1999).

A crucial aspect of any evolutionary design system is the *representation*: that is, the way in which the problem is formalized, typically for computer use. It determines the space in which evolution will search. The choice of representation can be seen as a type of metadesign: the creator of the design system implicitly chooses to include in the search space only certain designs, and to exclude others.

One concept commonly used in design representations is the *grammar*. The *shape grammar* (SG) formalism (Stiny and Gips, 1972) is popular in the design domain and has several advantages for design applications. On the other hand, the *context-free grammar* (CFG), much more common in other domains, is the representation chosen for the projects described here. This choice is made partly because CFGs are relatively easy to implement and to combine with EC methods. There are also advantages and disadvantages arising from differences in the languages which can be defined using CFGs and SGs. CFGs are less obviously suitable than SGs for the generation of graphical, 3D, or architectural designs. SGs are not a focus of our work, but because of their popularity in the design domain and their intuitive nature they are a useful reference point for comparison with CFGs.

In this paper, then, our aims are as follows:

- To show how CFGs can be used in evolutionary design. In section 3 the use of a CFG to define a design language is introduced. An algorithm and an implementation which combine EC ideas with CFGs are presented. The user's workflow during a typical evolutionary run is also described.
- To examine the interaction between the processes of evolutionary design and metadesign mentioned above. In section 4 the process of defining and refining a CFG-based design language with influence from evolutionary runs is described.
- To set out some of the key similarities and differences between SGs and CFGs. In section 5 they are compared using minimal examples and drawing on existing literature, but not presenting new experiments. Their advantages and disadvantages for different types of tasks are discussed.

We begin with background material on EC, grammars, and design.

2 Background

Computational approaches to design are now very common. Our focus is on systems which use algorithmic methods and formalisms as an essential part of the design process.

2.1 Grammars in design

One strand of research focusses on open-ended formalisms such as grammars. A grammar consists of a start symbol, sets of terminal and nonterminal symbols, and a set of rules each transforming a left-hand side (LHS) to one of several possible productions in the rule's right-hand side (RHS).

Among the best-known formalisms in the design domain is the SG defined by Stiny and Gips (1972). SGs are, firstly, grammars, consisting of alphabets and production rules. Here the *primitives are geometric shapes*, rather than symbols drawn from an alphabet. *Production rules add, delete, or transform shapes*.

SGs have been applied in art, design, engineering, and architecture. They have been used to generate new designs which inherit aspects of preexisting styles (Pugliese and Cagan, 2002); quickly generate large numbers of varied building designs (Wonka et al, 2003); and generate

landscapes consisting of objects such as houses, trees, and fences and conforming to spatial syntax rules (Mayall and Hall, 2007).

String-rewriting grammars can also be used in design and are the representation used in this paper. They include various types of *L-systems* (Prusinkiewicz et al, 1990), intended to model plant growth but later also used for design, as well as CFGs and others. L-systems differ from CFGs in the way rules are applied (maximally parallel for L-systems, as opposed to sequentially for CFGs), and so can encode a different class of languages.

2.2 Evolutionary computation

EC is a class of algorithms inspired by the process of biological evolution by natural selection. They have been used by many researchers to explore and optimize in many design domains (Hornby and Pollack, 2001; Jacob, 1996; Janssen et al, 2004; Roberts et al, 2010; Rosenman, 1997; Watabe and Okino, 1993; Yang and Soh, 2002). EC has been combined with SGs in a beam cross-section design task (Gero et al, 1994). A wide survey of the EC design literature is given by Kicinger et al (2005).

In EC, solutions to problems are evolved over a number of generations using populations of candidate solutions (often called individuals). Different methods of creating the initial generation (initialization), choosing individuals (selection), exchanging information between individuals (crossover), varying individuals (mutation), evaluating individuals' fitness (evaluation), and choosing individuals for the next generation (replacement) are employed by these algorithms. Although EC uses randomness in some or all of these steps, EC search is *informed*, rather than completely random. The mutation and crossover operators are heuristics which, given some known good individuals, produce new individuals considered *likely* to be good. A typical EC process is shown in figure 1.

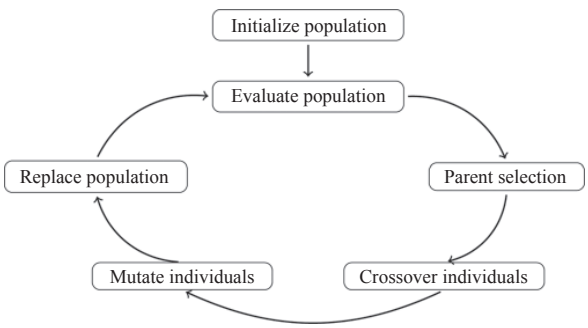

Figure 1. Schematic for a typical evolutionary algorithm.

2.3 Interactive evolutionary computation

Key to the success of EC is the *fitness function*. Typical EC algorithms use an explicit computational fitness function to evaluate the quality of individuals. In some domains it is common to use an *interactive* fitness function or selection method instead. Here a user is required to evaluate or select designs on subjective grounds. This *interactive EC* allows EC to be applied to aesthetic problems.

Possibly the earliest example of interactive EC was Dawkins's *Biomorphs* (Dawkins, 1986), in which images of 2D branching structures were bred according to user selections. The seminal paper by Sims (1991) showed that interactive EC was capable of creating complex and beautiful graphics. It has also been applied in music (Shao et al, 2010), engineering (Parmee, 2001), and form design (Hemberg et al, 2007). Surveys of the issues in this strand of research are given by McCormack (2005) and Takagi (2001). An interactive EC design system based on CFGs is presented next.

3 Grammatical evolution for a design project

The ideas of EC are easily combined with those of CFGs (McKay et al, 2010). One of the best-known approaches is grammatical evolution (GE) which has been applied successfully in many domains including finance, design, and the arts (Dempsey et al, 2009; O'Neill and Ryan, 2003; Shao et al, 2010). GE shares with other EC methods the ideas of a population of individuals, the genotype–phenotype mapping, the fitness function, selection, recombination, mutation, and replacement into the population. The distinguishing feature of GE is the genotype–phenotype mapping, which corresponds to derivation with a CFG.

One example in the design domain is the surface generation tool Genr8 (Hemberg et al, 2007), which uses interactive GE. It works with a fixed CFG which can generate many different *map L-systems*. Each individual corresponds to a deterministic map L-system, which when run gives rise to a 3D-surface design. Thus, the aim of evolution is to find map L-systems which lead to good designs. Jacob (1996) used a similar approach, using EC to search the space of deterministic L-systems corresponding to designs.

GE has also been used to design structural forms constructed from homogeneous beams of variable length (Byrne et al, 2010, 2011; McDermott et al, 2010; O'Neill et al, 2010). The use of GE in this domain is described next.

