Use of Automatically Defined Functions and Architecture-Altering Operations in Automated Circuit Synthesis with Genetic Programming

John R. Koza
Computer Science Dept.
258 Gates Building
Stanford University
Stanford, California 94305
koza@cs.stanford.edu

David Andre
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
andre@flamingo.stanford.edu

Forrest H Bennett III
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
fhb3@slip.net

Martin A. Keane
Econometrics Inc.
5733 West Grover
Chicago, Illinois 60630
makeane@ix.netcom.com

ABSTRACT

This paper demonstrates the usefulness of automatically defined functions and architecture-altering operations in designing analog electrical circuits using genetic programming.

A design for a lowpass filter is genetically evolved in which an automatically defined function is profitably reused in the 100% compliant circuit. The symmetric reuse of an evolved substructure directly enhances the performance of the circuit. Genetic programming rediscovered the classical ladder topology used in Butterworth and Chebychev filters as well as the more complex topology used in Cauer (elliptic) filters.

A design for a double-passband filter is genetically evolved in which the architecture-altering operations discover a suitable program architecture dynamically during the run. Two automatically defined functions are profitably reused in the genetically evolved 100% complaint circuit.

1. Introduction

Human-designed electrical circuits are replete with instances of the hierarchical reuse of previously designed subcircuits. Indeed, design of a large electronic circuit would be impractical if the human designer had to start from first principles and rethink the design of each subcircuit each occasion time it is needed.

The previous paper in this volume – "Automated WYWIWYG Design of Both the Topology and Component Values of Electrical Circuits Using Genetic Programming" – demonstrated that complex structures such as electrical circuits can be evolved by means of natural selection (Koza, Bennett, Andre, and Keane 1996).

The regularity, symmetry, and reuse of substructures found in human-designed electrical circuits suggests that automatically defined functions might be useful in the problem domain of circuit synthesis. Accordingly, this paper explores the usefulness of automatically defined functions and architecture-altering operations for the problem of automated circuit synthesis.

Section 2 provides background on automatically defined functions and the architecture-altering operations. Section 3 describes the design of a lowpass filter in which an automatically defined function is profitably reused in the genetically evolved electrical circuit. Section 4 describes the design of a double-passband filter in which the architecture-altering operations successfully evolve the architecture of the solution and in which automatically defined functions are profitably reused in the genetically evolved solution.

2. Background

Genetic programming is an extension of John Holland's genetic algorithm (1975) in which the population consists of computer programs of varying sizes and shapes (Koza 1992, Koza and Rice 1992).

The book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a, 1994b) describes how to evolve multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms. This book is based on the premise that no approach to automated programming is likely to be successful on non-trivial problems unless it provides some hierarchical mechanism to exploit, by reuse and parameterization, the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments.

An *automatically defined function* (*ADF*) is a function (subroutine, subprogram, `DEFUN`, procedure, or module) that is dynamically evolved during a run of genetic programming and that 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 creates 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 (function-defining branches), and different hierarchical references between the branches.

When automatically defined functions are used, it is necessary to determine the architecture of the yet-to-be-evolved programs. The specification of the architecture consists of (a) the number of function-defining branches 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 them.

The architecture-altering operations (Koza 1994c, 1995a) provide one way to automate this architectural choice so that it can be made dynamically during a run of genetic programming. 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.

The potential usefulness of automatically defined functions and architecture-altering operations have been previously demonstrated by both proof-of-principle ("toy") problems and by non-trivial problems. For example, on the transmembrane segment identification problem (Koza and Andre 1996), the results produced by using genetic programming with automatically defined functions and the architecture-altering operations were competitive with the human-written algorithm for that problem.

3. Evolving a Lowpass Filter Using Automatically Defined Functions

A *filter* is a one-input, one-output circuit that receives a signal as its input and passes the frequency components of the incoming signal that lie in a certain specified range (the *passband*) while stopping the frequency components of the signal that lie in other frequency ranges (the *stopband*).

Consider a circuit synthesis problem in which the design goal is to design a lowpass filter using inductors and capacitors with a passband below 1,000 Hertz and a stopband above 2,000 Hz. The voltages in the filter's passband are to lie in the narrow range between 970 millivolts and 1 volt (i.e., the permissible passband ripple is 30 millivolts) and the voltages in the stopband are to lie in the narrow range between 0 volts and 1 millivolt. The circuit is to be driven from an alternating-current input signal with a 2 volt amplitude with a source resistance of 1,000 Ohms and a load resistance of 1,000 Ohms.

3.1. Preparatory Steps

This paper assumes that the reader is familiar with the use of genetic programming in electrical circuit design described in the previous paper in this volume (Koza, Bennett, Andre, and Keane 1996).

3.1.1 Embryonic Circuit and Program Architecture

A one-input, one-output embryonic circuit with two writing heads is suitable for this problem. Thus, there are two result-producing branches in each program tree in the population.

In this section of this paper, we pre-specified that the common architecture of each individual in the population would consist of four zero-argument automatically defined functions and that there would be no hierarchical references among the automatically defined functions. Consequently, the architecture of every program tree in the population throughout the entire run consists of a total of six branches joined by a `LIST` function, as shown in figure 1.

