Reuse, Parameterized Reuse, and Hierarchical Reuse of Substructures in Evolving Electrical Circuits Using Genetic Programming

John R.Koza[1] Forrest H Bennett III[2]
David Andre[3] Martin A. Keane[4]

1) Computer Science Department, Stanford University
Stanford, California 94305 USA
koza@cs.stanford.edu http://www-cs-faculty.stanford.edu/~koza/

2) Visiting Scholar, Computer Science Department,
Stanford University

3) Computer Science Department, University of California, Berkeley, California

4) Econometrics Inc., 5733 West Grover
Chicago, IL 60630 USA

Abstract: Most practical electrical circuits contain modular substructures that are repeatedly used to create the overall circuit. Genetic programming with automatically defined functions and the recently developed architecture-altering operations provides a way to build complex structures with reused substructures. In this paper, we successfully evolved a design for a two-band crossover (woofer and tweeter) filter with a crossover frequency of 2,512

Hz. Both the topology and the sizing (numerical values) for each component of the circuit were evolved during the run. The evolved circuit contained three different noteworthy substructures. One substructure was invoked five times thereby illustrating reuse. A second substructure was invoked with different numerical arguments. This second substructure illustrates parameterized reuse because different numerical values were assigned to the components in the different instantiations of the substructure. A third substructure was invoked as part of a hierarchy, thereby illustrating hierarchical reuse.

1. Introduction

Computer programs are replete with modular substructures that are repeatedly used within the overall program. For example, segments of useful computer code are typically encapsulated as subroutines and then reused. Moreover, subroutines may be called with different instantiations of their formal parameters (dummy variables), thereby creating parameterized reuse. In addition, subroutines may be called hierarchically, thereby creating hierarchical reuse.

Practical electrical circuits are replete with modular substructures that are repeatedly used (sometimes with different numerical values) within the overall circuit. These observations suggest that a mechanism for the automated design of a complex structure such as an electrical circuit should incorporate some kind of mechanism to exploit such modularities by reuse, parameterized reuse, and hierarchical reuse.

2. Genetic Programming

Genetic programming is an extension of the genetic algorithm described in John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975).

The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992) provides evidence that genetic programming can solve, or approximately solve, a variety of problems. Additional details are in Bennett, Koza, Andre, and Keane in this volume. See also Koza and Rice 1992.

The book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a, 1994b) describes a way to evolve multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms. Specifically, an *automatically defined function* (*ADF*) is a function (i.e., subroutine, subprogram, DEFUN, procedure, module) that is dynamically evolved during a run of genetic programming and which may be called by a calling program (or subprogram) that is concurrently being evolved. When automatically defined functions are being used, a program in the population consists of a hierarchy of one (or more) *reusable* function-defining branches (i.e., automatically defined functions) along with a main result-producing branch. Typically, the automatically defined functions possess one or more dummy arguments (formal parameters) and are reused with different instantiations of these dummy arguments. During a run, genetic programming evolves different subprograms in the function-defining branches of the overall program, different main programs in the result-producing branch, different instantiations of the dummy arguments of the automatically defined functions in the function-defining branches, and different hierarchical references between the branches.

Genetic programming with automatically defined functions has been shown to be capable of solving numerous problems. More importantly, the evidence so far indicates that, for many problems, genetic programming requires less computational effort (i.e., fewer fitness evaluations to yield a solution with a satisfactorily high probability) with automatically defined functions than without them (provided the difficulty of the problem is above a certain relatively low break-even point). Also, genetic programming usually yields solutions with smaller average overall size with automatically defined functions than without them (provided, again, that the problem is not too simple). That is, both learning efficiency and parsimony appear to be properties of genetic programming with automatically defined functions.

Moreover, there is also evidence that genetic programming with automatically defined functions is scalable. For several problems for which a progression of scaled-up versions was studied, the computational effort increases as a function of problem size at a *slower rate* with automatically defined functions than without them. In addition, the average size of solutions similarly increases as a function of problem size at a *slower rate* with automatically defined functions than without them. This observed scalability results from the profitable reuse of hierarchically-callable, parameterized subprograms within the overall program.

When automatically defined functions are being evolved in a run of genetic programming, the architecture of the overall program must be determined in some way. The specification of the architecture consists of (a) the number of function-defining branches (i.e., automatically defined functions) in the overall program, (b) the number of arguments (if any) possessed by each function-defining branch, and (c) if there is more than one function-defining branch, the nature of the hierarchical references (if any) allowed between the function-defining branches (and between the function-defining branches and the result-producing branch(es).

The user may supply the specification of this architectural information as a preparatory step that occurs prior to executing the run of genetic programming. However, it is preferable, in many situations, to automate these architectural decisions so that the user is not required to prespecify the architecture. Recent work on genetic programming has demonstrated that it is possible to evolve the architecture of an overall program dynamically during a run of genetic programming using six recently developed architecture-altering operations, namely branch duplication, argument duplication, branch deletion, argument deletion, branch creation, and argument creation (Koza 1995). These architecture-altering operations provide an automated way to enable genetic programming to dynamically determine, during the run, whether or not to employ function-defining branches, how many function-defining branches to employ, and the number of arguments possessed by each function-defining branch. The architecture-altering operations and automatically defined functions together provide an automated way to decompose a problem into a non-prespecified number of subproblems of non-pre-specified dimensionality; to solve the subproblems; and to assemble the solutions of the subproblems into a solution of the overall problem.