3.1 Grammatical representation of the project domain

A simplified version of the CFG used in this section is shown in grammar 1, written in the standard Backus–Naur form. It produces simple trusses warped along curves and surfaces.

Grammar 1. The grammar used in the evolutionary runs described in subsection 3.5 (simplified). Indentation indicates the scope of loops. Nonterminals are enclosed in angle brackets.

```
<design>::=<init><body>
<init> ::= nlayers=<nlayers>
           npts=<npts>
           prev_pts=[]
<body> ::= for layer in nlayers:
             cur_pts = []
             for pt in npts + 1:
                <set_pt>
             prev_pts = cur_pts
<set_pt> ::= t=pt/npts
             cur_pt=<path>
             add_pt(cur_pt)
             cur_pts.append(cur_pt)
             connect(cur_pt, cur_pts[pt-1]) # Connect this layer.
             connect(cur_pt, prev_pts[pt])  # Make truss connections
             connect(cur_pt, prev_pts[pt-1]) # to previous layer.
<path> ::= <ellipsoid> | <hypotrochoid> | [...]
<ellipsoid> ::= ((<float> <bop> <var>) * cos(2 * pi * t * <int>),
                 (<float> <bop> <var>) * sin(2 * pi * t * <int>),
                 <expr> * sin(2 * pi * t * f) + <expr>)
<expr> ::= <uop>(<expr>) | (<expr> <bop> <expr>) | <float> | <var>
<uop> ::= sin | cos
<bop> ::= * | +
<var> ::= t | layer
<f> ::= (npts * <int>) | (npts / <int>)
<float> ::= float[0.0...5.0]
<int> ::= 0 | 1 | [...] | 10
<nlayers> ::= 3 | 4 | [...] | 8
<npts> ::= 4 | 5 | [...] | 30
```

In particular, the variation of designs in the x–y plane is constrained to curves such as ellipsoids. Note that the variation available in this grammar is not reducible to a set of purely numerical parameters. The output of this grammar (and of all CFGs used in this paper) is a program in the Python language (http://python.org), which when executed produces a 3D design. Some example programs and designs are given in the following sections.

3.2 Derivation in GE

The GE genotype–phenotype mapping process corresponds to CFG derivation. At each step the left-most nonterminal in the current derivation string is selected to be rewritten. If multiple RHS productions exist for that nonterminal, then a choice is required and so the next unused integer (a 'gene') is read from the genotype. It determines the choice of production according to the 'mod rule': the nth production is chosen, where n equals the gene *modulo* the number of RHS productions in the rule.

In this example, the genotype is an array beginning as follows:

32	28	62	90	71	56	…

Derivation begins with the start symbol in the grammar. Using grammar 1, the start symbol is <design>. Since the rule for this symbol has only a single RHS production, no choice is required: <design> is rewritten to <init><body>. Next, <init> must be rewritten, and again no choice is required: it is rewritten to a string of three lines, beginning with nlayers=<nlayers>. Next, the nonterminal <nlayers> must be rewritten. There are six possible productions in the rule for <nlayers>, so it is necessary to read an integer. The first integer in the genome has the value 32, and 32%6 = 2, so the production with index 2 is chosen. <nlayers> is therefore rewritten to 5.

This mapping process continues until all the nonterminals in the phenotype string have been replaced. If the genotype is exhausted before this happens, the individual is considered *invalid* and awarded a poor fitness value. In this example, however, the mapping process completes and the result is program 1.

Program 1. A program, derived from grammar 1, which generates the individual shown in figure 3(g). Some details and syntax are omitted.

```
nlayers=5
npts=16
for layer in nlayers:
  cur_pts=[]
  for pt in npts+1:
    t=pt/npts
    cur_pt=((1.79 * layer) * cos(2 * pi * t * 1),
          (0.30 + layer) * sin(2 * pi * t * 1),
          (1.33 * sin(2 * pi * layer * 0.48)
          + cos(layer) * sin(2 * pi * t)))
    add_node(cur_pt)
    cur_pts.append(cur_pt)
    add_edge(cur_pt,cur_pts[pt-1])
    add_edge(cur_pt,prev_pts[pt])
    add_edge(cur_pt,prev_pts[pt-1])
  prev_pts=cur_pts
```

3.3 Software

Two main pieces of software were used. Blender (Stichting Blender Foundation, 2009) is an open-source 3D modelling tool. GEVA (O'Neill et al, 2008, see http://ncra.ucd.ie/geva) is an open-source system for general-purpose GE developed by the Natural Computing

Research and Applications group in University College Dublin. A custom interface allows an interactive GEVA session to run inside Blender.

The graphical user interface (GUI) is shown in figure 2. The 3D view on the right allows the current design to be viewed, zoomed, and rotated. The evolutionary controls are shown on the left. They allow the user to view the current population, select good individuals and deselect bad ones, selectively mutate individuals, and eventually tell the algorithm to iterate. The algorithm then works behind the scenes, recombining and mutating the genotypes of selected individuals to create a new population, rendering this new population to the screen, and then returning control to the user.

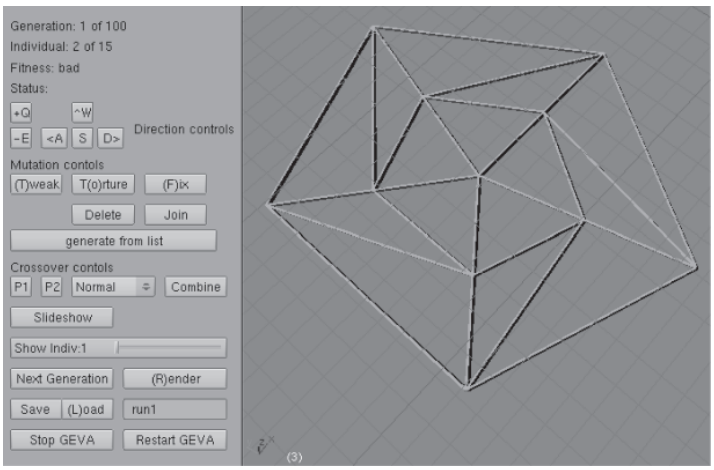

Figure 2. The GEVA/Blender graphical user interface. Evolutionary computation controls are on the left. The 3D view on the right can be zoomed and rotated.

The GEVA software is configured by a number of parameters. The best parameter values for interactive EC differ from typical values for noninteractive EC, as described next.

3.4 Human–computer interaction

A key issue in interactive EC is the *fitness evaluation bottleneck* (Takagi, 2001). Human evaluation of fitness is much slower than computer evaluation. Humans also become bored and fatigued over long runs (Takagi, 2001). Interactive EC is therefore restricted to small populations and few generations. A central goal in the design of the system's GUI and the setting of its parameters is, therefore, to alleviate these problems.