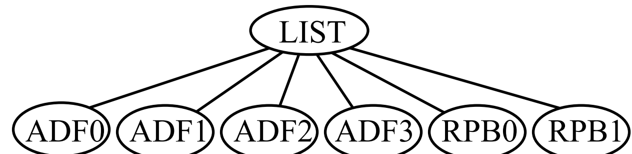


Figure 1 Program architecture.

3.1.2 Function and Terminal Sets

The function set, $\mathcal{F}_{ccs}$, for each construction-continuing subtree is

$\mathcal{F}_{ccs}$ = {`C`, `L`, `SERIES`, `PSS`, `FLIP`, `NOP`, `THGND`, `THVIA0`, `THVIA1`, `THVIA2`, `THVIA3`, `THVIA4`, `THVIA5`, `THVIA6`, `THVIA7`, `ADF0`, `ADF1`, `ADF2`, `ADF3`}.

The `C`, `L`, `SERIES`, `PSS`, `FLIP`, and `NOP` functions are defined in Koza, Bennett, Andre, and Keane (1996).

The eight three-argument "via" functions (called `THVIA0`, ..., `THVIA7`) and the three-argument "ground" function (called `THGND`) enable distant parts of a circuit to be connected together. These functions are similar to the `VIA` and the `GND` functions, except that they each create three (instead of two) modifiable wires.

`ADF0`, `ADF1`, `ADF2`, and `ADF3` are automatically defined functions.

The terminal set, $\mathcal{T}_{ccs}$, for each construction-continuing subtree is

$\mathcal{T}_{ccs}$ = {`END`, `CUT`}.

The zero-argument `CUT` function causes the highlighted component to be removed from the circuit. The `CUT` function also causes the writing head to be lost.

The function set, $\mathcal{F}_{aps}$, for each arithmetic-performing subtree is

$\mathcal{F}_{aps}$ = {+, −}.

The terminal set, $\mathcal{T}_{aps}$, for each arithmetic-performing subtree is

$\mathcal{T}_{aps} = \{\leftarrow\}$,

where $\leftarrow$ represents floating-point random constants between –1.000 and +1.000.

3.1.3 Fitness Measure

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 circuit is then created. Each circuit is then simulated to determine its behavior using a version of the SPICE3 (Quarles et al. 1994) simulator that we 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 for each of 101 frequency values chosen over five decades of frequency (from 1 Hz to 100,000 Hz). Each decade is divided into 20 parts (on a logarithmic scale).

Fitness is measured in terms of the sum, over these 101 fitness cases, of the absolute weighted deviation between the actual value of the voltage in the frequency domain that is produced by the circuit at its output probe point and the desired voltage at that point. The smaller the fitness, the better (with a fitness of zero being best).

The fitness measure is constructed so that it does not penalize ideal values; it slightly penalizes acceptable deviations; and it heavily penalizes unacceptable deviations.

Specifically, the procedure for each of the 61 points in the three-decade interval from 1 Hz to 1,000 Hz (i.e., the desired passband) is as follows: If the voltage at the output probe point 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 is acceptable, and the fact that a voltage below 970 millivolts in the passband is not acceptable.

The procedure for each of the 35 points between 2,000 Hz and 100,000 Hz (i.e., the desired stopband) is as follows: If the voltage is between 0 millivolts (the ideal value) 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 millvolt, 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 millivolts, 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.

The deviation is deemed to be zero for each of the 5 points in the interval between 1,000 Hz and 2,000 Hz (i.e., the "don't care" band of frequencies).

Hits are defined as the number (from 5 to 101) of fitness cases for which the voltage is acceptable or ideal or which lie in the "don't care" band.

3.1.4 Control Parameters

The population size, *M*, is 640,000. The crossover percentage is 89% (producing 569,600 offspring), the reproduction percentage was 10%, and the mutation percentage was 1%. A maximum size of 300 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). The problem was run on a parallel computer system with a migration rate of *B* = 2% as described in Koza and Andre 1996.

3.2. Results for the Lowpass Filter

3.2.1 Initial Random Generation

The best circuit from generation 0 has a fitness of 58.6 and scores 52 hits (out of 101). Figure 3 shows that this best-of-generation circuit from generation 0 consists of a single inductor and a single capacitor. The topological arrangement of these two components is that of the first rung of a classical ladder. Figure 9 shows the frequency domain behavior of this best-of-generation circuit from generation 0.

3.2.2 Emergence of Reuse – A Two-Rung Ladder

The best circuit from generation 9 (figure 4) has the two-rung ladder topology. The ladder topology for a lowpass filter consists of repeated instances of various *series* inductors (so named because they run "in series" horizontally across the top of the figure) and repeated instances of various vertical *shunt* capacitors. The classical Butterworth or Chebychev filters are based on the ladder topology (Van Valkenburg 1982). Automatically defined function `ADF0` supplies a group of three inductors (equivalent to one 154,400 μH inductor). Figure 2 shows this twice-called two-ported substructure developed by `ADF0`.

