Use of Architecture-Altering Operations to Dynamically Adapt a Three-Way Analog Source Identification Circuit to Accommodate a New Source

John R. Koza
Computer Science Dept.
Stanford University
Stanford, California 94305-9020
koza@cs.stanford.edu
http://www-cs-faculty.stanford.edu/~koza/

Forrest H Bennett III
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
forrest@evolute.com

Jason Lohn
Visiting Scholar
Computer Science Dept.
Stanford University
Stanford, California 94305
jlohn7@leland.stanford.edu

Frank Dunlap
Dunlap Consulting
Palo Alto, California

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

David Andre
Computer Science Division
University of California
Berkeley, California
dandre@cs.berkeley.edu

ABSTRACT

The problem of source identification involves correctly classifying an incoming signal into a category that identifies the signal's source.

The problem is difficult because information is not provided concerning each source's distinguishing characteristics and because successive signals from the same source differ. The source identification problem can be made more difficult by dynamically changing the repertoire of sources while the problem is being solved.

We used genetic programming to evolve *both* the topology and the sizing (numerical values) for each component of an analog electrical circuit that can correctly classify an incoming analog electrical signal into three categories. Then, the repertoire of sources was dynamically changed by adding a new source during the run. The paper describes how the architecture-altering operations enabled genetic programming to adapt, during the run, to the changed environment. Specifically, a three-way source identification circuit was evolved and then adapted into a four-way classifier, during the run, thereby successfully handling the additional new source.

1. Introduction

In nature, living things exhibit considerable ability to adapt to a change in their environment by acquiring new capabilities. One mechanism that enables living things to adapt involves changes in the architecture of their genome. When we refer to architectural changes in the genome, we do not mean mere changes in the value of an allele at a particular preexisting location on the chromosome. Instead, we mean a structural change that permits the manufacture of an entirely new protein that, in turn, supports a new structure, new behavior, or new functionality.

There is an analog in the world of computer programming to a change in the architecture of the genome of a living organism. That analog consists of a change in the architecture of a computer program. When we refer to a change in architecture of a computer program, we mean a structural change in the program (i.e., a change in the number of subprograms, the number of arguments possessed by each subprogram, or the nature of the hierarchical references among the subprograms) as opposed to a mere change in the sequence of work-performing primitive operations or the number of such operations in a particular preexisting branch of the program.

The problem of source identification involves correctly classifying an incoming signal into a category that identifies the signal's source. The problem is difficult because information is not provided concerning each source's distinguishing characteristics and because successive signals from the same source differ.

The source identification problem can be made more difficult by dynamically changing the repertoire of sources while the problem is being solved. This kind of change occurs, for example, when a living organism encounters something fundamentally new and different in its environment (and must adapt to it).

This paper first considers the problem of evolving the design for an analog electrical circuit that can solve the problem of source identification for signals coming from two different sources, each emitting various signals from a certain range of frequencies. Each incoming signal is identified as coming from the first source, the second source, or neither source in this three-way version of the problem.

The paper then considers a second version of the problem in which an additional source is dynamically introduced, during the run, as soon as genetic programming successfully evolves a solution to the original three-way source identification problem. The addition of the source during the run (thereby creating a four-way source identification problem) can be viewed as a change in the environment.

One way to solve a source identification problem involves evolving the design of an analog electrical circuit that satisfies the specified goal of correctly classifying the incoming signals as to their source. Automated design (synthesis) of analog electronic circuits (involving both the circuit topology and component sizing) is recognized as a difficult problem. As Aaserud and Nielsen (1995) observe,

> "Analog designers are few and far between. In contrast to digital design, most of the analog circuits are still handcrafted by the experts or so-called 'zahs' of analog design. The design process is characterized by a combination of experience and intuition and requires a thorough knowledge of the process characteristics and the detailed specifications of the actual product.
>
> "Analog circuit design is known to be a knowledge-intensive, multiphase, iterative task, which usually stretches over a significant period of time and is performed by designers with a large portfolio of skills. It is therefore considered by many to be a form of art rather than a science."

When genetic programming was used to adapt to a changing environment during the run and solve the four-way source identification problem, we included automatically defined functions (Koza 1992, 1994). Of course, when automatically defined functions are used, we must address the question of how to determine the architecture of the to-be-evolved computer program (i.e., number of automatically defined functions, the number of arguments that they each possess, and the nature of the hierarchical references, if any, among them). Architecture-altering operations (Koza 1994c) enable genetic programming to determine the architecture of a multi-part computer program dynamically during a run. A change in the architecture of a multi-part computer program during a run of genetic programming corresponds to a change in genome structure in the natural world. Thus, we used both automatically defined functions and architecture-altering operation for the version of the problem in which the number of sources changes during the run.