A small population size (fifteen) is chosen to give the user a sense of progress. The number of generations per run is not predetermined, so the user feels a sense of control. Keyboard shortcuts are provided to help the user perform repetitive tasks quickly. A relatively high mutation rate (0.1 per gene) ensures a high degree of novelty, preventing boredom. Selection is binary, that is, each individual is simply 'good' or 'bad', to avoid the time-consuming and fatiguing tasks of setting numerical fitness or ranking individuals (Takagi, 2001). Individuals are marked bad by default, saving some further time. The user is free in each generation to mark as good any number of individuals, an example of user-centric design. There is no *elite*, that is, no individuals passed unchanged from generation to generation, since it is assumed that the user will not wish to evaluate any individual more than once. Custom interactive mutation operators are provided which allow the user to alter a single design, in order to try out variations without going through an entire new generation. Two types of interactive mutation are available: 'nodal' makes a very small, local change in a design, whereas 'structural' makes more fundamental changes (Byrne et al, 2010). These are termed

'tweak' and 'torture', respectively, in the GUI. A custom crossover operator, allowing the user to choose pairs of parents directly, is also available.

3.5 The evolutionary process

Some designs from a typical evolutionary run using grammar 1 are depicted in figure 3. The process begins with great diversity in the population [figures 3(a)–(c)]. The user selects some preferred designs, in this case figure 3(d), on purely aesthetic grounds. Typically just one or two individuals are selected per generation. They become the parents of new individuals by recombination and variation of their genetic material. This leads to gradual refinement [figures 3(e) and (f)]. A good search tactic when several good designs have been created and very slight variations are required is to use the nodal mutation mentioned above [figure 3(g) and (h)]. The design in figure 3(i) is an anomaly, included to indicate the potential for unexpected outcomes even with relatively simple grammars: it is purely planar.

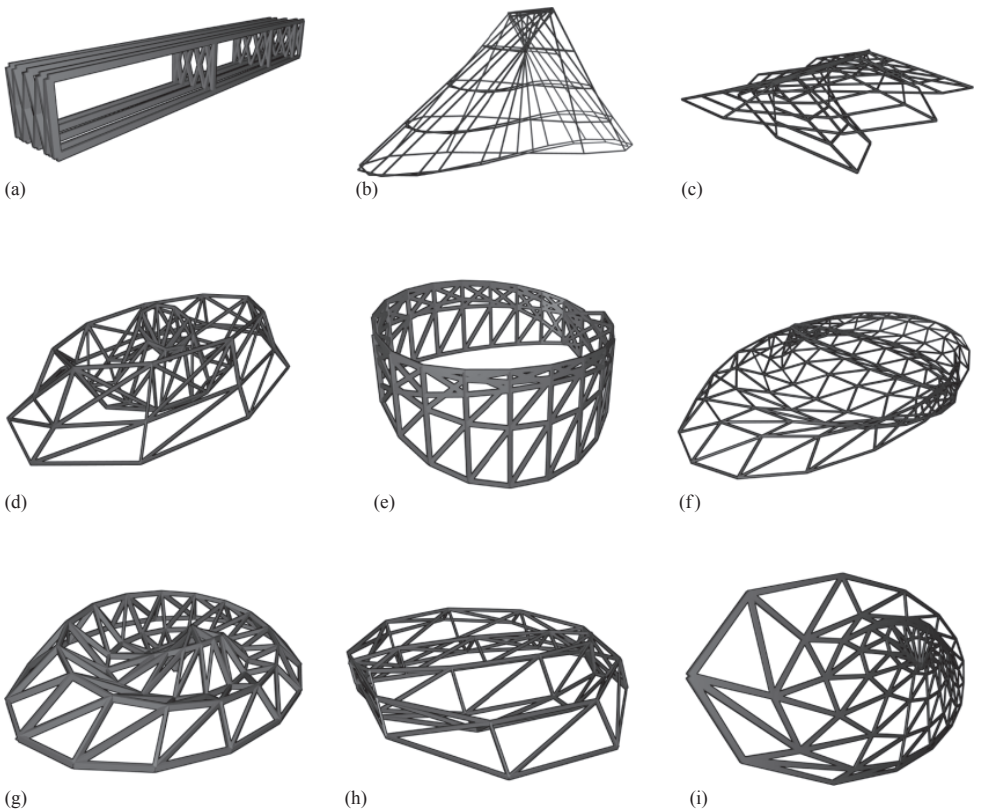

Figure 3. Progress in an evolutionary run, demonstrating early diversity (a)–(c); selection and exploration (d)–(f); focused search (g)–(h); and an anomalous 2D design (i).

One strength of the evolutionary process is that there is no obligation to choose just one design as the final outcome. The option of producing multiple related candidate solutions is highly appreciated by designers (Hemberg et al, 2007). In this case, the designs depicted in figures 3(d) and (f)–(i) form a family of designs related to each other and distinct from those in figures 3(a)–(c). Each is formed of concentric ellipsoid layers, with a truss arrangement of beams joining consecutive layers. The number of subdivisions per layer and the arrangement of the planar ellipsoids in 3D space are the principle variations between members of the family.

Sometimes the designs available with a particular grammar will be seen as overly similar to each other, or the majority of available designs will be seen as very poor. Then a better grammar is required. This *metadesign* problem is discussed next.

4 Design languages

Grammatical approaches to design can consist of two steps: firstly, a grammar is created and refined in a 'metadesign' process and, secondly, the (now fixed) grammar is used to create the designs themselves. There is feedback between the two steps, in that the results achieved in the second inform the refinement of grammars in the first.

4.1 Coverage of desired design space

The basic motivation when creating and refining grammars is to make the design process using those grammars more productive. It is desirable that a language G defined by the grammar should cover the *desired* design language D (otherwise some good designs will be inaccessible). It is also desirable that G should not *exceed* D greatly (otherwise the grammar will generate many unwanted designs). In figure 4 five common cases are distinguished. In the ideal case, figure 4(a), the two languages coincide. In figure 4(b), some desired designs cannot be produced by the grammar. A more extreme version is figure 4(c), where the grammar's language consists of only a few discrete points. In figure 4(d), not only are some desired designs inaccessible, but some undesired ones are not excluded. In figure 4(e), the grammar can produce all desired designs at the expense of producing many undesired ones.

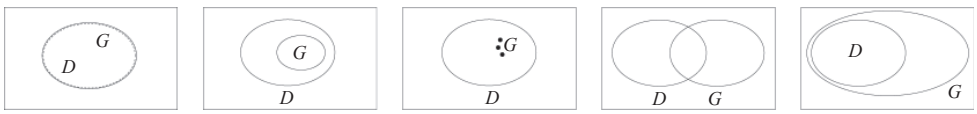

Figure 4. Overlapping and intersection of the desired design language D and the language produced by the grammar G.