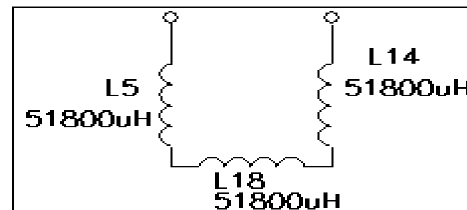


Figure 2 Twice-called two-ported automatically defined function `ADF0` from generation 9.

Figure 10 shows the behavior in the frequency domain of the best-of-generation circuit from generation 9. Figures 10 – 14 were made with 200 points per decade for greater detail.

3.2.3 A Thrice-Called Substructure Further Enhances Performance

The best circuit from generation 16 (figure 5) has a fitness of 4.1 and scores 90 hits. This circuit has a three-rung ladder topology. 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). Figure 11 shows the behavior in the frequency domain of the best circuit from generation 16. ADF0 is used three times in creating this best circuit of generation 16. Figure 15 shows this thrice-called automatically defined function.

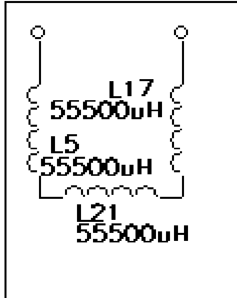


Figure 15 Thrice-called two-ported automatically defined function ADF0 from generation 16.

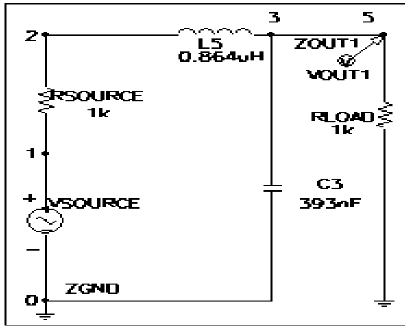


Figure 3 The best circuit from generation 0 is a one-rung ladder.

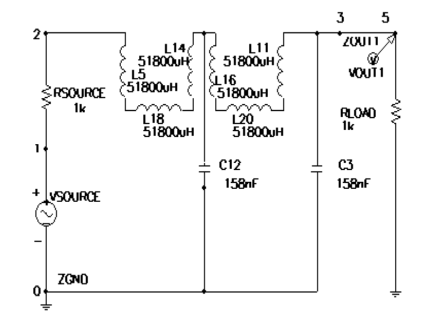


Figure 4 The best circuit from generation 9 is a two-rung ladder.

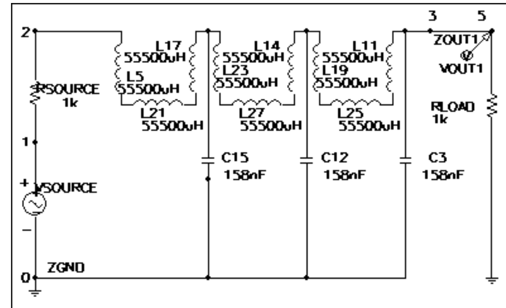


Figure 5 The best circuit of generation 16 is a three-rung ladder.

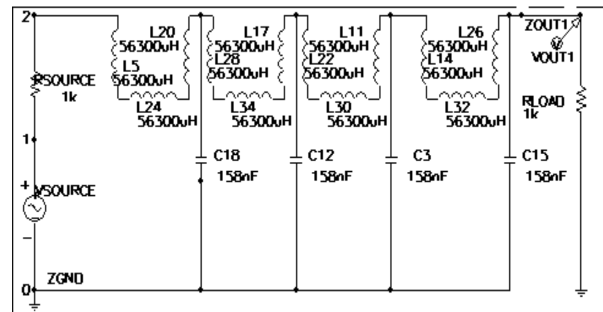


Figure 6 The best circuit of generation 20 is a four-rung ladder.

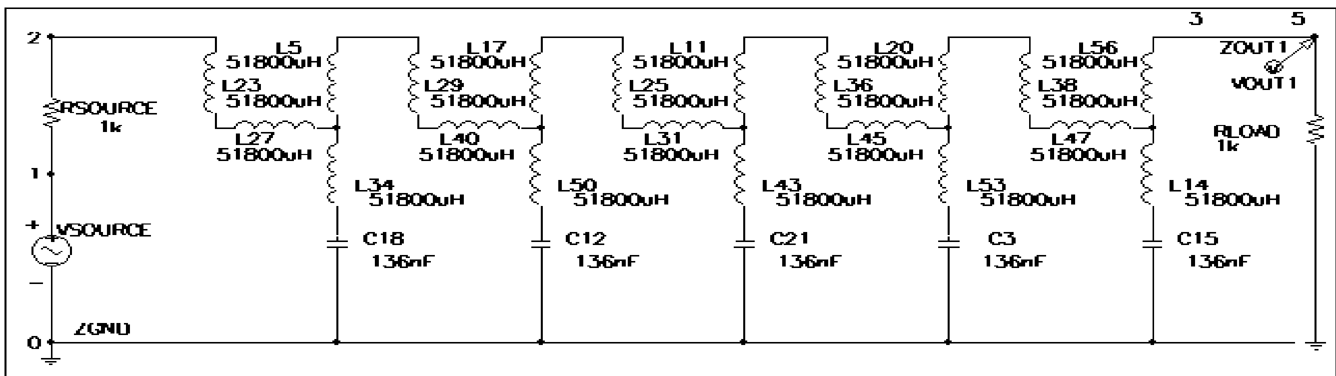


Figure 7 The 100% compliant best circuit of generation 31 has the Cauer (elliptic) topology.