The architecture-altering operations for genetic programming are motivated by the naturally occurring mechanisms of gene duplication and gene deletion in chromosome strings as described in Susumu Ohno's seminal book *Evolution by Gene Duplication* (1970). In nature, sexual recombination ordinarily recombines a part of the chromosome of one parent with a *homologous* part of the second parent's chromosome. However, in certain rare and unpredictable occasions, recombination does not occur in this normal way. A gene duplication is an aberrant recombination event that results in the duplication of a lengthy subsequence of nucleiotide bases of the DNA.

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.

After a subsequence of nucleiotide bases that code for a particular protein becomes duplicated in the DNA, there are two identical ways of manufacturing the same protein (but no immediate change in the set of proteins that are manufactured). However, over time, some other genetic operation, such as mutation or crossover, may change one or the other of the two initially identical genes. Over short periods of time, the changes accumulating in a gene may be of no practical effect or value. As long as one of the two genes remains unchanged, the original protein manufactured from the unchanged gene continues to be manufactured and the structure and behavior of the organism involved may continue as before. The changed gene is simply carried along in the DNA from generation to generation.

Natural selection exerts a powerful force in favor of maintaining a gene that encodes for the manufacture of a protein that is important for the survival and successful performance of the organism. However, after a gene duplication has occurred, there is usually no disadvantage associated with the loss of a *second way* of manufacturing the original protein. Consequently, natural selection usually exerts little or no pressure to maintain a second way of manufacturing a particular protein. Over time, the second gene may accumulate additional changes and diverge more and more from the original gene. Eventually the changed gene may lead to the manufacture of a distinctly new and different protein that actually does affect (advantageously or disadvantageously) the structure and behavior of the living thing. When a changed gene leads to the manufacture of a viable and advantageous new protein, natural selection again works to preserve that new gene.

Ohno also points out that ordinary point mutation and crossover are insufficient to explain major changes.

> "...while allelic changes at already existing gene loci suffice for racial differentiation within species as well as for adaptive radiation from an immediate ancestor, they cannot account for large changes in evolution, because large changes are made possible by the acquisition of new gene loci with previously non-existent functions."

Ohno continues,

> "Only by the accumulation of *forbidden* mutations at the *active* sites can the gene locus change its basic character and become a new gene locus. An escape from the ruthless pressure of natural selection is provided by the mechanism of gene duplication. By duplication, a redundant copy of a locus is created. Natural selection often ignores such a redundant copy, and, while being ignored, it accumulates formerly forbidden mutations and is reborn as a new gene locus with a hitherto non-existent function." (Emphasis in original).

Ohno concludes,

> "Thus, gene duplication emerges as the major force of evolution."

In other words, it is gene duplication that enables living things to adapt to changing environments by acquiring new structure, new behavior, and new functionality. The analogy, in the realm of computer programming, of nature's ability to adapt to changing environments is the set of architecture-altering operations that enable an evolving program to acquire new structure, new behavior, and new functionality.

Section 2 of this paper provides background on the process of evolving analog electrical circuits using genetic programming. Section 3 presents the preparatory steps for the three-way source identification problem and section 4 presents the results. In section 5, adaptation to a changing environment is required. In section 6, the three-way source identification problem is first solved and, then, the four-way version of the problem is solved during the same run.

2. Evolution of Circuits

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, 1994a, 1994b; Koza and Rice 1992). Recent work is described in Kinnear (1994), Angeline and Kinnear (1996), and Koza, Goldberg, Fogel, and Riolo (1996).

Genetic algorithms have been applied to the problem of circuit synthesis in the past. For example, a CMOS operational amplifier (op amp) circuit was designed using the genetic algorithm with a problem-specific crossover operation (Kruiskamp and Leenaerts 1995); however, the topology of each op amp was one of 24 pre-selected topologies based on the conventional human-designed stages of an op amp. In his paper "Silicon Evolution," Thompson (1996) used a genetic algorithm to evolve a frequency discriminator on a Xilinx 6216 reconfigurable digital gate array operating in analog mode.

Genetic programming evolves computer programs that are represented as rooted, point-labeled trees with ordered branches. Genetic programming can be applied to circuits if a mapping is established between the rooted, point-labeled trees with ordered branches found in genetic programming and the line-labeled cyclic graphs germane to circuits. Gruau's innovative work on cellular encoding (1996) enables genetic programming to evolve neural networks.

The principles of developmental biology suggest a way to map program trees into circuits. The starting point of the growth process can be a very simple embryonic electrical circuit. This embryo contains certain fixed and invariant elements for the circuit that is to be designed (e.g., the number of inputs and outputs) as well as 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 to the modifiable wires and components of successor circuits).

The functions in the circuit-constructing program trees include (1) connection-modifying functions that modify the topology of the circuit, (2) component-creating functions that insert components into the circuit, (3) arithmetic-performing functions that appear in arithmetic-performing subtrees as argument(s) to the component-creating functions and that fix the component's numerical value, and possibly (4) calls to automatically defined functions (if used).