The six architecture-altering operations are motivated by the naturally occurring mechanisms of gene duplication and gene deletion in chromosome strings as described in Susumu Ohno's book *Evolution by Gene Duplication* (1970). In that book, Ohno advanced the thesis that the creation of new proteins (and hence new structures and new behaviors in living things) begins with a gene duplication.

Recent work in the field of genetic programming is described in Kinnear 1994, Angeline and Kinnear 1996, Koza, Goldberg, Fogel, and Riolo 1996.

3. The Problem of Circuit Synthesis

The problem of circuit synthesis involves designing an electrical circuit that satisfies user-specified design goals. A complete design of an electrical circuit includes both its topology and the sizing of all its components. The *topology* of a circuit consists of the number of components in the circuit, the type of each component, and a list of all the connections between the components. The *sizing* of a circuit consists of the component value(s) of each component.

Evolvable hardware is one approach to automated circuit synthesis. Early pioneering work in this field includes that of Higuchi, Niwa, Tanaka, Iba, de Garis, and Furuya (1993a, 1993b); Hemmi, Mizoguchi, and Shimohara (1994); Mizoguchi, Hemmi, and Shimohara (1994); and the work presented at the 1995 workshop on evolvable hardware in Lausanne (Sanchez and Tomassini 1996).

The design of analog circuits and mixed analog-digital circuits has not proved to be amenable to automation (Rutenbar 1993). CMOS operational amplifier (op amp) circuits have been designed using a modified version of the genetic algorithm (Kruiskamp 1996; Kruiskamp and Leenaerts 1995); however, the topology of each op amp was one of 24 topologies based on the conventional human-designed stages of an op amp. Thompson (1996) used a genetic algorithm to evolve a frequency discriminator on a Xilinx 6216 reconfigurable processor. In Gruau's innovative cellular encoding technique (1996), genetic programming is used to evolve the architecture, weights, thresholds, and biases of neurons in a neural network.

4. Circuit Synthesis Using Evolution

Genetic programming can be applied to circuits if a mapping is established between the kind of rooted, point-labeled trees with ordered branches used in genetic programming and the line-labeled cyclic graphs encountered in the world of circuits. Developmental biology provides the motivation for this mapping. The starting point of the growth process used herein is a very simple embryonic electrical circuit. The embryonic circuit contains certain fixed parts appropriate to the problem at hand and certain wires that are capable of subsequent modification. An electrical circuit is progressively developed by applying the functions in a circuit-constructing program tree to the modifiable wires of the embryonic circuit (and, later, to both the modifiable wires and other components of the successor circuits).

These functions manipulate the embryonic circuit (and its successors) so as to produce valid electrical circuits at each step. The functions are divided into four categories: (1) connection-modifying functions that modify the topology of the circuit (starting with the embryonic circuit), and (2) component-creating functions that insert components into the topology of the circuit, (3) arithmetic-performing functions that appear in arithmetic-performing subtrees as argument(s) to the component-creating functions and specify the numerical value of the component, and (4) calls to automatically defined functions.

Each branch of the program tree is created in accordance with a constrained syntactic structure. Branches are composed from construction-continuing subtree(s) that continue the developmental process and arithmetic-performing subtree(s) that determine the numerical value of the component. Connection-modifying functions have one or more construction-continuing subtrees, but no arithmetic-performing subtrees. Component-creating functions have one construction-continuing subtree and typically have one arithmetic-performing subtree. Structure-preserving crossover with point typing (Koza 1994a). then preserves the constrained syntactic structure.

4.1. The Embryonic Circuit

The developmental process for converting a program tree into an electrical circuit begins with an embryonic circuit.

The bottom of figure 1 shows an embryonic circuit for a one-input, two-output circuit. The energy source is a 2 volt sinusoidal voltage source VSOURCE whose negative (–) end is connected to node 0 (ground) and whose positive (+) end is connected to node 1. There is a source resistor RSOURCE between nodes 1 and 2. There is a modifiable wire (i.e., a wire with a writing head) Z0 between nodes 2 and 3, a second modifiable wire Z1 between nodes 2 and 6, and third modifiable wire Z2 between nodes 3 and 6. There is an isolating wire ZOUT1 between nodes 3 and 4, a voltage probe labeled VOUT1 at node 4, and a fixed load resistor RLOAD1 between nodes 4 and ground. Also, there is an isolating wire ZOUT2 between nodes 6 and 5, a voltage probe labeled VOUT2 at node 5, and a load resistor RLOAD2 between nodes 5 and ground. All three resistors are 0.00794 Kilo Ohms.

All of the above elements of this embryonic circuit (except Z0, Z1, and Z2) are fixed forever; they are not subject to modification during the process of developing the circuit. Note that little domain knowledge went into this embryonic circuit. Specifically, (1) the embryonic circuit is a circuit, (2) this embryonic circuit has one input and two outputs, and (3) there are modifiable connections Z0, Z1, and Z2 providing full point-to-point connectivity between the one input (node 2) and the two outputs VOUT1 and VOUT2 (nodes 4 and 5).