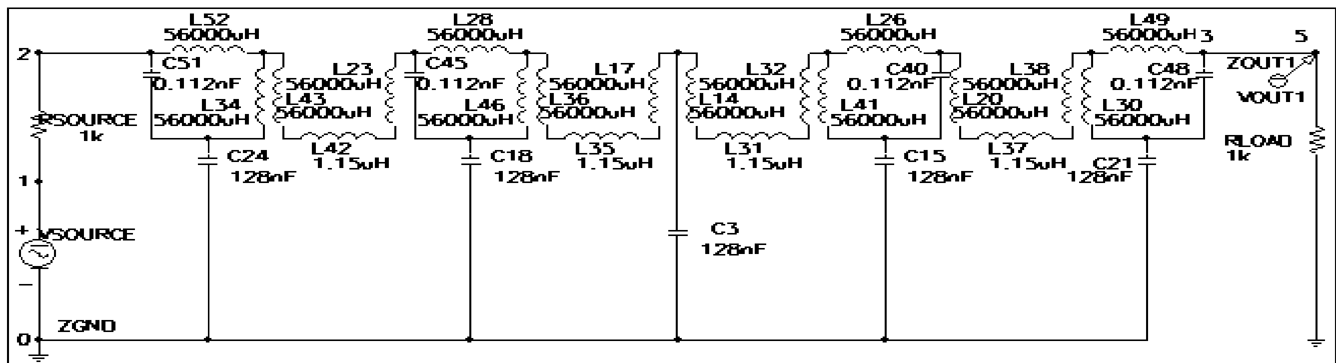


Figure 8 The 100% compliant best-of-run circuit from generation 35.

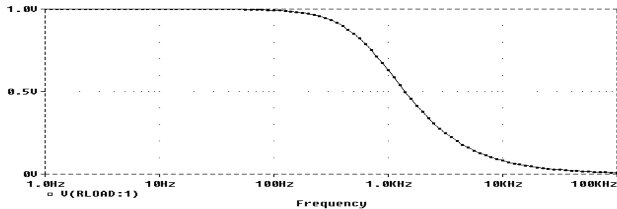


Figure 9 Frequency domain behavior of best circuit of generation 0.

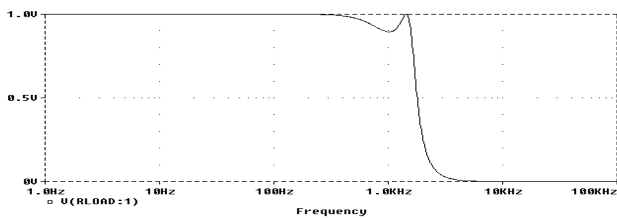
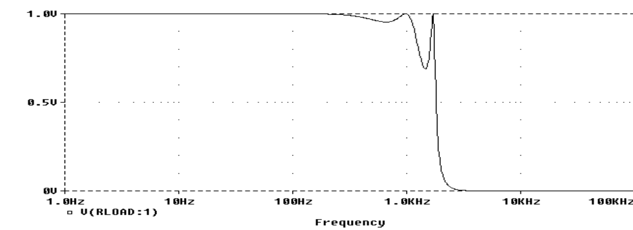


Figure 10 Frequency domain behavior of best circuit and generation 9.



9.

Figure 11 Frequency domain behavior of best circuit from generation 16.

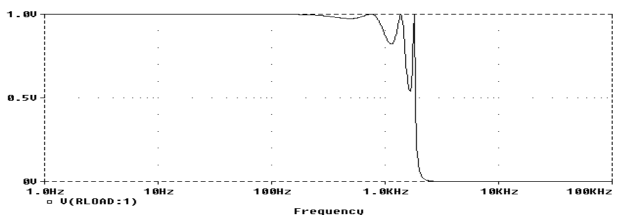


Figure 12 Frequency domain behavior of best circuit from generation 20.

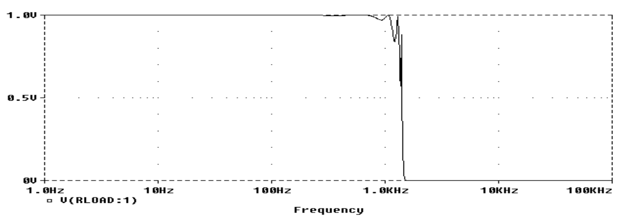


Figure 13 Frequency domain behavior of best circuit from generation 31.

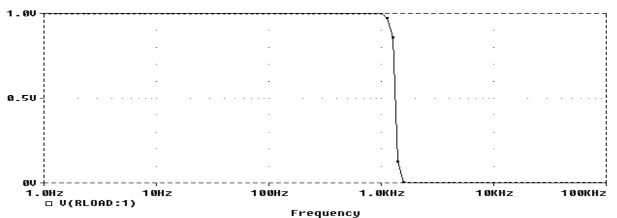


Figure 14 Frequency domain behavior of best-of-run circuit from generation 35.