The process of changing a grammar to produce a better or different design language has been discussed previously in the context of SGs by Knight (1983). The related processes of *addition to state space* and *substitution in state space* are described by Gero (1996). Here, we focus on two approaches to metadesign using CFGs. One natural approach starts with a single design exemplifying the desired language, as in figure 4(c), and then iteratively *widens* the language by allowing previously constant aspects of this design to vary, ideally approaching figure 4(a). It can also, as in figure 4(e), be useful to *narrow* the language by preventing aspects of variation seen as unhelpful. The processes of widening and narrowing are described next.

4.2 Narrowing the language

Narrowing the design language has the aim of preventing some undesired designs. An example is illustrated in figure 5. Here successive designs are representative of the design languages producible by refinements of the grammar. (Note the contrast with figure 3, where successive designs arise in a single evolutionary run with a fixed grammar.)

The narrowing process begins with the natural grammar 2, capable of producing a very great variety of beam designs. The derived program creates a variable number of beams, each characterized by the x, y, and z and coordinates of its end points. Although the language produced by this grammar includes many types of designs, including all of those shown throughout figure 5, the vast majority are disordered, as in figure 5(a). The situation is as depicted in figure 4(e). Search in this space is therefore unproductive.

Grammar 2. A simple grammar creating a variable number of independently-placed beams.

<design> ::= <beam> | <beam><design>
<beam> ::= beam(<pt>, <pt>)
<pt> ::= (<x>, <x>, <x>)
<xt> ::=-10 |-9 | [...] | 9 | 10

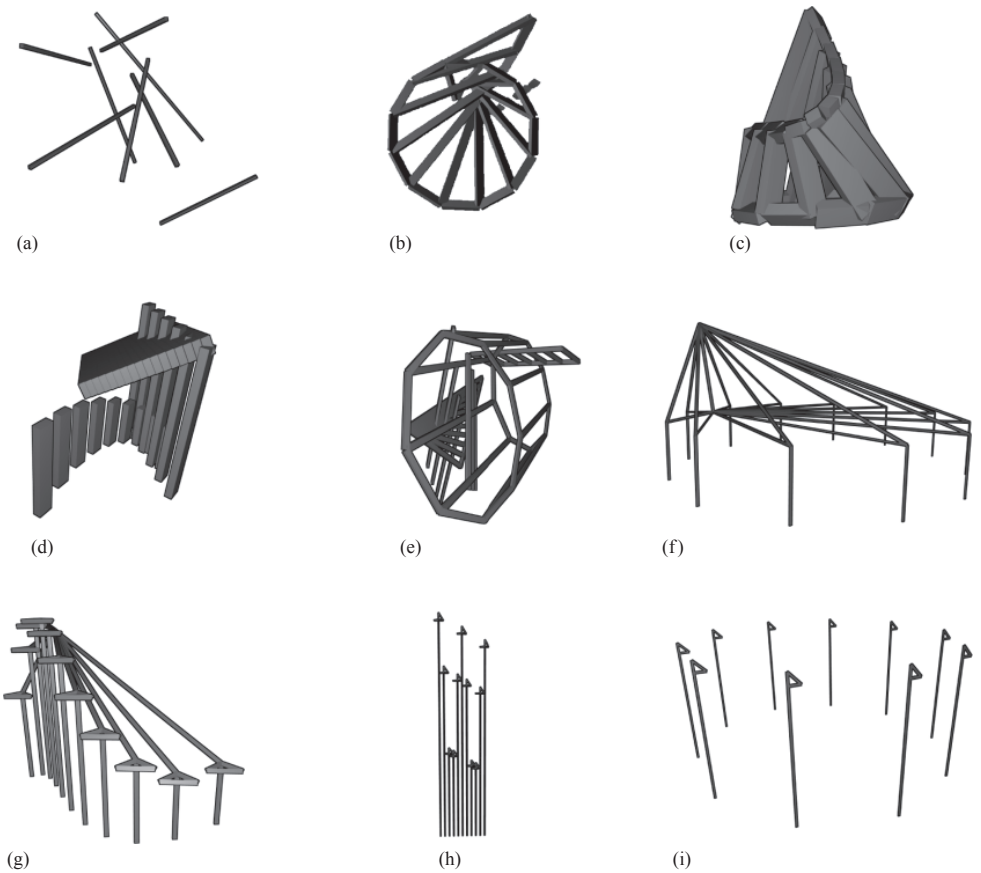

Figure 5. The process of narrowing the design language.

It is natural to narrow the language by placing some constraints on it. One such constraint is that the design be composed of well-formed subcomponents, such as 'ladders' between parallel curves, beams along bezier paths, and beams connecting a given point to several points along a circular path. This constraint is naturally represented using a grammar, as shown in grammar 3. Now, some well-formed designs are achieved, such as those in figures 5(b), (c), and (d). However, these designs arise only through significant evolutionary effort ('weeding out' bad designs during evolution). Again, a more common result is quite disordered, as in figure 5(e). Therefore, the situation remains as depicted in figure 4(e).

Grammar 3. Enforcing the use of subcomponents in the grammar, to avoid independently placed beams. Only a fragment of the grammar is shown.

<design> ::= <component> | <component><design>
<component> ::= <ladder> | <bezier_path> | [...]

A stronger constraint is therefore introduced: grammar 4 creates the entire design as a single conceptual component, using the same types of curves and features as before.

Grammar 4. A more constrained grammar leading to designs each composed of a single, well-formed component. Only a fragment is shown.

```
<design> ::= apply(<behaviour_list>, <path>)
<behaviour> ::= beams_along_path | connect_to_origin | parallel_ladder | [...]
<path> ::= <linear> | <bezier> | <circle> | <sinusoid> | [...]
```

Results including figures 5(f)–(h) and (i) reveal that a much narrower design space has now been achieved—all designs are well formed. The user is not required to evaluate large numbers of obviously bad designs, reducing a problem known to cause boredom and mental fatigue in users of interactive EC systems (Takagi, 2001). In this way the task is far more pleasant, especially in early generations. However the space is still wide enough to allow interesting exploration in evolutionary runs. The situation is now somewhat closer to that depicted in figure 4(a).

4.3 Widening the language

An alternative approach to metadesign is to begin with a program which deterministically generates a single design, and then to progressively *widen* the language produced. This is illustrated in figure 6. These designs were produced as part of a timber bridge design project. Designs are again evaluated purely aesthetically. A separate strand of work, not reported here, also evaluates the structural behaviour of bridge designs using finite-element analysis (Byrne et al, 2011).

Figure 6(a) shows a simple bridge design with a tree motif, created by the deterministic program 2.