A circuit is developed by modifying the component to which a writing head is pointing in accordance with the associated function in the circuit-constructing program tree. The figure shows L, C, and C functions just below the LIST and three writing heads pointing to Z0, Z1, and Z2. The L, C, and C functions will cause Z0, Z1, and Z2 to become an inductor and two capacitors, respectively.

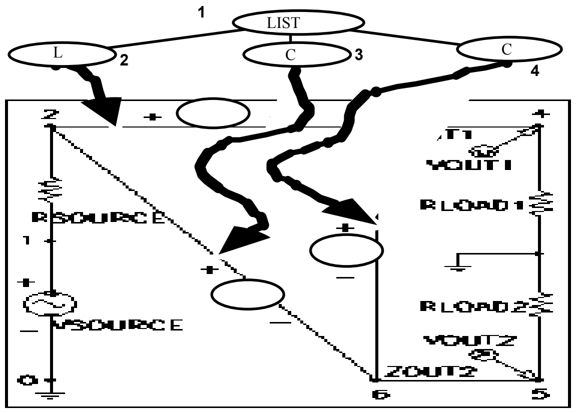

Figure 1 One-input, two-output embryonic electrical circuit.

4.2. Component-Creating Functions

Each circuit-constructing program tree in the population contains component-creating functions and connection-modifying functions. Each component-creating function inserts a component into the developing circuit and assigns component value(s) to the component. We use the inductor-creating L function and the capacitor-creating C function in this paper. Space here does not permit a detailed description of these functions (or the functions in the next section). For details, see Bennett, Koza, Andre, and Keane (1996) in this volume; Koza, Andre, Bennett, and Keane (1996); and Koza, Bennett, Andre, and Keane (1996a, 1996b, 1996c, 1996d).

4.3. Connection-Modifying Functions

Each connection-modifying function in a circuit-constructing program tree modifies the topology of the developing circuit. The

SERIES function creates a series composition; PSS and PSL a creates parallel composition; FLIP reverses polarity; NOP performs no operation; T_GND is a via-to-ground; PAIR_CONNECT connects points; SAFE_CUT cuts connections; and END ends growth.

5. Preparatory Steps

A *two-band crossover* (woofer and tweeter) filter is a one-input, two-output circuit that passes all frequencies below a certain specified frequency to its first output port (the woofer) and that passes all higher frequencies to a second output port (the tweeter). Our goal is to design a two-band crossover filter with a crossover frequency of 2,512 Hz.

Before applying genetic programming to circuit synthesis, the user must perform seven major preparatory steps, namely (1) identifying the embryonic circuit that is suitable for the problem, (2) determining the architecture of the overall circuit-constructing program trees, (3) identifying the terminals of the to-be-evolved programs, (4) identifying the primitive functions contained in the to-be-evolved programs, (5) creating the fitness measure, (6) choosing certain control parameters (notably population size and the maximum number of generations to be run), and (7) determining the termination criterion and method of result designation.

The one-input, two-output embryo of figure 1 is suitable for this problem.

Since the embryonic circuit has three writing heads – one associated with each of the result-producing branches – there are three result-producing branches (called RPB0, RPB1, and RPB2) in each program tree. The number of automatically defined functions, if any, will be determined by the evolutionary process using the architecture-altering operations. The automatically defined functions are called ADF0, ADF1, ... as they are created. Each program in the initial population of programs (generation 0) has a uniform architecture with three result-producing branches and no automatically defined functions.

The function sets are identical for all three result-producing branches of the program trees. The terminal sets are identical for all three result-producing branches.

For the three result-producing branches, the initial function set, $\mathcal{F}_{\text{ccs--rpb-initial}}$, for each construction-continuing subtree is

$\mathcal{F}_{\text{ccs--rpb-initial}}$ = {L, C, SERIES, PSS, PSL, FLIP, NOP, T_GND_0, T_GND_1, PAIR_CONNECT_0, PAIR_CONNECT_1}.

For the three result-producing branches, the initial terminal set, $\mathcal{T}_{\text{ccs-initial}}$, for each construction-continuing subtree is

$\mathcal{T}_{\text{ccs-rpb-initial}}$ = {END, SAFE_CUT}.

For the three result-producing branches, the function set, $\mathcal{F}_{\text{aps-rpb}}$, for each arithmetic-performing subtree is,

$\mathcal{F}_{\text{aps-rpb}}$ = {+, -}.

For the three result-producing branches, the terminal set, $\mathcal{T}_{\text{aps-rpb}}$, for each arithmetic-performing subtree consists of

$\mathcal{T}_{\text{aps-rpb}}$ = {←},

where ← represents floating-point random constants from –1.0 to +1.0.

The architecture-altering operations create new function-defining branches (automatically defined functions). The set of potential new functions, $\mathcal{F}_{\text{potential}}$, is

$\mathcal{F}_{\text{potential}}$ = {ADF0, ADF1, ...},

where ADF0, ADF1, ... are automatically defined functions.