3.2.4 A Quadruply-Called Substructure Further Enhances Performance

The best circuit from generation 20 (figure 6) has a fitness of 2.8 and scores 96 hits. It is a four-rung ladder.

`ADF0` (figure 16) is used four times.

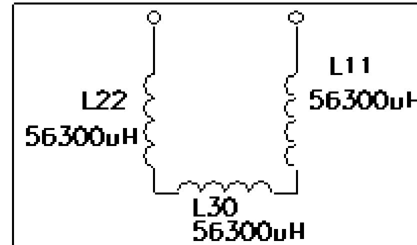


Figure 16 Quadruply-called two-ported automatically defined function `ADF0` from generation 20.

Figure 12 shows the behavior in the frequency domain of the best circuit from generation 20. The enhanced performance of this circuit is the consequence of the additional repeated calls to `ADF0`. Note that in figures 10 – 12, there is ripple in the passband, but none in the stopband (i.e., these circuits exhibit a Chebychev-like response).

3.2.5 Emergence of the Elliptic Topology Further Enhances Performance

The best circuit in generation 31 (figure 7) has a fitness of 0.0850 and scores 101 hits (i.e., is 100% compliant). This individual has 41 points in its `RPB0` and 7 points in its `RPB1`; it has 52, 297, 22, and 2 points in `ADF0`, `ADF1`, `ADF2`, and `ADF3`, respectively. Figure 13 shows its behavior in the frequency domain.

Automatically defined function `ADF0` is interesting because it constructs a three-ported substructure. This `ADF0` (figure 17) is used five times in the best circuit from generation 31.

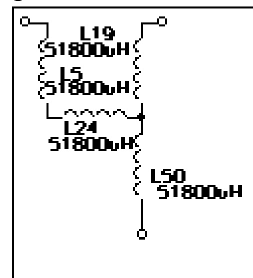


Figure 17 Quintuply-called three-ported automatically defined function `ADF0` from generation 31.

This genetically evolved 100% compliant circuit is especially interesting because it has the topology of an elliptic (Cauer) filter.

After all of the pairs and triplets of inductors are consolidated, it can be seen that the circuit has the equivalent of six inductors horizontally across the top of the circuit and five vertical shunts. Each shunt consists of an inductor and a capacitor (e.g., L34 and C18 appear in the first shunt). This is the topology of the elliptic invented by Cauer.

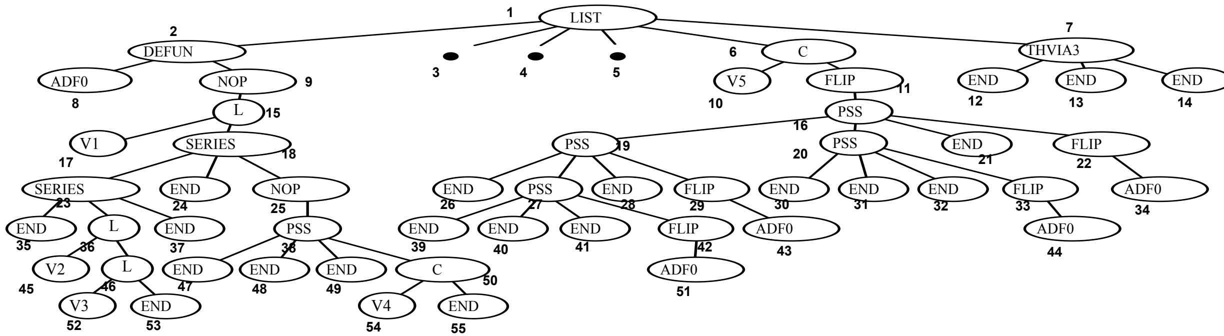


Figure 18 Best-of-run individual from generation 35.

The Cauer filter was a significant advance (both theoretically and commercially) over the Butterworth and Chebychev filters. For example, for one illustrative set of specifications, a fifth-order elliptic filter could equal the performance of 17th order Butterworth filter or an eighth order Chebychev filter. The eighth order Chebychev filter has one more component than the fifth order elliptic filter. As Van Valkenburg (1982, page 379) explains:

> "Cauer first used his new theory in solving a filter problem for the German telephone industry. His new design achieved specifications with one less inductor than had ever been done before. The world first learned of the Cauer method not through scholarly publication but through a patent disclosure, which eventually reached the Bell Laboratories. Legend has it that the entire Mathematics Department of Bell Laboratories spent the next two weeks at the New York Public library studying elliptic functions. Cauer had studied mathematics under Hilbert at Goettingen, and so elliptic functions and their applications were familiar to him."

Thus, this run of genetic programming illustrates the rediscovery of the ladder topology used in Butterworth and Chebychev filters as well as the more complex topology used in elliptic (Cauer) filters.

3.2.6 An Even Better Circuit

The best circuit in generation 35 (figure 8) has a fitness of 0.00752 (i.e., about an order of magnitude better than that of the best-of-generation individual from generation 31) and is also 100% compliant (i.e., scores 101 hits).

Only `ADF0` is referenced by the result-producing branches.