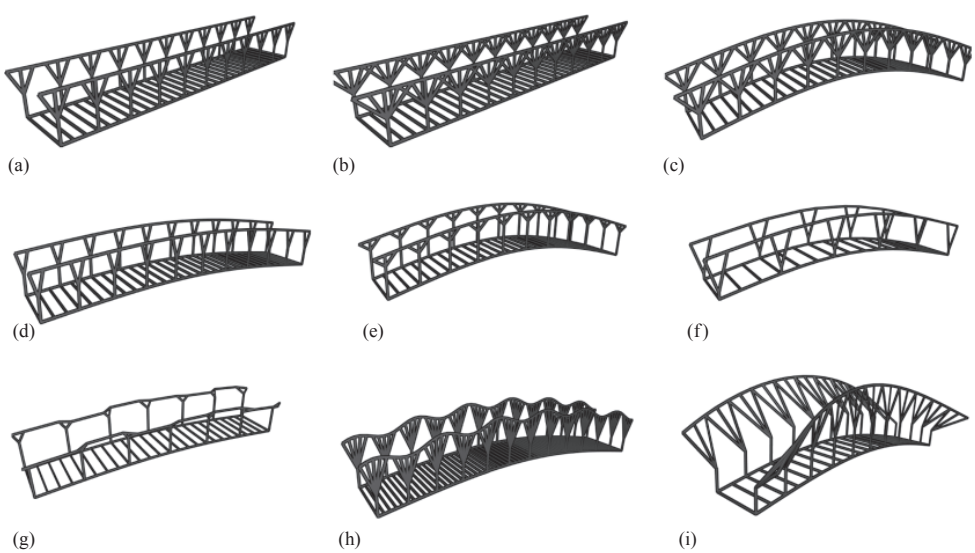

Figure 6. The process of widening the design language.

A grammar is then written which produces this program deterministically. Then the number of branches in each tree in the handrail support is made into a variable, as in figure 6(b). This is achieved by making the numerical constant which sets the number of branches into a new nonterminal. Since the original program included the code

Program 2. Deterministic code which generates the bridge shown in figure 6(a).

```
nstruts=11
nbranches=3
ncrossbeams=3
branch_height=0.6
walkway_height=1.2

def add_strut(pt):
  branch_pt=pt+(0, 0, branch_height)
  add_edge(pt, branch_pt)
  for branch in nbranches:
    xoffset=branch/nbranches- 0.5
    top_pt=pt+(xoffset, 0, walkway_height)
    add_edge(branch_pt, top_pt)
    add_edge(top_pt, prev_top_pt)
    prev_top_pt=top_pt

for strut in nstruts:
  for crossbeam in ncrossbeams:
    xpos=10*strut/nstruts # bridge is 10m long
    cur_pt_l=(xpos, 0, 0)
    cur_pt_r=(xpos, 2, 0) # bridge is 2m wide
    if crossbeam == 0:
      add_strut(cur_pt_l)
      add_strut(cur_pt_r)
    add_edge(cur_pt_l, cur_pt_r)
    add_edge(prev_pt_l, cur_pt_l)
    add_edge(prev_pt_r, cur_pt_r)
    prev_pt_l=cur_pt_l
    prev_pt_r=cur_pt_r
```

nbranches = 3, the new grammar will create the code nbranches = <nbranches> and include a new rule <nbranches> ::= 1 | 2 | … | 5. Similar steps introduce a variable arch to the bridge, as in figures 6(c) and (d), and vary the point at which each tree begins to branch, as in figures 6(e) and (f). Throughout these steps, the situation is as depicted in figure 4(c).

At each step of this process, the grammar designer expands the variation available in the language by allowing some aspect of the design, previously hard-coded in the grammar, to vary. Hofstadter (1985, page 251) says that this metadesign process requires one to "see a variable where there is actually a constant".

In the final stage, a more complex type of widening is performed. Here, the numerical values which give the coordinates of points on the handrail (relative to the walkway) are replaced, not by simple numerical ranges as before, but by functions. This allows the handrail to vary in the y and z planes as a function of t, the proportion of the distance from one end of the bridge to the next. These functions of t are allowed to be quite free form, consisting of straight lines, sinusoids, bezier curves, exponentials, and sums of all of these. The possible forms for these functions are represented grammatically. Some of the results produced by grammar 5 are shown in figures 6(g)–(i).

Grammar 5. Grammar capable of generating any bridge in figure 6 (simplified).

```
<design> ::= <init><defs><body>
<init> ::= nstruts=<nstruts>
        nbranches=<nbranches>
        ncrossbeams=<ncrossbeams>
        branch_height=0.4+<x>*0.5 # branch height varies from 0.4m to 0.9m
        walkway_height=branch_height+<x>*0.5 # walkway is up to 0.5m higher
        arch_height=<x> # arch varies from 0m to 1m
<defs> ::= def y(t):
          return <f>(t)
        def z(t):
          return <f>(t)
        def add_strut(pt):
          t=pt[0]/10 # t indicates relative x-position of pt
          branch_pt=pt+(0, 0, branch_height)
          add_edge(pt, branch_pt)
          for branch in nbranches:
            xoffset=branch/nbranches- 0.5
            yoffset=y(t) # yoffset and zoffset depend on t
            zoffset=z(t) # leading to handrail curves
            new_pt=pt+(xoffset, yoffset, zoffset+walkway_height)
            add_edge(branch_pt, new_pt)
            add_edge(new_pt, prev_new_pt)
            prev_new_pt=new_pt

<body> ::= for strut in nstruts:
          for crossbeam in ncrossbeams:
            [...]

<nstruts> ::= 1 | 2 | [...] | 12
<nbranches> ::= 1 | 2 | [...] | 5
<f> ::= <linear> | <sin> | <bezier> | <exp> | sum(<f>, <f>)
<sin> ::= <x>*sin(2*pi*<x>*t)
<x> ::= [0.0, 1.0]
```

5 Comparison of grammatical representations

In previous sections the use of CFGs in evolutionary design has been introduced. Next, CFGs are contrasted for suitability in design tasks with another type of grammar, SGs as described in section 2.1. SGs are chosen as a reference point because they are well known and successful in the design domain. The aim is to compare the two in general, rather than particular instances. Specific examples are used, however, and differences in their representational power are described directly.

Key similarities and differences between SGs and CFGs are summarized in table 1 and described in detail (highlighted in italic) in the following subsections.

5.1 Implementation and usage

Throughout this paper and in almost all grammar-based EC applications, *grammars are used generatively*. That is, the grammar is intended as a way of creating new objects, and the aim of evolution is to search for good objects. Grammars can also function *analytically*, where the aim is to describe the process by which a preexisting design was created. Similar distinctions are drawn by Krishnamurti (2011) and by Gips (1999). *SGs tend to be used more for analytical than for generative purposes*. A good example is the description of Chinese

Table 1. Similarities and contrasts between context-free grammars and shape grammars for evolutionary design.