The developmental process for converting a program tree into a circuit begins with an embryonic circuit. Figure 1 shows a one-input, one-output embryonic circuit. This embryo contains a voltage source **VSOURCE** connected to nodes 0 (ground) and 1, a fixed source resistor **RSOURCE** between nodes 1 and 2, a modifiable wire **Z0** between nodes 2 and 3, a fixed isolating wire **ZOUT** between nodes 3 and 5, a fixed output point (voltage probe) **VOUT** at node 5, and a fixed load resistor **RLOAD** between nodes 5 and ground. Only the modifiable wire **Z0** is subject to modification during the developmental process.

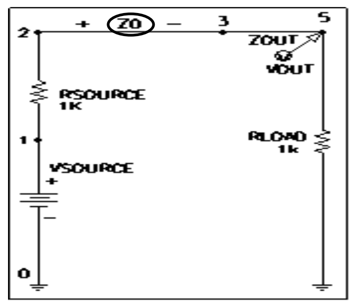

Figure 1 Embryonic circuit.

Each circuit-constructing program tree in the population contains component-creating functions and connection-modifying functions. Each connection-modifying function in a program tree points to an associated highlighted component and modifies the topology of the developing circuit. Each branch of the program tree is created in accordance with a constrained syntactic structure. Branches are composed from construction-continuing subtrees that continue the developmental process and arithmetic-performing subtrees that set the value of components. Connection-modifying functions have one or more construction-continuing subtrees, but no arithmetic-performing subtrees. Component-creating functions have one construction-continuing subtree (and often an arithmetic-performing subtree). Structure-preserving crossover with point typing preserves the constrained syntactic structure.

The component-creating functions insert a component into the developing circuit and assign component value(s) to the component. Each component-creating function has a writing head that points to an associated highlighted component in the developing circuit and modifies the highlighted component in a specified way. The construction-continuing subtree of each component-creating function points to a successor function or terminal in the circuit-constructing program tree. The arithmetic-performing subtree of a component-creating function consists of a composition of arithmetic functions and random constants that specify, after interpretation, the value of a component.

Space does not permit giving details for each component-creating and connection-modifying function. For details, see Koza, Andre, Bennett, and Keane (1996), and Koza, Bennett, Andre, and Keane (1996a, 1996b, 1996c, 1996d, 1997).

3. Preparatory Steps for the Three-Way Problem

The goal is to evolve the design for an analog electrical circuit that classifies the incoming signal into three categories. Successive incoming signals from the same source are different; however, their differences are small in comparison to signals coming from another source.

Specifically, the desired circuit is to produce an output of 1/2 volt (plus or minus 240 millivolts) if the frequency of the incoming signal is within 10% of 256 Hz, produce an output of 1 volt (plus or minus 240 millivolts) if the frequency of the incoming signal is within 10% of 2,560 Hz, and otherwise produce an output of 0 volts (plus or minus 240 millivolts). The tolerance of 240 (rather than 250) millivolts was chosen to avoid the possibility of a tie and to clearly separate the classifications.

Before applying genetic programming to a problem of 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 circuit-constructing program trees, (3) identifying the terminals, (4) identifying the primitive functions contained in the programs, (5) creating the fitness measure, (6) choosing control parameters, and (7) setting the termination criterion and method of result designation.

A one-input, one-output embryo (figure 1) was used.

We did not use automatically defined functions for the three-way source identification problem. Since the embryonic circuit has one modifiable wire (and hence one writing head), there is one result-producing branch in each circuit-constructing program tree.

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

$\mathcal{F}_{\text{ccs}}$ = {R, L, C, SERIES, PSS, PSL, FLIP, NOP, T_PAIR_CONNECT_0, T_PAIR_CONNECT_1}.

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

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

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

$\mathcal{F}$aps = {+, −}.

The terminal set for an arithmetic-performing subtree is

$\mathcal{T}$aps = {$\Re$},

where $\Re$ represents random constants from –1.0 to +1.0.

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.

For this problem, the voltage **VOUT** is probed at node 5 and the circuit is simulated in the frequency domain. SPICE is requested to perform an AC small signal analysis and to report the circuit's behavior for each of 101 frequency values chosen over four decades of frequency (between 1 and 10,000 Hz). Each decade is divided into 25 parts (using 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 output voltage at the probe point **VOUT** and the target value for voltage.

The three points that are closest to the band located within 10% of 256 Hz are 229.1 Hz, 251.2 Hz, and 275.4 Hz. The procedure for each of these three points is as follows: If the voltage equals the ideal value of 1/2 volts in this interval, the deviation is 0.0. If the voltage is within 240 millivolts of 1/2 volts, the absolute value of the deviation from 1/2 volts is weighted by a factor of 20. If the voltage is more than 240 millivolts from 1/2 volts, the absolute value of the deviation from 1/2 volts is weighted by a factor of 200. This arrangement reflects the fact that the ideal output voltage for this range of frequencies is 1/2 volt, that a 240 millivolts discrepancy is acceptable, and that a larger discrepancy is not acceptable.