Figure 14 shows the "brick wall" behavior in the frequency domain of this best-of-run best circuit from generation 35.

Notice this circuit's two-fold symmetry involving the repetition of a total of four modular substructures. The symmetry of the best-of-run circuit from generation 35 is a consequence of its quadruply-called three-ported automatically defined function, `ADF0` (figure 19). Two inductors (L52 and L34) and one capacitor (C51) form a triangle. There is an additional induction element (specifically, a series composition of three inductors, L43, L42, and L23) branching away from the triangle at the node where inductors L52 and L34 meet.

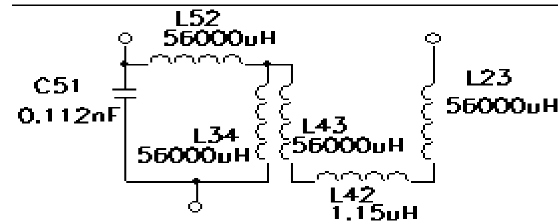


Figure 19 Quadruply-called three-ported automatically defined function `ADF0` of best-of-run circuit from generation 35.

Figure 18 presents the best-of-run individual from generation 35 as a rooted, point-labeled tree with ordered branches. Note that `ADF0` is invoked at points 34, 43, 44, and 51 of the figure. Since `ADF0` is the only function-defining branch that is actually referenced by either result-producing branch, the three unreferenced function-defining branches (`ADF1`, `ADF2`, and `ADF3`) are represented by the filled circles labeled 3, 4, and 5. The complete arithmetic-performing subtrees are not shown, but, instead, are abbreviated as `V1` through `V5`.

4. Evolving a Double-Bandpass Filter Using Architecture-Altering Operations

In the previous section, the user pre-specified the number of automatically defined functions as one of the preparatory steps for the run of genetic programming. In this section, the architecture-altering operations will be used to evolve the program architecture dynamically during the run.

The goal is to design a double-bandpass filter using inductors and capacitors as primitive components. Specifically, the goal is to design a double-bandpass filter whose first passband starts at 1,000 Hz and ends at 2,000 Hz, whose second passband starts at 1,000,000 Hz and ends at 2,000,000 Hz, and whose passband and stopband ripple are that of an elliptic filter (as detailed below).

The regularity inherent in a filter with two passbands suggests that automatically defined functions may potentially be useful for this design problem.

4.1. Preparatory Steps

4.1.1 Embryonic Circuit and Program Architecture

A one-input, one-output embryonic circuit with one writing head is suitable for this problem. Since the embroyonic circuit has one writing head, each program in the initial population of programs has one result-producing branch. Since we did not use the operation of branch creation on this problem, each program tree in the initial random population at generation 0 also has one one-argument automatically defined function. Thus, each program tree at generation 0 starts with a uniform architecture consisting of a `LIST` function joining `ADF0` and the result-producing branch, `RPB`. The number of automatically defined functions in the eventual solution will be determined dynamically, during the run, using the architecture-altering operations of branch duplication and branch deletion. Thus, starting with generation 1 of each run, the population becomes architecturally diverse.

4.1.2 Function and Terminal Sets

For any branch, the function set, $\mathcal{F}_{ccs}$, for each construction-continuing subtree is

$\mathcal{F}_{ccs}$ = {`ADF0`, `C`, `L`, `SERIES`, `PSS`, `FLIP`, `NOP`, `THGND`, `THVIA0`, `THVIA1`, `THVIA2`, `THVIA3`, `THVIA4`, `THVIA5`, `THVIA6`, `THVIA7`}.

Here `ADF0` is the first automatically defined function.

The terminal set, $\mathcal{T}_{ccs\text{-}rpb}$, for each construction-continuing subtree in the result-producing branch is

$\mathcal{T}_{ccs\text{-}rpb}$ = {`END`, `CUT`}.

The terminal set, $\mathcal{T}_{ccs\text{-}adf}$, for each construction-continuing subtree in any function-defining branch is

$\mathcal{T}_{ccs\text{-}adf}$ = {`ARG0`, `END`, `CUT`}.

Since architecture-altering operations can create automatically defined functions, the set of potential new functions, $\mathcal{F}_{potential}$, is

$\mathcal{F}_{potential}$ = {`ADF1`, `ADF2`, `ADF3`, `ADF4`}.

The function set, $\mathcal{F}_{aps}$, for an arithmetic-performing subtree in any branch and the terminal set, $\mathcal{T}_{aps}$, for an arithmetic-performing subtree in any branch are the same as in the previous section of this paper.

4.1.3 Fitness Measure

The SPICE simulator is requested to perform an AC small signal analysis and to report the circuit's behavior for each of 176 frequency values chosen over seven decades of frequency (from 10 Hz to 100,000,000 Hertz). Each decade is divided into 25 parts (using a logarithmic scale).

Fitness is measured in terms of the sum, over these 176 fitness cases, of the absolute weighted deviation between the actual value of the voltage that is produced by the circuit at the output probe point and the target value for voltage.

The frequency range is divided into two passbands, three stopbands, and four "don't care" regions.