	SG	CFG	See (sub)section
Primitives	0D–3D shapes	textual symbols	2.1, 3
Rules	shape insertion/deletion	string rewriting	2.1, 3
Most common uses	analytic	generative	5.1
Computer implementation	difficult	easy	5.1
Link between derivation and design	direct, intuitive	indirect, unintuitive	5.2
Distinction between terminals and nonterminals		clear cut	5.2
Early termination of derivation	possible	impossible	5.2
Emergent/implicit objects as left-hand side	yes	no	5.3
Consistency enforced through grammar	difficult	easy	5.4
Reuse of material	difficult	easy	5.4
Influence between design components	local control	global control	5.4

ice-ray designs by SGs (Stiny, 1977). Given a successful SG analysis there may be little motivation to define a generative CFG.

This distinction in purpose leads to a distinction in implementation. Carrying out even a single CFG derivation by hand is somewhat painful, and the resulting string still requires translation to become a design. *Computer implementation of generative CFGs is necessary and is not difficult*, being based on iterative string matching and rewriting. Using the GE approach described in section 3, or other similar approaches, a generative CFG can be easily integrated into search algorithms such as EC. In contrast, a grammar designed for analytical use may not require a computer implementation. Paper-and-pencil SGs are common, for example the Palladian villa grammar (Stiny and Mitchell, 1978) and the Frank Lloyd Wright prairie house grammar (Koning and Eizenberg, 1981).

The existence of paper-and-pencil SGs is partly due to the fact that *computer implementation of SGs is an extremely difficult task*. The primitive is a graphical object, as opposed to a string. The problem of finding the applicable rules is frustrated by subshapes and emergence, as discussed in subsection 5.3. The authors are unaware of any available computer implementation of fully general SGs. Some authors implement only a single SG, rather than allowing the user to specify it. Others implement highly restricted forms of SG, such as split and set grammars (Wonka et al, 2003). Some do not allow control of rule application (Trescak, 2009). Some work only in 2D (Trescak, 2009), in rectilinear 2D (Tapia, 1999), or only with rectilinear, nonoverlapping solids (Piazzalunga and Fitzhorn, 1998). Few implement fully general subshape recognition and emergence (Gips, 1999).

5.2 Concreteness

Using SGs, *every step of the derivation process produces a concrete design*. The outcome of derivation is simply the final design in a sequence, so derivation can be seen as a journey through a design space. The user can *choose to halt* at any time and take the current design as the output. That is, *the distinction between terminals and nonterminals can be blurred* as desired, since nonterminals are themselves concrete shapes. Existing shapes can also be deleted, if a rule exists whose LHS is the shape in question and whose RHS is the empty shape.

As a consequence of concreteness, the user of a paper-and-pencil SG can avoid or in a sense not even see rules which are obviously wrong in particular circumstances. An SG is then still useful even if the language it produces greatly exceeds the desired language [as in figure 4(e)].

A good example turns up in work by Knight (2003), dealing with Hepplewhite-style chair-back designs. The grammar included a symmetry-imposing 'emergent' rule E -> r(E), where E is any emergent shape and r is a reflection in the *y*-axis. However, the vast majority of the designs generated by this grammar—those in which the reflection rule is not applied last—are still asymmetrical. It is only the user's taste which avoids them. A computer implementation would need to ensure the reflection rule was applied last.

The lack of concreteness in CFGs has its own consequences. The *intermediate stages of CFG derivation do not correspond to concrete designs*. A *derivation string containing nonterminals is incomplete* and does not correspond to any design, so the user *cannot choose to end derivation* at an arbitrary time. The outcome of each choice of rule application is not immediately amenable to visualization, and so the link between a grammar and the designs it produces is rather *indirect and unintuitive*. The two-step process of CFG derivation and program execution prevents deletion or modification of 'existing' objects.

5.3 Shape recognition and emergence

In SGs, shapes can exist in two ways in a design without being explicitly created. Firstly, every shape contains infinitely many subshapes. Secondly, shapes can 'emerge' through the interaction of multiple applications of production rules (Knight, 2003; Stiny, 1994). Consider the SG rule illustrated in figure 7, which duplicates and offsets the square which is the LHS. After two applications of this rule [figure 7(b)], smaller squares emerge despite not being explicitly coded for. These emergent squares are now available to be used as the LHS of later applications.

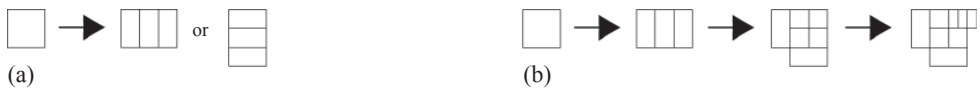

Figure 7. A simple shape grammar (a). After two applications of the rule, smaller squares 'emerge' and are available for use as the left-hand side, as in the final rule application in (b).

Subshapes and emergent shapes are available for use as the LHS of production rules. On paper this presents no problem, since humans are good at recognizing shapes. For computer SG implementations, however, it is a difficult problem. The system must be made to 'see' the shape, in order to determine that it matches the LHS of a rule. It must be recognizable despite being only a part of an explicitly created shape; despite translation, rotation, scaling, and differing surrounding context; and despite being composed of parts created at different times. Subshape recognition and emergence are not always implemented in SG systems, as described in subsection 5.1.

By contrast, LHS recognition is easy for CFGs. The list of nonterminals available as possible LHSs during derivation is always finite. An LHS is a *single, unambiguous, nonterminal symbol*. Emergent shapes can arise when the derived program is translated into a design, but this presents no difficulty during the derivation process. Although the specific design derived in figure 7(b) could be produced by a CFG, a CFG implementing similar displacement–addition rules would not lead to smaller emergent squares as in figure 7(b).

5.4 Reuse for organized complexity and variety

In EC, reuse of information is widely regarded as necessary for the generation of complex, organized designs. In general, a capacity for reuse leads to an increase in the size and complexity of the problems which can be addressed (Hornby, 2005). Two examples of the benefits of reuse follow.

The simple grammar shown in figure 8 generates a bridge of a fixed shape and a variable number of courses. The LHS of the rule is a single course: an underfoot beam and two struts. The rule can add an extra course, with walkway and handrail connections to the

previous course, replacing the previous course with a terminal version (shown with square nodes) to avoid unexpected expansions; or it can simply replace a course with a terminal version, ending the derivation. The rule shown here produces designs which are internally consistent, but no variation between designs is possible other than differing numbers of courses.

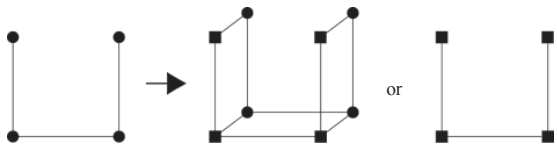

Figure 8. A simple rule for generating courses of a bridge, consisting of a left-hand side and two productions on the right-hand side.

