The Importance of Reuse and Development in Evolvable Hardware

John R. Koza
Stanford University
Stanford, California
koza@stanford.edu

Martin A. Keane
Econometrics Inc.
Chicago, Illinois
mak@sportsmrkt.com

Matthew J. Streeter
Genetic Programming Inc.
Mountain View, California
matt@genetic-programming.com

Abstract

Reuse will become increasingly important as larger digital and analog circuits are created by the techniques of the field of evolvable hardware. This paper discusses the ways by which genetic programming can facilitate reuse and the associated advantages of using a developmental process.

1 Introduction

The field of digital and analog evolvable hardware (Higuchi, Niwa, Tanaka, Iba, de Garis, and Furuya 1993; Grimbleby 1995; Thompson 1996; Kruiskamp 1996; Koza, Bennett, Andre, and Keane 1996a, 1996b; Stoica, Zebulum, and Keymeulen 2001) has produced a progression of successful results, including those described in

- edited collections of papers (Sanchez and Tomassini 1996; Mazumder and Rudnick 1999),
- books (Kruiskamp 1996; Thompson 1998; Drechsler 1998; Zebulum, Pacheco, and Vellasco 2002),
- the proceedings of the International Conference on Evolvable Systems starting in 1996 (Higuchi, Iwata, and Lui 1997) and later conferences, and
- the proceedings of the NASA/DoD Conference on Evolvable Hardware starting in 1999 (Stoica, Keymeulen, and Lohn 1999) and later conferences.

In generating this progression of successful results, the field of evolvable hardware has employed a variety of different techniques of genetic and evolutionary computation, including

- genetic algorithms (Holland 1975),
- evolution strategies (Rechenberg 1973), and
- genetic programming (Koza 1992, 1994; Koza, Bennett, Andre, and Keane 1999; Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003).

1.1 The Importance of Reuse

Anyone who has ever looked at a blueprint for a building, a corporate organization chart, a musical score, a city map, a protein molecule, a computer program, or an electrical circuit will be struck by the massive reuse of certain basic substructures within the overall structure. Indeed, complex structures in the real world are almost always replete with modularities, symmetries, and regularities.

Reuse is important because it avoids the need to reinvent the wheel on each occasion when an already-learned substructure can be used. Reuse thus provides a means to accelerate the evolutionary process for problems entailing considerable modularity, symmetry, and regularity.

As the complexity of the problems solved by the field of evolvable hardware increases, the opportunities for profitable reuse will increase. As this occurs, genetic programming may be especially advantageous because of its ability to facilitate

(1) reusing substructures,
(2) discovering the number of substructures,
(3) discovering the nature of the hierarchical references among substructures,
(4) passing parameters to a substructure, and
(5) discovering the dimensionality of a substructure (i.e., the number of arguments possessed by the substructure).

The developmental process is of particular use in conjunction with genetic programming because of its ability to maintain syntactic validity and locality.

Section 2 discusses reuse and section 3 discusses the developmental process.

2 Reuse

2.1.1 Reuse of Substructures

Larger analog electrical circuits almost always contain multiple occurrences of certain subcircuits (e.g., voltage gain stages, Darlington emitter-follower sections, current mirrors, cascodes, voltage divider subcircuits). At a higher level, analog circuits typically contain multiple occurrences of more complex entities, such as filters, op amps, oscillators, and voltage-controlled current sources.

Similarly, large digital circuits almost always contain multiple occurrences of certain standard cells. And, large digital circuits typically contain multiple occurrences of higher-level entities (e.g., multiplexers, counters, registers).

The design of large circuits would be considerably more difficult—perhaps completely impractical—if the designer had to separately think through the design of each subcircuit from first principles on each occasion when the subcircuit is needed. Reuse enables the designer to solve a particular problem once and, thereafter, simply reuse the solution.

Reuse of a subcircuit can be realized in the context of genetic programming in several different ways.

First, the circuit-constructing functions responsible for a useful subcircuit may reside in an automatically defined function (ADF). In this kind of reuse, the automatically defined function resembles a subroutine in an ordinary computer program. Multiple occurrences of the subcircuit result when the automatically defined function is repeatedly invoked. (see chapters 27 and 28 of Koza, Bennett, Andre, and Keane 1999).