The procedure for each of the 8 points in each of the two passbands (a total of 16 points) is as follows: If the voltage is between 960 millivolts and 1,000 millivolts (the ideal value), the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 1.0. If the voltage is less than 960 millivolts, the absolute value of the deviation from 1,000 millivolts is weighted by a factor of 10.0.

The procedure for each of the 120 points in the three stopbands is as follows: If the voltage is between 0 millivolts (the ideal value) and 40 millivolts, the absolute value of the deviation from 0 millivolts is weighted by a factor of 1.0. If the voltage is more than 40 millivolts, the absolute value of the deviation from 0 millivolts is weighted by a factor of 10.0.

There are 10 "don't care" points just before each passband and 10 "don't care" points just after each passband. The deviation is deemed to be zero for each of the 40 points in these four "don't care" bands.

Hits are defined as the number (between 40 and 176) of fitness cases for which the voltage is acceptable or ideal or that lie in one of the "don't care" bands.

4.1.4 Parameters

The control parameters were the same as in the previous section, except for the following:

(1) The architecture-altering operations are intended to be used sparingly on each generation. The percentage of operations on each generation after generation 5 was 87.5% one-offspring crossovers; 10% reproductions; 1% mutations; 1% branch duplications; 0% argument duplications; 0.5% branch deletions; 0% argument deletions; 0% 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, the percentage of operations on each generation before generation 6 was 82.0% one-offspring crossovers; 11% reproductions; 1% mutations; 5.0% branch duplications; 0% argument duplications; 1% branch deletions; 0% argument deletions; 0.0% branch creations; and 0% argument creations.

(2) The maximum number of automatically defined functions is five.

(3) The number of arguments for each automatically defined function is one.

(4) In randomly choosing functions during the creation of the initial random population, the `C`, `L`, `SERIES`, `PSS`, `FLIP`, `NOP`, `THGND`, and `ADF0` functions were each assigned a relative weight of 8, while each of the eight `THVIA` functions were assigned a relative weight of 1.

4.2. Results for Double-Bandpass Filter Using Architecture-Altering Operations

This run illustrates the progressive addition of function-defining branches and references to them.

The best circuit from generation 0 has a fitness of 720.3 and scores 54 hits; however, the result-producing branch of the program tree does not reference its 45-point automatically defined function `ADF0`.

In generation 2, the first best-of-generation individual with a referenced automatically defined function appears (with a fitness of 697.3 and 58 hits).

The first best-of-generation individual with two automatically defined functions appears in generation 22 (with a fitness of 184.7 and 110 hits); however, there are no references to either automatically defined function.

In generation 29, the first best-of-generation individual with two *referenced* automatically defined functions appears (with a fitness of 105.4 and 128 hits).

The first circuit scoring a full 176 hits is produced in generation 82 (with a fitness of 0.731).

A 100% complaint best-of-run individual emerged at generation 89 with four automatically defined functions. This individual has a fitness of 0.549 and scores 176 hits. The result-producing branch has 296 points and the function-defining branches have 82, 71, 64, and 82 points, respectively. ADF0 is called four times; ADF1 is called once; ADF2 is ignored; and ADF3 is called twice.

Figure 23 shows the best-of-run circuit from generation 89. Note that L79, C78, and L66 are superfluous. Boxes are used to indicate the parts of the overall circuit that are developed by the automatically defined functions. The letters, A, B, C, and D indicate the interface points at which the substructure specified by each automatically defined function connects to the remainder of the circuit.

Figure 20 shows the three-ported substructure developed from ADF0 at each of the four places in the result-producing branch where ADF0 is invoked. The figure shows ADF0's three ports (A, B, and C) indicating the interface points at which this substructure developed from ADF0 connects to the remainder of the circuit.

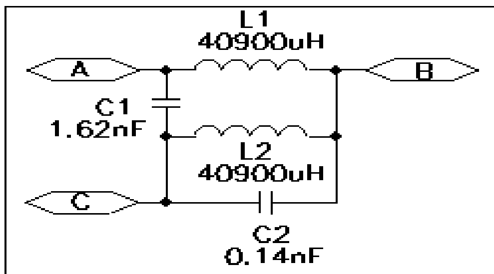


Figure 20 Three-ported quadruply-called ADF0 from generation 89.

Figure 21 shows the three-ported substructure developed from ADF1. ADF1 is invoked once.

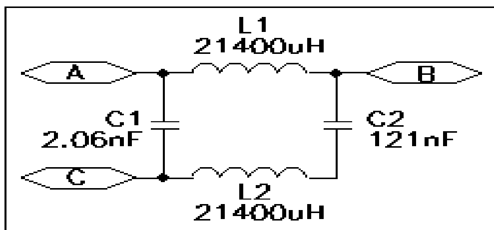


Figure 21 Three-ported ADF1 from generation 89.

Figure 22 shows the four-ported substructure developed from ADF3 at each of the two places in the result-producing branch where ADF3 is invoked. The figure shows ADF3's four ports (A, B, C, and D) indicating the interface points at which this substructure developed from ADF3 connects to the remainder of the circuit.

The insertion process for automatically defined functions (even those with no arguments) is context-sensitive. For example, in figure 23, the substructures developed by the two invocations of ADF3 are different. Specifically, one invocation employs ports A, B, and C (of figure 22) while the other employs ports A, B, and D. This difference was caused by parts of the overall program tree outside ADF3.