For this problem, the set of potential new terminals, $\mathcal{T}_{\text{potential}}$, is limited to

$\mathcal{T}_{\text{potential}}$ = {ARG0},

where ARG0 is a dummy variable (formal parameter) to an automatically defined function.

For each newly created function-defining branch, the function set, $\mathcal{F}_{\text{aps-adf}}$, for an arithmetic-performing subtree is,

$\mathcal{F}_{\text{aps-adf}}$ = $\mathcal{F}_{\text{aps-rpb}}$ = {+, -}.

For each newly created function-defining branch, the terminal set, $\mathcal{T}_{\text{aps-adf}}$, for an arithmetic-performing subtree is

$\mathcal{T}_{\text{aps-adf}}$ = {←} ≈ $\mathcal{T}_{\text{potential}}$ = {←} ≈ {ARG0}.

The architecture-altering operations progressively change the function set for the construction-continuing subtrees of the three result-producing branches. After ADF0 is created,

$\mathcal{F}_{\text{ccs--rpb-0}}$ = $\mathcal{F}_{\text{ccs--rpb-initial}}$ ≈ ADF0.

After ADF1 is created,

$\mathcal{F}_{\text{ccs--rpb-1}}$ = $\mathcal{F}_{\text{ccs--rpb-initial}}$ ≈ ADF0 ≈ ADF1,

and so forth.

Since hierarchical references are to be permitted among the progressively created automatically defined function, the architecture-altering operations progressively change the function set for the construction-continuing subtrees of the function-defining branches. After ADF0 is created, the function set for the construction-continuing subtrees of ADF0 is

$\mathcal{F}_{\text{ccs--adf-0}} = \mathcal{F}_{\text{ccs--rpb-initial}}$.

However, after ADF1 is created, the function set for the construction-continuing subtrees of ADF1 is

$\mathcal{F}_{\text{ccs--rpb-1}} = \mathcal{F}_{\text{ccs--rpb-initial}} \approx$ ADF0,

and so forth. Thus, ADF1 can potentially refer to ADF0 and ADF2 can potentially refer to both ADF1 and ADF0.

The evaluation of fitness for each individual circuit-constructing program tree in the population begins with its execution. This execution applies the functions in the program tree to the embryonic circuit, thereby developing the embryonic circuit into a fully developed circuit. A netlist describing the fully developed circuit is then created. The netlist identifies each component of the circuit, the nodes to which that component is connected, and the value of that component. Each circuit is then simulated to determine its behavior. The 217,000-line SPICE (Simulation Program with Integrated Circuit Emphasis) simulation program (Quarles et al. 1994) was modified to run as a submodule within the genetic programming system.

The SPICE simulator is requested to perform an AC small signal analysis and to report the circuit's behavior at the two probe points, VOUT1 and VOUT2, for each of 101 frequency values chosen from the range between 10 Hz and 100,000 Hz. Each decade of frequency is divided into 25 parts (using a logarithmic scale), so there are 101 fitness cases for each probe point and a total of 202 fitness cases.

Fitness is the sum, over the 101 VOUT1 frequency values, of the absolute weighted deviation between the actual value of voltage that is produced by the circuit at the first probe point VOUT1 and the target value for voltage for that first probe point *plus* the sum, over the 101 VOUT2 frequency values, of the absolute weighted deviation between the actual value of voltage that is produced by the circuit at the second probe point VOUT2 and the target value for voltage for that second probe point. The smaller the value of fitness, the better. A fitness of zero represents an ideal filter. Specifically, the standardized fitness is

$$F(t) = \sum_{i=0}^{100} \left[W_1(d_1(f_i), f_i) d_1(f_i) + W_2(d_2(f_i), f_i) d_2(f_i) \right]$$

where $f(i)$ is the frequency (in Hertz) of fitness case i; $d_1(x)$ is the difference between the target and observed values at frequency x for probe point VOUT1; $d_2(x)$ is the difference between the target and observed values at frequency x for probe point VOUT2; $W_1(y,x)$ is the weighting for difference y at frequency x for probe point VOUT1; and $W_2(y,x)$ is the weighting for difference y at frequency x for VOUT2.

The fitness measure does not penalize ideal values; it slightly penalizes every acceptable deviation; and it heavily penalizes every unacceptable deviation.

Consider the woofer portion and VOUT1 first. The procedure for each of the 58 points in the woofer passband interval from 10 Hz to 1,905 Hz is as follows: If the voltage equals the ideal value of 1.0 volts in this interval, the deviation is 0.0. If the voltage is between 970 millivolts and 1,000 millivolts, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 1.0. If the voltage is less than 970 millivolts, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 10.0. This arrangement reflects the fact that the ideal voltage in the passband is 1.0 volt, the fact that a 30 millivolt shortfall satisfies the design goals of the problem, and the fact that a voltage below 970 millivolts in the passband is not acceptable.

For the 38 fitness cases representing frequencies of 3,311 and higher for the woofer stopband, the procedure is as follows: If the voltage is between 0 millivolts and 1 millivolt, the absolute value of the deviation from 0 millivolts is weighted by a factor of 1.0. If the voltage is more than 1 millivolt, the absolute value of the deviation from 0 millivolts is weighted by a factor of 10.0. This arrangement reflects the fact that the ideal voltage in the stopband is 0.0 volts, the fact that a 1 millivolt ripple above 0 millivolts is acceptable, and the fact that a voltage above 1 millivolt in the stopband is not acceptable.