It is possible to create sinusoidally curved handrails, as in section 3, using a parametric version of this grammar such that the height (say) of each new course is calculated from the previous according to an equation containing several numerical parameters. Different curves can be produced by varying these parameters, giving an expanded design language. However, in order to make the courses consistent with each other within a single design, it is necessary that the parameters be fixed throughout its derivation. This requires an extra, *nongrammatical step* prior to the derivation proper. Essentially, a new grammar is required for each design. One of the biggest advantages of the SG approach—the ability to *see* shapes directly in the rules—is also being diluted.

This problem is easily avoided in the CFG approach. Since the strings created by the CFG in representations such as GE are regarded as computer programs, it is natural to allow them to contain various features characteristic of programs. These include variables, loops, functions, and complex data structures. In this example, a variable is used to store information on the number of branches used per strut. It is varied during evolution but fixed per design. Setting the variable's value early in the evolved program makes a consistent value available *throughout construction*. This is a means of *global control*, in contrast to the *local flow of information* in SGs.

For the second example, suppose that the desired design language includes bridge designs with variable numbers of branches in each handrail strut, as in figure 6. It is natural to think of SG production rules for setting the number of branches explicitly, as in figure 9(a). It may be possible instead to write a single, general rule which increases the number of branches from n to $n+1$, as in figure 9(b). In either case, however, it is likely that different struts will end up with different numbers of branches. This can be avoided by fixing the number of branches using a rule such as figure 9(c). However, then the grammar cannot produce, as desired, multiple designs which differ in their numbers of branches. *Consistency with variation is not trivially achieved.*

A CFG can again solve the problem with a procedural idea, this time the *function*. A function is a compound program structure made from lower-level primitives and exposed as a single new primitive. A common implementation in GE is the *automatically defined function* (ADF) (Koza, 1994). Here, each program first defines one or more functions. Each

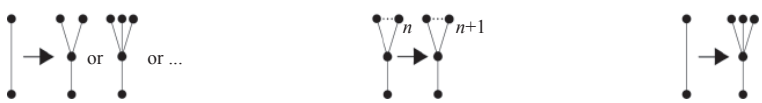

Figure 9. Three possible shape grammar production rules for producing handrail struts with multiple branches: (a) an explicit production for each possible number; (b) a general rule for increasing the number; (c) a fixed number.

function can then be *called* multiple times in the body of the program, performing the *same* work each time. In this case, an ADF named 'strut' might be created to encapsulate all the variations available within each handrail strut. Again, this allows structures to be varied under evolution but used consistently within each individual. *Higher-order functions* (HOFs), that is, functions parametrised by other functions, can implement more sophisticated types of reuse (McDermott et al, 2010).

5.5 Choosing between SGs and CFGs

At the outset of a grammar-based design project, it is natural to ask whether to use a SG or a CFG (or something else). The question may depend more on the project's purpose than on technical distinctions.

In conceptual design, paper-and-pencil settings, the early stages of a project, and analytical or pedagogical projects, it may be most natural to use an SG. They have a concrete, graphical format well-suited to design. They can take advantage of subshapes, emergence, and a blurring of the distinction between terminals and nonterminals. These properties may allow simple grammars to produce more-complex designs and afford the user more-direct control.

In generative projects, in computational settings, and when procedural or programmatic concepts of reuse are of interest, CFGs have advantages. They are easy to implement, and allow the machinery of grammar-based EC to be used for design. Features such as variables, ADFs, and HOFs aid consistency and reuse of material within designs. These are important goals in the projects described in this paper, where CFGs were found to be suitable.

It is likely that some projects would benefit from using both SGs and CFGs as complementary methods with contrasting strengths and weaknesses, perhaps at different stages of the workflow.

5.6 Future work on CFG and SG comparison

The comparisons above constitute an incremental step in understanding the differences between SG-based and CFG-based approaches to design. Future work could continue this investigation. The development of a general SG implementation would be necessary (but difficult: see subsection 5.1). Questions might be asked about the number of objects in the languages defined by different grammars, and the proportion of CFG-producible objects which were also SG-producible, and vice versa. For a fair experiment, it would presumably be necessary to define 'equivalent' SGs and CFGs. To the authors' knowledge, no criteria for such equivalence have been proposed to date. This avenue of research remains in need of new insights.

A second major challenge is to incorporate structural and materials considerations into SG and CFG systems. One strand of our ongoing research deals with the latter (Byrne et al, 2011).

6 Conclusions

In this paper we have described a CFG-based approach to evolutionary design. This approach, using the GE algorithm and GEVA/Blender software, affords the designer a novel and pleasant workflow and gives interesting and unexpected results. A library of CFGs has been built up, each of which produces a different design language.

A key distinction has been drawn between grammars which *can* generate particular good designs, and grammars in which good designs are common. It is easy to create a grammar which is quite universal but fails to restrict the language sufficiently, and so gives rise to vastly more ill-formed designs than good ones. Such grammars are not useful in stochastic settings such as EC. This motivates consideration of the metadesign process in which grammars are created and refined. *Widening* and *narrowing* of language are two possibilities in the context of CFGs. A similar process can be used for SGs (Knight, 1983).

We have described some of the key similarities and differences between SGs and CFGs, leading to a range of conclusions concerning the types of tasks for which SGs and CFGs are best suited.

Acknowledgements. This work was carried out with funding for James McDermott from the Irish Research Council for Science, Engineering, and Technology under the Empower scheme, and for Jonathan Byrne and John Mark Swafford from Science Foundation Ireland under the Grammatical Representations in Evolutionary Design project. Martin Hemberg was supported by grants NIH 1R21NS070250-01A1 and NSF 0954570.

References

Bentley P J, 1999, "Is evolution creative?", in *Proceedings of the AISB'99 Symposium on Creative Evolutionary Systems (CES)* Eds P J Bentley, D W Corne (ASIB) pp 28–34

Byrne J, Fenton M, McDermott J, Hemberg E, O'Neill M, Shotton E, McNally C, 2011, "Combining structural analysis and multi-objective criteria for evolutionary architectural design", http://ncra.ucd.ie/papers/multiDesign.pdf

Byrne J, McDermott J, Galván-Lopéz E, O'Neill M, 2010, "Implementing an intuitive mutation operator for interactive evolutionary 3D design", in *CEC 2010: Proceedings of the 12th Annual Congress on Evolutionary Computation* (IEEE) pp 1–7

Dawkins R, 1986 *The Blind Watchmaker* (Longman, Harlow, Essex)

Dempsey I, O'Neill M, Brabazon A, 2009 *Foundations in Grammatical Evolution for Dynamic Environments* (Springer, Berlin)

Gero J S, 1996, "Creativity, emergence and evolution in design" *Knowledge-Based Systems* **9** 435–448

Gero J S, Louis S J, Kundu S, 1994, "Evolutionary learning of novel grammars for design improvement" *AIEDAM* **8** 83–94