The three points that are closest to the band located within 10% of 2,560 Hz are 2,291 Hz, 2,512 Hz, and 2,754 Hz. The procedure for each of these three points is as follows: If the voltage equals the ideal value of 1 volt in this interval, the deviation is 0.0. If the voltage is within 240 millivolts of 1 volt, the absolute value of the deviation from 1 volt is weighted by a factor of 20. If the voltage is more than 240 millivolts from 1 volt, the absolute value of the deviation from 1 volt is weighted by a factor of 200.

The procedure for each of the remaining 95 points is as follows: If the voltage equals the ideal value of 0 volts, the deviation is 0.0. If the voltage is within 240 millivolts of 0 volts, the absolute value of the deviation from 0 volts is weighted by a factor of 1.0. If the voltage is more than 240 millivolts from 0 volts, the absolute value of the deviation from 0 volt is weighted by a factor of 10.

Greater weights (20 and 200) were used in the two passbands because they contain only 6 of the 101 points.

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

The number of hits was defined as the number of fitness cases (0 to 101) for which the voltage is acceptable or ideal.

The population size, M, was 640,000. The percentage of genetic operations on each generation was 89% one-offspring crossovers, 10% reproductions, and 1% mutations. The architecture-altering operations were not used on this problem. Since only one result-producing branch was used in the embryo for this problem, the maximum size, H_{rpb}, for the result-producing branch was 600 points. The other parameters for controlling the runs of genetic programming were the default values specified in Koza 1994 (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 (Andre and Koza 1996).

4. Results for the Three-Way Source Identification Problem

A satisfactory solution to the problem was found on our first run of this problem.

The best circuit from generation 0 (figure 2) has a fitness of 286.2 and scores 64 hits. It has no inductors, two capacitors, and two resistors (in addition to the source and load resistors in the embryo).

Figure 5 shows the behavior of the best circuit of generation 0 in the frequency domain. The horizontal axis is logarithmic and ranges between 1 and 10,000 Hz. Notice that this inadequate circuit pays no special attention to the frequencies around 256 Hz and 2,560 Hz.

The best circuit from generation 20 (figure 3) has a fitness of 129.1 and 76 hits. Figure 6 shows its behavior. Notice the distinct areas around 256 and 2,560 Hz.

The best circuit from generation 106 (figure 4) achieves a fitness of 21.4 and scores 101 hits. It has seven inductors, 15 capacitors, and four resistors. Figure 7 shows its behavior in the frequency domain. This circuit produces an output voltage in the correct band for incoming signals from the first source, the second source, and neither source.

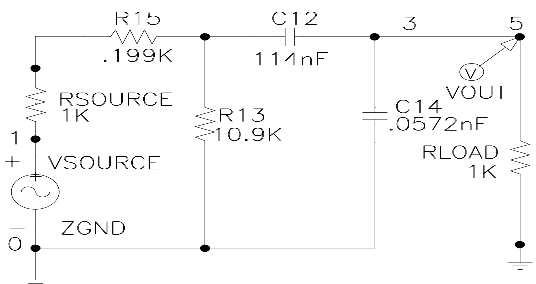

Figure 2 Best circuit of generation 0.

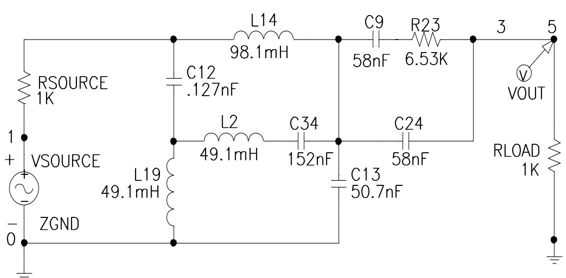

Figure 3 Best circuit of generation 20.

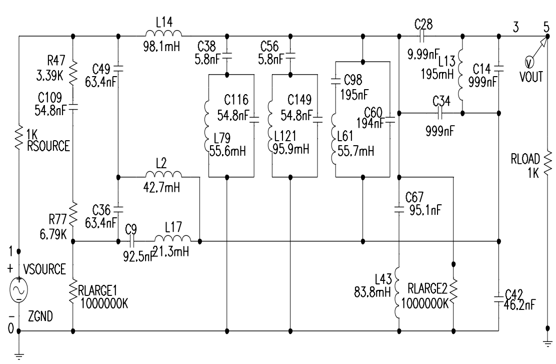

Figure 4 Best circuit of generation 106.

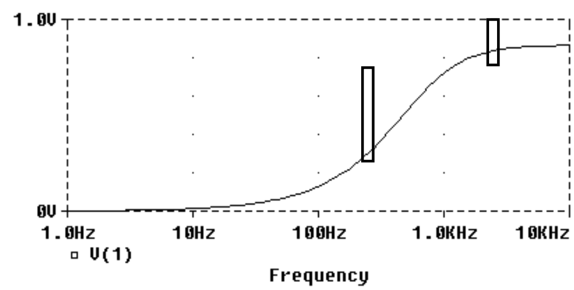

Figure 5 Frequency domain behavior of the best circuit of generation 0.