For the two fitness cases at 2,089 Hz and 2,291 Hz, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 1.0. For the fitness case at 2,512 Hz, the absolute value of the deviation from 500 millivolts is weighted by a factor of 1.0. For the two fitness cases at 2,754 Hz and 3,020 Hz, the absolute value of the deviation from 0 millivolts is weighted by a factor of 1.0.

The fitness measure for the tweeter portion is a mirror image (reflected around 2,512 Hz) of the arrangement for the woofer portion.

Many of the circuits that are randomly created in the initial random population and many that are created by the crossover and mutation operations are so bizarre that they cannot be simulated by SPICE. Such circuits are assigned a high penalty value of fitness (10^8).

The population size, *M*, was 640,000. The architecture-altering operations are used sparingly on each generation. The percentage of operations on each generation after generation 5 was 86.5% one-offspring crossovers; 10% reproductions; 1% mutations; 1% branch duplications; 0% argument duplications; 0.5% branch deletions; 0.0% argument deletions; 1% branch creations; and 0% argument creations. Since we do not want to waste large amounts of computer time in early generations where only a few programs have any automatically functions at all, the percentage of operations on each generation before generation 6 was 78.0% one-offspring crossovers; 10% reproductions; 1% mutations; 5.0% branch duplications; 0% argument duplications; 1% branch deletions; 0.0% argument deletions; 5.0% branch creations; and 0% argument creations. A maximum size of 200 points was established for each of the branches in each overall program. The other minor parameters were the default values in Koza 1994a (appendix D).

This problem was run on a medium-grained parallel Parsytec computer system consisting of 64 80 MHz Power PC 601 processors arranged in a toroidal mesh with a host PC Pentium type computer. The distributed genetic algorithm was used with a population size of *Q* = 10,000 at each of the *D* = 64 demes. On each generation, four boatloads of emigrants, each consisting of *B* = 2% (the migration rate) of the node's subpopulation (selected on the basis of fitness) were dispatched to each of the four toroidally adjacent processing nodes. See Andre and Koza 1996 for details.

6. Results

There are no automatically defined functions in any of the 640,000 individuals of the initial random generation. The best individual program tree of generation 0 has a fitness of 410.3 and scores 98 hits (out of 202). Its first result-producing branch has 15 points; its second result-producing branch has 103 points; and its third result-producing branch has 12 points. Figure 2 shows the behavior of this best-of-generation circuit from generation 0 in the frequency domain. As can be seen, the intended lowpass (woofer) output VOUT1 has the desired value of 1 volt for low frequencies, but then drops off in a leisurely way and reverses and rises to around 1/2 volt for higher frequencies. The intended highpass (tweeter) output VOUT2 has the desired value of 0 volts for low frequencies but then slowly rises to only about 1/2 volt.

Automatically defined functions are created starting in generation 1; however, the first pace-setting best-of-generation individual with an automatically defined function does not appear until generation 8. The circuit for the best-of-generation individual from generation 8 has a fitness of 108.1 and scores 91 hits). Its three result-producing branches have 187, 7, and 183 points, respectively. Its one automatically defined function, ADF0, has 17 points. Figure 3 shows the behavior of the best-of-generation circuit from generation 8 in the frequency domain. Although the general shape of the two curves now resembles that of a crossover filter, the rise and fall of the two curves is far too leisurely.

The best-of-generation circuit from generation 158 has a fitness of 0.107 and scores 200 hits (out of 202). This fitness compares favorably with the fitness of 0.7807 and 192 hits achieved in a previously reported run of this problem without the architecture-altering operations (Koza, Bennett, Andre, and Keane, 1996b). Figure 4 shows the behavior of this circuit in the frequency domain.

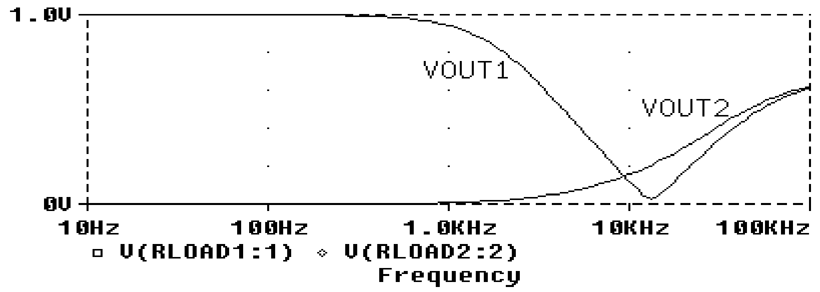

Figure 2 Frequency domain behavior of best circuit of generation 0.