Gips J, 1999, "Computer implementation of shape grammars", in *NSF/MIT Workshop on Shape Computation* http://www.shapegrammar.org/implement.pdf

Hemberg M, O'Reilly U-M, Menges A, Jonas K, da Costa Goncalves M, Fuchs S, 2007, "Genr8: Architect's experience using an emergent design tool", in *The Art of Artificial Evolution* Eds P Machado, J Romero (Springer, Berlin) pp 167–188

Hofstadter D R, 1985 *Metamagical Themas: Questing for the Essence of Mind and Pattern* (Bantam Books, Toronto)

Hornby G S, 2005, "Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design", in *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, volume 2* (ACM, New York) pp 1729–1736

Hornby G S, Pollack J B, 2001, "The advantages of generative grammatical encodings for physical design", in *Proceedings of the Congress on Evolutionary Computation* (IEEE) pp 600–607

Jacob C, 1996, "Evolving evolution programs: Genetic programming and L-systems", in *Proceedings of the first annual conference on genetic programming* (MIT Press, Cambridge, MA) pp 107–115

Janssen P, Frazer J, Ming-xi T, 2004, "Evolutionary design systems and generative processes" *Applied Intelligence* **16** 119–128

Kicinger R, Arciszewski T, Jong K D, 2005, "Evolutionary computation and structural design: a survey of the state-of-the-art" *Computers and Structures* **83** 1943–1978

Knight T W, 1983, "Transformations of languages of designs: part 3" *Environment and Planning B: Planning and Design* **10** 155–177

Knight T, 2003, "Computing with emergence" *Environment and Planning B: Planning and Design* **30** 125–155

Koning H, Eizenberg J, 1981, "The language of the prairie: Frank Lloyd Wright's prairie houses" *Environment and Planning B* **8** 295–323

Koza J R, 1994 *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press, Cambridge, MA)

Krishnamurti R, 2011, Bridging parametric shape and parametric design", in *SDC'10: NSF International Workshop on Studying Visual and Spatial Reasoning for Design Creativity* (Springer, Berlin)

McCormack J, 2005, "Open problems in evolutionary music and art", in *Applications of Evolutionary Computation, Proceedings of EvoMUSART 2005* volume LNCS 3449 (Springer, Berlin) pp 428–436

McDermott J, Byrne J, Swafford J M, O'Neill M, Brabazon A, 2010, "Higher-order functions in aesthetic EC encodings", in *CEC 2010. Proceedings of the 12th IEEE Conference on Evolutionary Computation* http://ncra.ucd.ie/papers/hofWCCI2010.pdf

McKay R I, Hoai N X, Whigham P A, Shan Y, O'Neill M, 2010, "Grammar-based genetic programming: a survey" *Genetic Programming and Evolvable Machines* **11** 365–396

Manos S, Large M C J, Poladian L, 2007, "Evolutionary design of single-mode microstructured polymer optical fibres using an artificial embryogeny representation", in *GECCO: Proceedings of the 2007 Conference on Genetic and Evolutionary Computation* (ACM) pp 2549–2556

Mayall K, Hall G B, 2007, "Landscape grammar 2: implementation" *Environment and Planning B: Planning and Design* **34** 28–49

O'Neill M, Ryan C, 2003 *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language* (Kluwer Academic, Dordrecht)

O'Neill M, Hemberg E, Bartley E, McDermott J, Brabazon A, 2008, "GEVA: grammatical evolution in Java" *SIGEVOlution* **3**(2)17–22

O'Neill M, McDermott J, Swafford J M, Byrne J, Hemberg E, Shotton E, McNally C, Brabazon A, Hemberg M, 2010, "Evolutionary design using grammatical evolution and shape grammars: designing a shelter" *International Journal of Design Engineering* **3** 4–24

Parmee I C, 2001 *Evolutionary and Adaptive Computing in Engineering Design* (Springer, Berlin)

Piazzalunga U, Fitzhorn P, 1998, "Note on a three-dimensional shape grammar interpreter" *Environment And Planning B: Planning and Design* **25** 11–30

Prusinkiewicz P, Lindenmayer A, Hanan J S, Fracchia F D, Fowler D, de Boer M J M, Mercer L, 1990 *The Algorithmic Beauty of Plants* (Springer, Berlin)

Pugliese M, Cagan J, 2002, "Capturing a rebel: modeling the Harley-Davidson brand through a motorcycle shape grammar" *Research in Engineering Design* **13** 139–156

Roberts S, Hall G, Calamai P, 2010, "Evolutionary multi-objective optimization for landscape system design" *Journal of Geographical Systems* **13** 299–326

Rosenman M A, 1997, "The generation of form using an evolutionary approach", in *Evolutionary Algorithms in Engineering Applications* Eds D Dasgupta, Z Michalewicz (Springer, Berlin) pp 69–86

Shao J, McDermott J, O'Neill M, Brabazon A, 2010, "Jive: a generative, interactive, virtual, evolutionary music system" *Lecture Notes in Computer Science* **6025** 341–350

Sims K, 1991, "Artificial evolution for computer graphics", in *SIGGRAPH '91: Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques* (ACM, New York) pp 319–328

Stichting Blender Foundation, 2009 *Blender 3D* http://www.blender.org/

Stiny G, 1977, "Ice-ray: a note on the generation of Chinese lattice designs" *Environment and Planning B* **4** 89–98

Stiny G, 1994, Shape rules: closure, continuity, and emergence. *Environment and Planning B: Planning and Design* **21** 49–78

Stiny G, Gips J, 1972, "Shape grammars and the generative specification of painting and sculpture", in *The Best Computer Papers of 1971* Ed. O R Petrocelli (Auerbach, Philadelphia, PA) pp 125–135. Originally published in Freiman C V (Ed.) *Information Processing 71: Proceedings of the 1971 Congress of the International Federation for Information Processing* (North-Holland, Amsterdam)

Stiny G, Mitchell W J, 1978, "The Palladian grammar" *Environment and Planning B* **5** 5–18

Takagi H, 2001, "Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation" *Proceedings of the IEEE* **89** 1275–1296

Tapia M, 1999, "A visual implementation of a shape grammar system" *Environment and Planning B: Planning and Design* **26** 59–74

Trescak T, 2009, "Shape grammar interpreter", http://sginterpreter.sourceforge.net/

Watabe H, Okino N, 1993, "A study on genetic shape design", in *Proceedings of the 5th International Conference on Genetic Algorithms* (Morgan Kaufmann, San Francisco, CA) pp 445–451

Wonka P, Wimmer M, Sillion F, Ribarsky W, 2003, "Instant architecture" *ACM Transactions on Graphics* **22** 669–677

Yang Y, Soh C K, 2002, "Automated optimum design of structures using genetic programming" *Computers and Structures* **80** 1537 – 1546