Second, the circuit-constructing functions responsible for a useful subcircuit may reside in an automatically defined copy (ADC). An automatically defined copy is an iteration specialized to problems of circuit synthesis. Multiple occurrences of the subcircuit result when the copy body branch (CBB) is repeatedly invoked by the copy control branch (CCB) of the automatically defined copy, as described in chapter 30 of Koza, Bennett, Andre, and Keane 1999.

Third, although often overlooked, the ordinary crossover operation often acts as a mechanism for reusing a subtree that is responsible for a useful subcircuit. Multiple occurrences of a useful subcircuit may be created when the subtree responsible for the useful subcircuit (being part of a reasonably fit individual) is included in a crossover fragment that crossover inserts into another (reasonably fit) individual that already produces the subcircuit.

The importance of reuse can be illustrated by means of a problem of synthesizing a lowpass filter composed of passive components with a passband boundary of 1,000 Hz, a stopband boundary of 2,000 Hz, and with 1,000-to-1 (or better) attenuation of the stopband voltage relative to the passband voltage (as described in additional detail in section 4.7 of Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003).

A *T-section* of a lowpass filter is a T-shaped subcircuit containing equally valued inductors on the two arms of the "T" and a capacitor on the vertical segment of the "T." It is often advantageous to reuse T-sections (and other sections, such as π-sections) in constructing filter circuits.

Figure 1 shows a circuit consisting of one T-section. This circuit contains a 105,500-microhenry inductor on each arm of the "T" and a 186-nanofarad capacitor on the vertical segment of the "T."

Figure 2 shows the frequency-domain behavior of the circuit (figure 1) consisting of one T-section. As can be seen, a single T-section acts as a (very poor) lowpass filter.

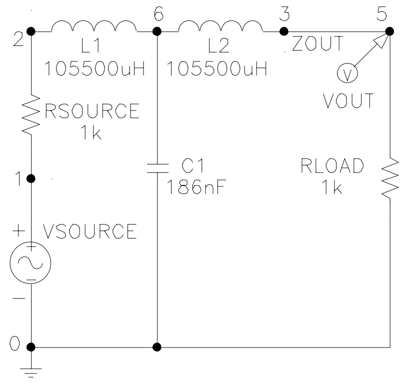

Figure 1 Circuit consisting of one T-section

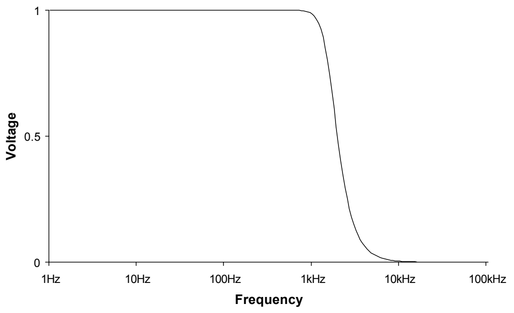

Figure 2 Frequency-domain behavior of a circuit consisting of a single T-section

One way to construct a lowpass filter satisfying this problem's requirements is by means of a cascade of *identical* T-sections in which each T-section contains one 186-nanofarad capacitor and two equal 105,500-microhenry inductors (Koza, Bennett, Andre, and Keane 1999, section 30.3.3). Figure 3 shows a circuit consisting of a cascade of two identical T-sections.

Figure 4 shows the frequency-domain behavior of a circuit (figure 3) consisting of a cascade of two identical T-sections. Although this circuit does not satisfy the problem's stated requirements, the important point is that its filtering performance is distinctly better than of the circuit consisting of only a single T-section (figures 1 and 2). That is, reuse of the T-section enhances performance.

The addition of a third, fourth, and fifth identical T-section further enhances filtering performance. For example, figure 5 shows a circuit consisting of a cascade of six identical T-sections. For reasons of space in this figure, each series pair of two 105,500-microhenry inductors is replaced by the equivalent 211,000-microhenry inductor.

Figure 6 shows the frequency-domain behavior of a circuit (figure 5) consisting of six T-sections. The filtering performance obtained from six T-sections is considerably better than that obtained from fewer T-sections (figures 2 and 4). In fact, this cascade of six identical T-sections is a 100%-complaint solution to the stated problem.