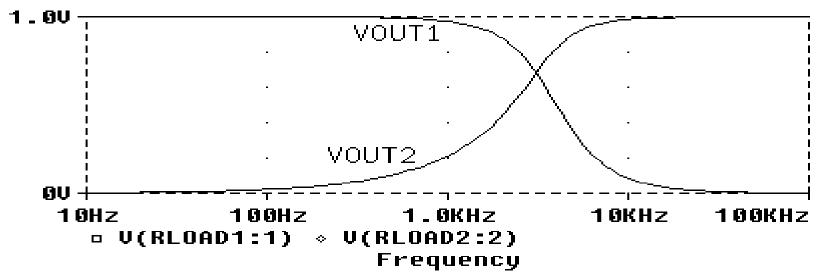

Figure 3 Frequency domain behavior of best circuit of generation 8.

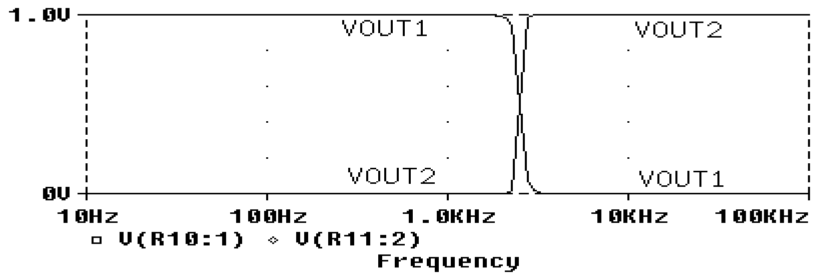

Figure 4 Frequency domain behavior of best circuit of generation 158.

Figure 5 shows the best-of-generation circuit from generation 158. Its three result-producing branches have 69, 158, and 127 points, respectively. This circuit has five automatically defined functions with 6, 24, 101, 185, and 196 points, respectively. Boxes indicate the use of `ADF2` , `ADF3`, and `ADF4`.

There is an intricate structure of reuse and hierarchical reuse of structures in this evolved circuit. Result-producing branch `RPB0` calls `ADF3` once; `RPB1` calls `ADF3` once; and `RPB2` calls `ADF4` twice. `ADF0` and `ADF1` are not called at all. `ADF2` is hierarchically called once by both `ADF3` and `ADF4`. Note that `ADF2` is called a total of five times – one time by `RPB2` directly, two times by `ADF3` (which is called once by `RPB0` and `RPB1`), and two times by `ADF4` (called twice by `RFP2`).

`ADF2` has two ports and supplies one unparameterized 259 μH inductor `L147`. Its dummy variable, `ARG0`, plays no role. `ADF0` and `ADF1` are not used.

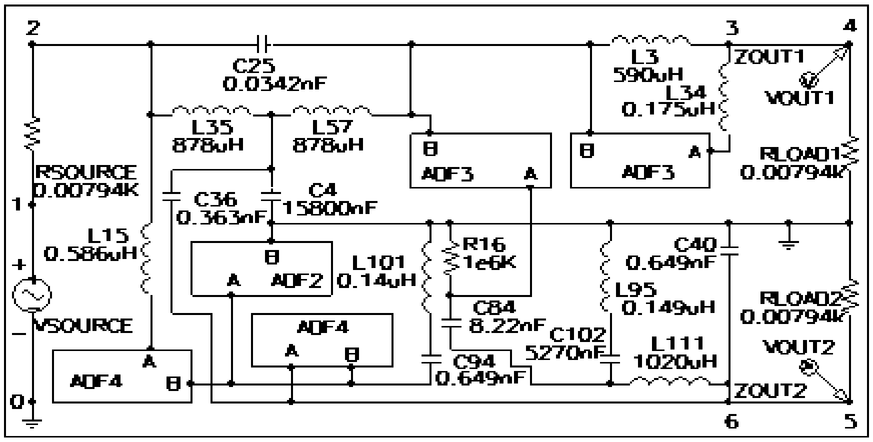

Figure 5 Best circuit of generation 158.

Figure 6 shows ADF3 of the best-of-generation circuit from generation 158. ADF3 has two ports. It supplies one unparameterized 5,130 uF capacitor C112. ADF3 is interesting in two ways. First, it has one parameterized capacitor C39 whose value is determined by ADF3's dummy variable, ARG0. Second, it has one hierarchical reference to ADF2 (which, in turn, supplies one unparameterized 259 µH inductor). Thus, the combined effect of ADF3 is to supply two capacitors (one of which is parameterized) and one inductor.

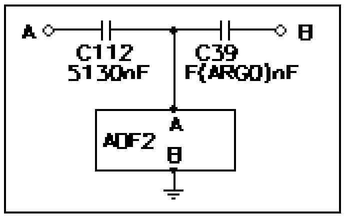

Figure 6 Automatically defined function ADF3.

Figure 7 shows ADF4 of the best-of-generation circuit from generation 158. ADF4 has three ports. It supplies one unparameterized 3,900 uF capacitor C137 and one unparameterized 5,010 uF capacitor C149. ADF4 has one hierarchical reference to ADF2 (which, in turn, supplies one unparameterized 259 µH inductor). Thus, the combined effect of ADF4 is to supply two capacitors and one inductor.

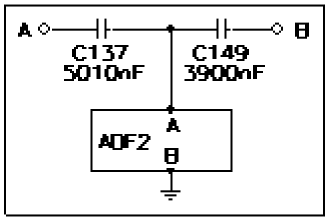

Figure 7 Three-ported automatically defined function ADF4.