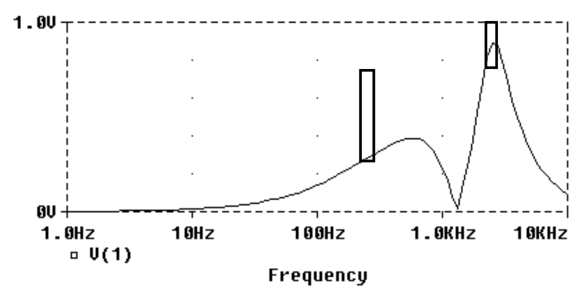

Figure 6 Frequency domain behavior of the best circuit of generation 20.

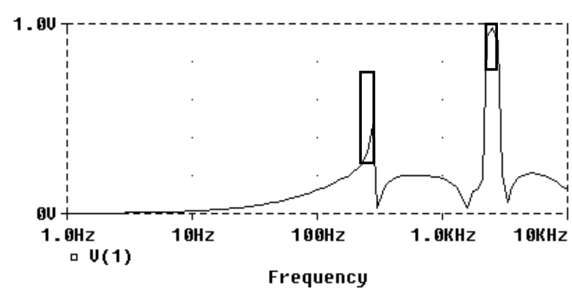

Figure 7 Frequency domain behavior of the best circuit of generation 106.

5. Preparatory Steps for the Changing Environment Problem

The goal is to evolve the design for a circuit that changes its structure as the number of different sources increases. Initially the circuit classifies incoming signals into three categories. Later the circuit undergoes modification so that it can successfully classify signals into four categories.

During the first phase, the requirements for the desired circuit are similar to those for the tri-state frequency discriminator except that one of the desired outputs is 1/3 volt (instead of 1/2 volt). Specifically, the desired circuit is to produce an output of 1/3 volts (plus or minus 166 millivolts) if the frequency of the incoming signal is within 10% of 256 Hz, produce an output of 1 volt (plus or minus 166 millivolts) if the frequency of the incoming signal is within 10% of 2,560 Hz, and otherwise produce an output of 0 volts (plus or minus 166 millivolts).

After a circuit is evolved that performs the tri-state task, the requirements are changed to include an additional frequency band. The run is continued with the existing population until a new circuit is evolved that performs the new task. Specifically, during the second phase, the circuit is to produce an output of 2/3 volts (plus or minus 166 millivolts) if the frequency of a signal is within 10% of 750 Hz while still producing an output of 1/3, 1, and 0 volts (plus or minus 166 millivolts) for the original three signals.

When genetic programming was called upon to adapt to a changing environment during the run and solve the four-way source identification problem, we included automatically defined functions. Since the embryonic circuit has one modifiable wire (and hence one writing head), there is one result-producing branch in each circuit-constructing program tree. Each program in the initial population of programs has a uniform architecture with no automatically defined functions. The number of automatically defined functions, if any, will emerge as a consequence of the evolutionary process using the architecture-altering operations.

The set of potential new functions, $\mathcal{F}_{\text{potential}}$, is

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

The set of potential new terminals, $\mathcal{T}_{\text{potential}}$, is

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

The architecture-altering operations change the function set, $\mathcal{F}_{\text{ccs}}$, for each construction-continuing subtree of the result-producing and function-defining branches, so

$\mathcal{F}_{\text{ccs}} = \mathcal{F}_{\text{ccs-initial}} \cup \mathcal{F}_{\text{potential}}$.

The architecture-altering operations change the terminal set, $\mathcal{T}_{\text{aps-adf}}$, for each arithmetic-performing subtree, so

$\mathcal{T}_{\text{aps-adf}} = \mathcal{T}_{\text{aps-initial}} \cup \mathcal{T}_{\text{potential}}$.

During the first phase, there are only two frequencies of interest (256 Hz and 2,560 Hz); however, in the second phase, there are three frequencies of interest (750 Hz in addition to the two just mentioned).

In the first phase, fitness is computed as follows.

The procedure for each of the three points that are closest to the band located within 10% of 256 Hz is as follows: If the voltage equals the ideal value of 1/3 volts in this interval, the deviation is 0.0. If the voltage is more than 166 millivolts from 1/3 volts, the absolute value of the deviation from 1/3 volts is weighted by a factor of 20. If the voltage is more than 166 millivolts from 1/3 volts, the absolute value of the deviation from 1/3 volts is weighted by a factor of 200.

The procedure for each of the three points that are closest to the band located within 10% of 2,560 is as follows: If the voltage equals the ideal value of 1 volt in this interval, the deviation is 0.0. If the voltage is within 166 millivolts of 1 volt, the absolute value of the deviation from 1 volt is weighted by a factor of 20. If the voltage is more than 166 millivolts from 1 volt, the absolute value of the deviation from 1 volt is weighted by a factor of 200.