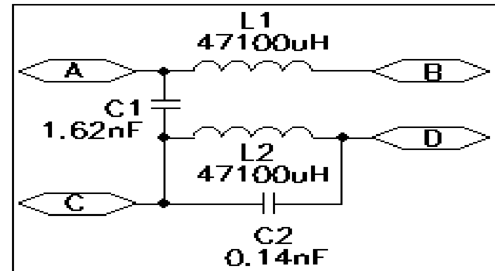


Figure 22 Four-ported twice-called ADF3 from generation 89.

Figure 24 shows the behavior in the frequency domain of the best-of-run circuit from generation 89. All 136 fitness cases that are not part of the "don't care" bands satisfy the design goals of this problem. Notice that the two small non-monotonic spikes lie in the "don't care" bands.

5. Conclusions

This paper demonstrates the usefulness of automatically defined functions and the architecture-altering operations in runs of genetic programming in which electrical circuits are being designed.

Acknowledgments

Simon Handley made helpful comments on drafts of this paper.

Bibliography

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

Holland, John H. 1975. *Adaptation in Natural and Artificial System*. Ann Arbor, MI: University of Michigan 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. 1994c. *Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming*. Stanford University Computer Science Department technical report STAN-CS-TR-94-1528. October 21, 1994.

Koza, John R. 1995. 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: MIT Press. Pages 695–717.

Koza, John R. and Andre, David. 1996. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: MIT Press. In Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996. 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). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. In this volume.

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

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, California. March 1994.

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

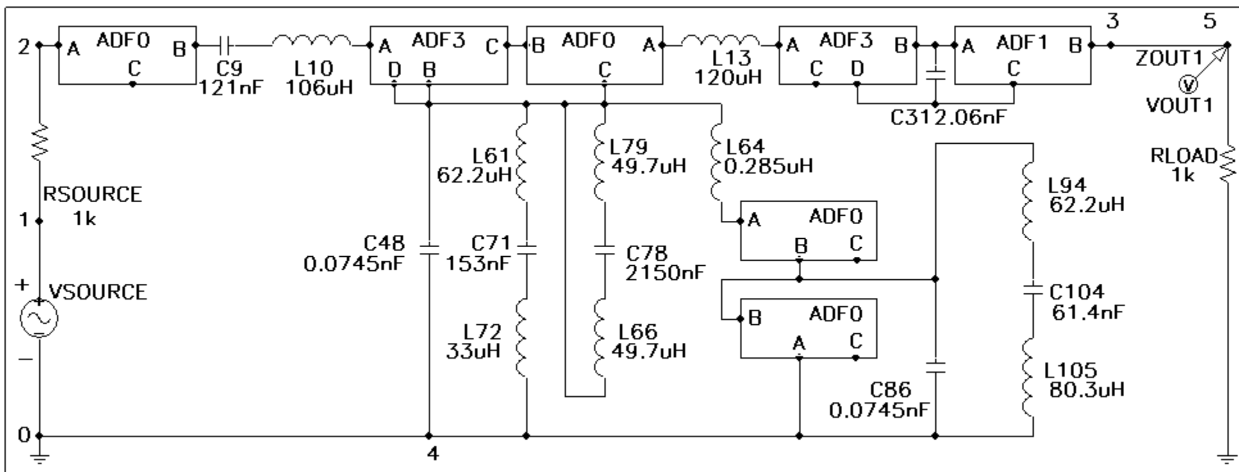


Figure 23 Best-of-run circuit from generation 89.

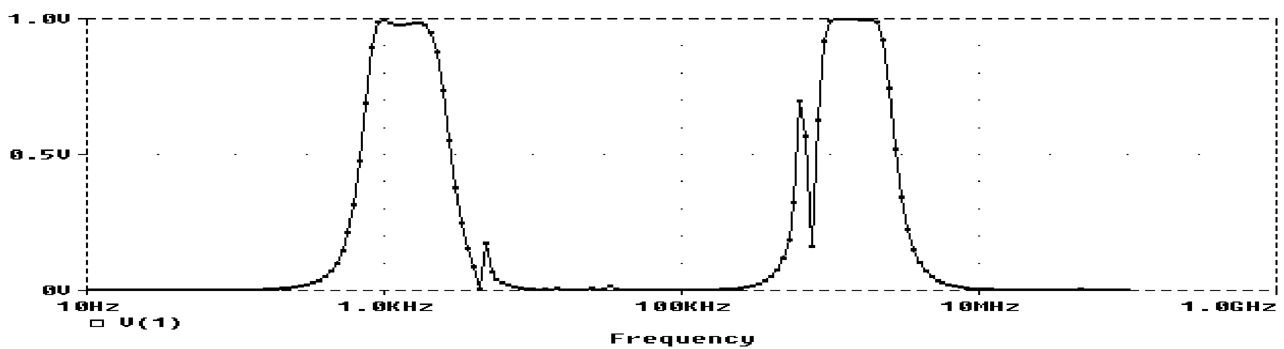


Figure 24 Frequency domain behavior of the best-of-run circuit from generation 89.