An electrical engineer knows that one conventional way to realize a crossover filter is to connect a lowpass filter between the input and the first output port and to connect a highpass filter between the input and the second output port. In this neat decomposition, the only point of contact between the woofer part of the circuit feeding VOUT1 and the tweeter part feeding VOUT2 is the node that provides the incoming signal from VSOURCE and RSOURCE. There is no fitness incentive in a run of genetic programming to evolve a circuit that employs a neat decomposition of the problem into two disjoint parts. Figure 8 shows the best-of-generation circuit from generation 158 after all components have been substituted in lieu of the automatically defined functions. As can be seen, the evolved circuit is holistic in the sense that there are numerous interconnections between the parts feeding VOUT1 and VOUT2.

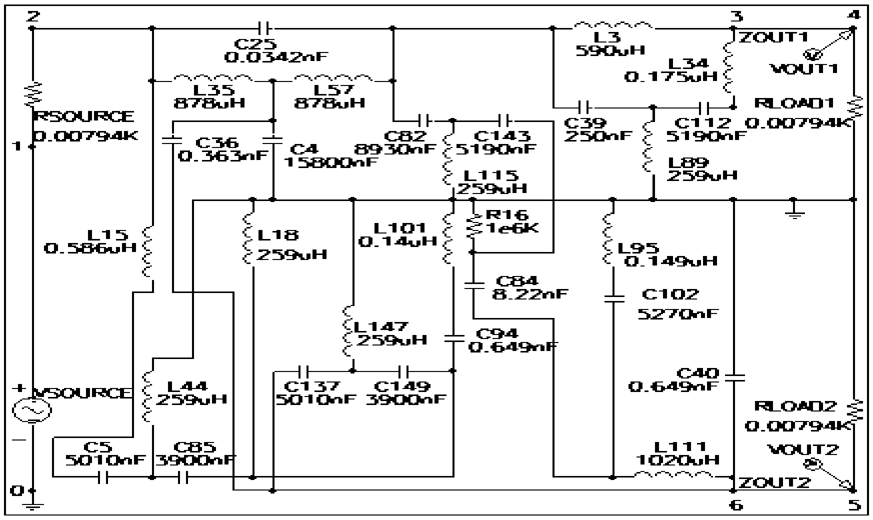

Figure 8 Best circuit of generation 158 after substitution.

7. Another Example of Reuse of Evolved Substructures Using Automatically Defined Functions

The usefulness of automatically defined functions has been demonstrated in other examples of the evolutionary design of electrical circuits. For example, in Koza, Andre, Bennett, and Keane 1996, both the topology and sizing of a fifth order elliptic (Cauer) lowpass filter was evolved.

In one run, the best circuit from generation 0 consisted of one inductor and one capacitor in the topology of one rung of the classical ladder. For a lowpass filter, this classical topology (used in Butterworth or Chebychev filters) consists of repeated instances of series inductors and vertical shunt capacitors (Van Valkenburg 1982).

When electrical engineers design Butterworth or Chebychev filters, additional rungs on the ladder (in conjunction with properly chosen numerical values for the components) generally improve the level of performance of the filter (at the expense of additional power consumption, space, and cost). As the run continued from generation to generation, the best circuit from generation 9 had the classical two-rung ladder topology. The two rungs were produced by a twice-called automatically defined function that supplied the equivalent of a 154,400 μH inductor as a two-ported substructure. The improved behavior of this circuit was a consequence of the two rungs of the ladder.

The best circuit from generation 16 consists a three-rung ladder topology and was better than its predecessors. A thrice-called automatically defined function providing a suitable inductance created the three rungs. The best circuit from generation 20 consisted of a four-rung ladder. An automatically defined function providing a suitable inductance created the four rungs of the ladder and was responsible for the improved performance.

The best circuit in generation 31 satisfied all the design requirements of the problem. An automatically defined function constructed a three-ported substructure that was used five times. This genetically evolved 100% compliant circuit is especially interesting because it had the topology of an elliptic (Cauer) filter. The circuit had the equivalent of six inductors horizontally across the top of the circuit and five vertical shunts. Each shunt consisted of an inductor and a capacitor. At the time of its invention, the Cauer filter was a significant advance (both theoretically and commercially) over the Butterworth and Chebychev filters (Van Valkenburg 1982). For example, for one illustrative set of specifications, a fifth-order elliptic filter can equal the performance of an eighth order Chebychev filter. The benefit is that the fifth order elliptic filter has one few component than the eighth order Chebychev filter.

The best circuit in generation 35 has a fitness that is about an order of magnitude better than that of the best-of-generation individual from generation 31. This 100% compliant circuit exhibits two-fold symmetry involving the repetition of four modular substructures. The symmetry of this circuit is a consequence of its quadruply-called three-ported automatically defined function. Two inductors and one capacitor form a triangle with the substructure produced by the automatically defined function. There is an additional induction element branching away from the triangle at one node.