The procedure for each of the remaining 95 points is as follows: If the voltage equals the ideal value of 0 volts, the deviation is 0.0. If the voltage is within 166 millivolts of 0 volts, the absolute value of the deviation from 0 volts is weighted by a factor of 1.0. If the voltage is more than 166 millivolts from 0 volts, the absolute value of the deviation from 0 volt is weighted by a factor of 10.

Greater weights (20 and 200) were used in the two passbands because they contain only 6 of the 101 points.

In the second phase, there is a source with a frequency of around 750 Hz.

The procedure for each of the three points that are closest to the band located within 10% of 750 Hz is as follows: If the voltage equals the ideal value of 2/3 volts in this interval, the deviation is 0.0. If the voltage is more than 166 millivolts from 2/3 volts, the absolute value of the deviation from 2/3 volts is weighted by a factor of 15. If the voltage is more than 166 mV of 2/3 volts, the absolute value of the deviation from 2/3 volts is weighted by 150.

In the second phase, the procedure for the six points nearest 256 Hz and 2,560 Hz are the same as above, except that the weight is 15 and 150 (instead of 20 and 200), respectively for the complaint and non-complaint points. Lesser weights (15 and 150) were used in the three passbands because 9 of the 101 points lie in the passbands.

In the second phase, the procedure for each of the remaining 92 points is as follows: If the voltage equals the ideal value of 0 volts, the deviation is 0.0. If the voltage is within 166 millivolts of 0 volts, the absolute value of the deviation from 0 volts is weighted by a factor of 1.0. If the voltage is more than 166 mV from 0 volts, the absolute value of the deviation from 0 is weighted by a factor of 10.

The control parameters were the same as above, except for the following: The architecture-altering operations were used sparingly on each generation. The percentage of operations on each generation after generation 5 were 86.5% one-offspring crossovers; 10% reproductions; 1% mutations; 1% branch duplications; 0.5% branch deletions; and 1% branch creations. Since we did 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; 1% branch deletions; and 5.0% branch creations. The maximum size, H_{rpb}, for the result-producing branch was 600 points. The maximum number of automatically defined functions was 2. The number of arguments for each automatically defined function is 1. The maximum size, H_{adf}, for each of the automatically defined functions, if any, is 300 points.

6. Results with the Changing Environment

The best circuit from generation 0 (figure 11) has a fitness of 200246.8 and 68 hits. Figure 14 shows its behavior.

The best circuit from generation 41 achieves a fitness of 200015.5 and 100 hits. It has 12 inductors, 13 capacitors, and two resistors (in addition to the source and load resistors in the embryo). Because of the action of the architecture-altering operations, there is one automatically defined function in the program tree for this circuit. `ADF0` is invoked three times by the result-producing branch. Figure 8 shows the best circuit from generation 41 before the three occurrences of `ADF0` are expanded.

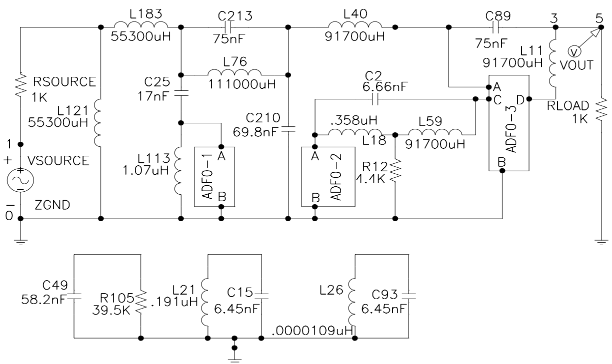

Figure 8 Best circuit from generation 41 before expanding the three occurrences of `ADF0`.

`ADF0` develops differently in different contexts. In figure 9, `ADF0` develops into one 326 nF capacitor in two instances (labeled `ADF0-1` and `ADF0-2` in figure 8).

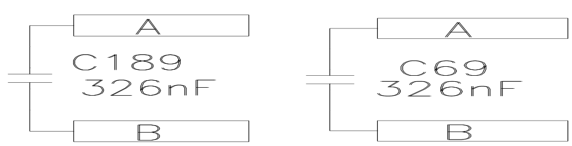

Figure 9 Result of developing `ADF0-1` and `ADF0-2` for the best circuit from generation 41.

As shown in figure 10, `ADF0` develops into two inductors and three capacitors in the third instance (labeled `ADF0-3` in figure 8).

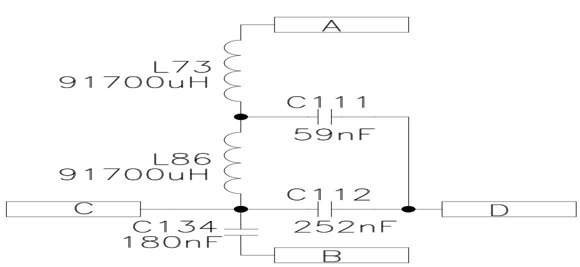

Figure 10 Result of developing `ADF0` for the best circuit from generation 41.