8. Conclusion

Genetic programming with automatically defined functions and architecture-altering operations successfully evolved a design for a two-band crossover (woofer and tweeter) filter with a crossover frequency of 2,512 Hz. Both the topology and the sizing (numerical values) for each component of the circuit were evolved during the run. The evolved circuit contained three different noteworthy substructures. One substructure was invoked five times thereby illustrating reuse. A second substructure was invoked with different numerical arguments. This second substructure illustrates parameterized reuse because different numerical values were assigned to the components in the different instantiations of the substructure. A third substructure was invoked as part of a hierarchy, thereby illustrating hierarchical reuse.

Acknowledgments

Jason Lohn, Simon Handley, and Scott Brave made helpful comments on various drafts of this paper.

Related Paper in this Volume

See also Bennett, Koza, Andre, and Keane (1996) in this volume.

References

Andre, David and Koza, John R. 1996. Parallel genetic programming: A scalable implementation using the transputer architecture. In Angeline, P. J. and Kinnear, K. E. Jr. (editors). *Advances in Genetic Programming 2*. Cambridge: MIT Press.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Bennett III, Forrest H, Koza, John R., Andre, David, and Keane, Martin A. 1996. Evolution of a 60 decibel op amp using genetic programming. In this volume.

Gruau, Frederic. 1996. Artificial cellular development in optimization and compilation. In Sanchez, Eduardo and Tomassini, Marco (editors). 1996. *Towards Evolvable Hardware*. Lecture Notes in Computer Science, Volume 1062. Berlin: Springer-Verlag. Pages 48 – 75.

Hemmi, Hitoshi, Mizoguchi, Jun'ichi, and Shimohara, Katsunori. 1994. Development and evolution of hardware behaviors. In Brooks, R. and Maes, P. (editors). *Artificial Life IV* Cambridge, MA: MIT Press. Pages 371–376.

Higuchi, Tetsuya, Niwa, Tatsuya, Tanaka, Toshio, Iba, Hitoshi, de Garis, Hugo, and Furuya, Tatsumi. 1993a. In Meyer, Jean-Arcady, Roitblat, Herbert L. and Wilson, Stewart W. (editors). *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: The MIT Press. 1993. Pages 417 – 424.

Higuchi, Tetsuya, Niwa, Tatsuya, Tanaka, Toshio, Iba, Hitoshi, de Garis, Hugo, and Furuya, Tatsumi. 1993b. *Evolvable Hardware – Genetic-Based Generation of Electric Circuitry at Gate and Hardware Description Language (HDL) Levels*.

Electrotechnical Laboratory technical report 93-4. Tsukuba, Japan: Electrotechnical Laboratory.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

Koza, John R. 1995a. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In McDonnell, John R., Reynolds, Robert G., and Fogel, David B. (editors). 1995. *Evolutionary Programming IV*: *Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press. 695–717.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1996. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996a. Toward evolution of electronic animals using genetic programming. *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*. Cambridge, MA: The MIT Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996b. Four problems for which a computer program evolved by genetic programming is competitive with human performance. *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*. IEEE Press. Pages 1–10.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996c. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In Gero, John S. and Sudweeks, Fay (editors). *Artificial Intelligence in Design '96*. Dordrecht: Kluwer. 151-170.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996d. Automated WYWIWYG design of both the topology and component values of analog electrical circuits using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

Kruiskamp, Marinum Wilhelmus. 1996. *Analog design automation using genetic algorithms and polytopes*. Eindhoven, The Netherlands: Data Library Technische Universiteit Eindhoven.

Kruiskamp Marinum Wilhelmus and Leenaerts, Domine. 1995. DARWIN: CMOS opamp synthesis by means of a genetic algorithm. *Proceedings of the 32nd Design Automation Conference*. New York, NY: Association for Computing Machinery. Pages 433–438.

Mizoguchi, Junichi, Hemmi, Hitoshi, and Shimohara, Katsunori. 1994. Production genetic algorithms for automated hardware design through an evolutionary process. *Proceedings of the First IEEE Conference on Evolutionary Computation.* IEEE Press. Vol. I. 661-664.

Ohno, Susumu. 1970. *Evolution by Gene Duplication*. New York: Springer-Verlag.

Quarles, Thomas, Newton, A. R., Pederson, D. O., and Sangiovanni-Vincentelli, A. 1994. *SPICE 3 Version 3F5 User's Manual.* Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA. March 1994.

Rutenbar, R. A. 1993. Analog design automation: Where are we? Where are we going? *Proceedings of the l5th IEEE CICC.* New York: IEEE. 13.1.1-13.1.8.

Sanchez, Eduardo and Tomassini, Marco (editors).1996.*Towards Evolvable Hardware*. Lecture Notes in Computer Science, Vol. 1062. Berlin: Springer-Verlag.

Thompson, Adrian. 1996. Silicon evolution. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

Van Valkenburg, M. E. 1982. *Analog Filter Design*. Fort Worth, TX: Harcourt Brace Jovanovich.

Version 2 – Camera-Ready – Submitted October 28, 1996 for the proceedings of the first International Conference on Evolvable Systems (ICES-96) held in Tsukuba on October 7 – 8, 1996.

Reuse, Parameterized Reuse, and Hierarchical Reuse of Substructures in Evolving Electrical Circuits Using Genetic Programming

John R.Koza[1] Forrest H Bennett III[2]
David Andre[3] Martin A. Keane[4]