Figure 12 shows the best circuit from generation 41 after expanding the three occurrences of `ADF0`. Figure 15 shows the behavior of the best circuit of generation 41 in the frequency domain. Notice the emergence of two distinct peaks around 256 Hz and 2,560 Hz.

The best circuit (figure 13) from generation 85 achieves a fitness of 404.3. It scores a total of 199 hits, including all 101 hits possible from the first phase. It has 23 inductors, 20 capacitors, and five resistors (in addition to the source and load resistors in the embryo). `ADF0` is invoked twice. Figure 13 shows the best circuit from generation 85 after expanding its two automatically defined functions, `ADF0` and `ADF1`. Figure 16 shows the behavior of the best circuit of generation 85 in the frequency domain.

7. Computer Time

The run for the three-way frequency discriminator described above took 43 hours and processed about 67,840,000 individuals through the SPICE simulation and the other steps. The 64 80 MHertz processors operate together at a combined rate of 5.12 giga Hertz, so that there were about 8×10^{14} clock cycles in the run. The run for the changing environment described above took about 48 hours (about 9×10^{14} clock cycles). We make the rough approximation of one clock cycle to one computer operation and round off both of the above numbers to 10^{15} operations.

Noting that the human brain has about 10^{12} neurons operating at an approximately millisecond rate, we designate the gross quantity of 10^{15} operations as a *brain second* (1 *bs*) of computer operations. Thus, both versions of the source identification problem used about one brain second (i.e., a petaflop of operations spread over two days, instead of one second) to produce a satisfactory circuit. However, as described in *Enabling Technologies for Petaflops Computing* (Sterling, Messina, and Smith 1995), the era of petaflops computing (in which 10^{15} operations are performed in one second) is imminent.

Interestingly, six other problems solved with genetic programming and one other solved with another evolutionary algorithm have required approximately one brain second to produce a result that is arguably competitive with the result produced by humans on the same problem.

Approximately 1 brain second was required to evolve a one-dimensional cellular automata rule for the majority classification task whose accuracy (82.326%) exceeds that of the original 1978 human-written Gacs-Kurdyumov-Levin (GKL) rule, all other known subsequent human-written rules, and all other known rules produced by automated approaches for this problem (Andre, Bennett, and Koza 1996).

Also, the performance of four different versions of genetic programming (Koza 1994a, Koza and Andre 1996a, 1996b) on the transmembrane segment identification problem is slightly superior to that of algorithms written by knowledgeable human investigators. Approximately 1 brain second was required to produce each of these four results.

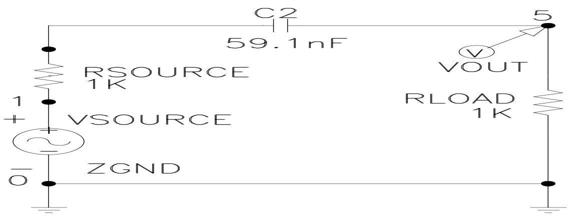

Figure 11 Best circuit from generation 0.

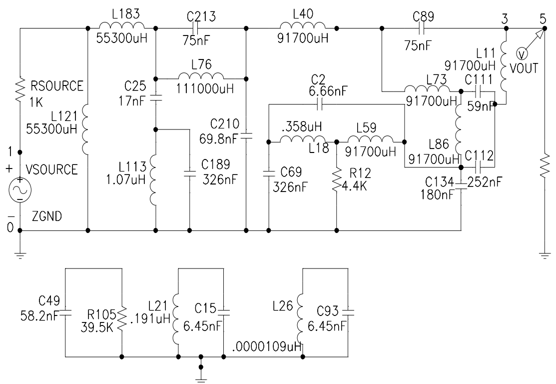

Figure 12 Best circuit from generation 41 after expanding the three occurrences of `ADF0`.

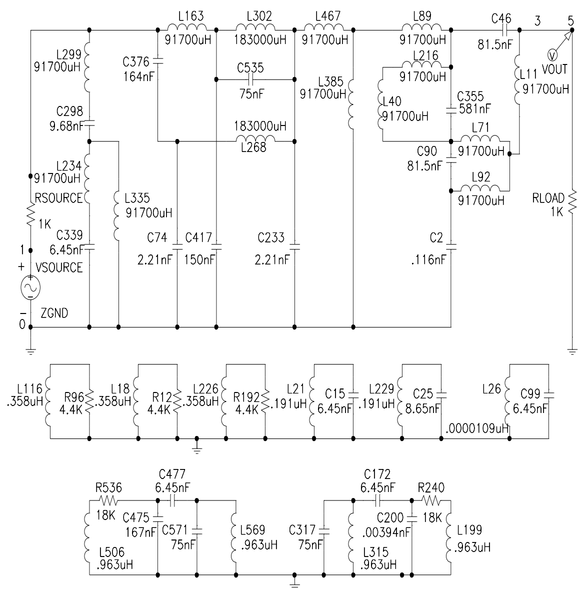

Figure 13 Best circuit from generation 85 after expanding its automatically defined functions.

In addition, approximately 1 brain second of computational effort was required for the runs of genetic programming that successfully evolved protein motifs for detecting the D-E-A-D box family of proteins and for detecting the manganese superoxide dismutase family as well or better than the comparable human-written motifs found in the PROSITE database (Koza and Andre 1996c).

Juille's discovery (1995), using evolutionary computation, of a sorting network for 13 items that was smaller than the best network in Knuth (1973) consumed approximately 0.8 brain seconds (Juillie 1997).

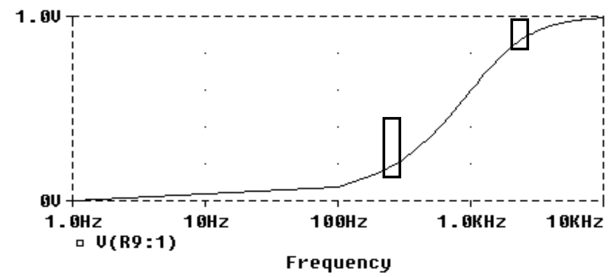

Figure 14 Frequency domain behavior of the best circuit of generation 0.

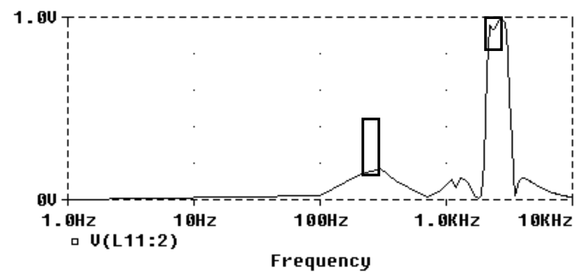

Figure 15 Frequency domain behavior of the best circuit of generation 41.

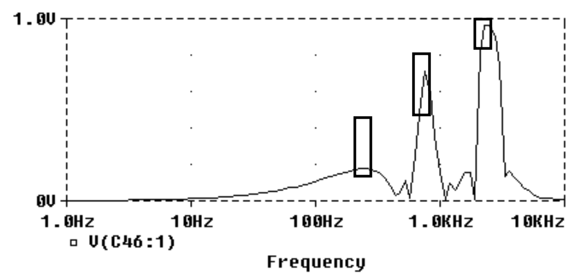

Figure 16 Frequency domain behavior of the best circuit of generation 85.

8. Conclusion

Genetic programming successfully evolved *both* the topology and the sizing for an analog electrical circuit that can perform source identification by correctly classifying an incoming analog electrical signal into three categories. Then, as the repertoire of sources was dynamically changed during the run, architecture-altering operations enabled genetic programming to adapt to a changed environment dynamically during a run. Specifically, a three-way source identification circuit was evolved and then adapted, during the run, to successfully handle the additional source.

References

Aaserud, O. and Nielsen, I. Ring. 1995. Trends in current analog design: A panel debate. *Analog Integrated Circuits and Signal Processing*. 7(1) 5-9.

Andre, David, Bennett III, Forrest H, and Koza, John R. 1996. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In Koza, John R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: MIT Press.

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). 1996. *Advances in Genetic Programming 2*. Cambridge: MIT Press.

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

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.

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

Juille, Hugues. 1995. Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces. In Eshelman, L. J. (editor). *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco: Morgan Kaufmann. 351–358.

Juille, Hugues. 1997. Personal communication.

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

Knuth, Donald E. 1973. *The Art of Computer Programming*. Vol. 3. Reading, MA: Addison-Wesley.

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: 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 Dept. technical report STAN-CS-TR-94-1528.

Koza, John R. and Andre, David. 1996a. 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 II*. Cambridge, MA: MIT Press.

Koza, John R. and Andre, David. 1996b. Evolution of iteration in genetic programming. In *Evolutionary Programming V*: *Proceedings of the Fifth Annual Conference* Cambridge, MA: MIT Press.

Koza, John R. and Andre, David. 1996c. Automatic discovery of protein motifs using genetic programming. In Yao, Xin (editor). 1996. *Evolutionary Computation: Theory and Applications*. Singapore: World Scientific.

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, D. E., Fogel, D. B., and Riolo, R. L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: 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. Pages 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, D. E., Fogel, D. B., and Riolo, R. L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: MIT Press.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1997. Evolution using genetic programming of a low-distortion 96 Decibel operational amplifier. *Proceedings of the 1997 ACM Symposium on Applied Computing, San Jose, California, February 28 – March 2, 1997*. New York: Association for Computing Machinery. Pages 207 - 216.

Koza, John R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

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

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. 433–438.

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*. Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, CA.

Sterling, Thomas, Messina, Paul, and Smith, Paul H. 1995. *Enabling Technologies for Petaflops Computing*. Cambridge, MA: The MIT Press.

Thompson, Adrian. 1996. Silicon evolution. In Koza, John R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: MIT Press.

Version 2 – **G-084** – Camera-Ready – Submitted March 25, 1997 to GP-97 Conference