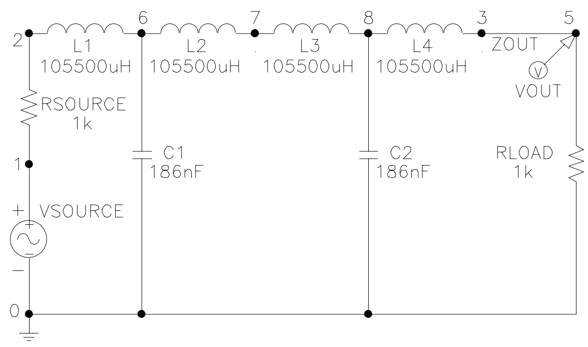

Figure 3 Circuit consisting of a cascade of two identical T-sections

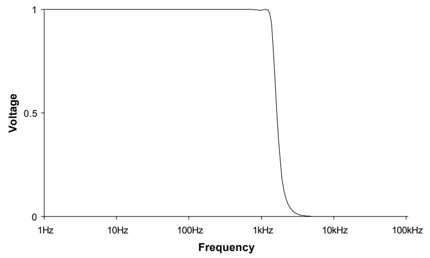

Figure 4 Frequency-domain behavior of the circuit consisting of a cascade of two identical T-sections shown in figure 3

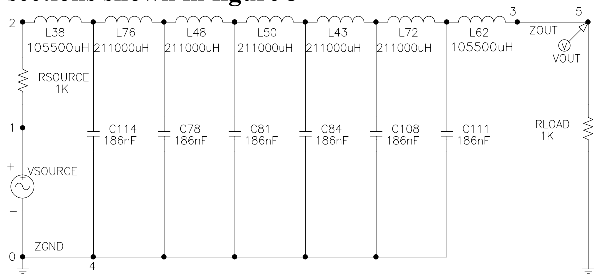

Figure 5 Circuit consisting of a cascade of six identical T-sections.

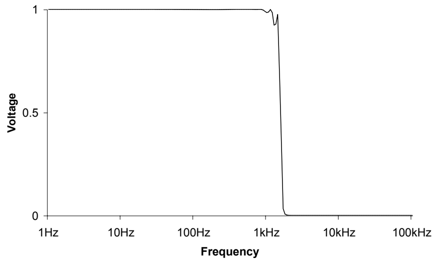

Figure 6 Frequency-domain behavior of the circuit consisting of a cascade of six identical T-sections shown in figure 5

Now consider how one might approach this problem of circuit synthesis using the genetic algorithm operating on fixed-length strings.

First, the human user would choose the maximum number of components in the circuit. In practice, the user would choose a generous maximum in order to accommodate a circuit of any size that is reasonably likely to solve the stated problem. For reasons of space in the tables below, we use an artificially low maximum (six).

Because the components involved in this problem have only two leads, each component can be conveniently represented by one symbolic variable and three numeric variables. The symbolic variable specifies the type of component (with "L" denoting an inductor, "C" denoting a capacitor, and "W" denoting a nonmodifiable wire). The first of the three numeric variables specifies the component's sizing (with inductors in millihenrys, capacitors in nanofarads, and units being irrelevant for wires). The other two numeric variables specify the nodes to which the component's leads are connected.

Of course, all symbolic and numeric variables could, at the user's option, be converted into bits. However, we will proceed herein using a chromosome in which symbolic variables ("L," "C," and "W") appear in chromosome positions of the form $4i+1$ (where i is an integer); floating-point component values appear in positions of the form $4i+2$; and integers representing nodes appear in positions of the forms $4i+3$ and $4i$.

Table 1 shows one way of representing the circuit consisting of one T-section (shown in figure 1) using the genetic algorithm operating on fixed-length strings. The first component is a 105.5-millihenry (105,500-microhenry) inductor whose leads are connected to nodes 2 (the incoming signal) and 6 (an intermediate node). The second component is a 186-nanofarad capacitor whose leads are connected to node 6 and node 0 (ground). The third component is a 105.5 millihenry inductor whose leads are connected to node 6 and node 3 (the circuit's output port). The fourth component is a nonmodifiable wire (with 0 as its component value) whose leads are connected to node 0 and node 7 (another intermediate point). The fifth component is a 100-nanofarad capacitor whose leads are connected to nodes 0 and 7. The sixth component is a 100-nanofarad capacitor whose leads are connected to node 0 and node 8 (yet another intermediate point). Note that the fourth and fifth components form an isolated loop and that the sixth component is a dangling component. However, in practice, isolated loops and dangling wires can be easily edited out by routine preprocessing.

The circuit consisting of a single T-section (represented by the chromosome in table 1 and shown in figure 1) is a lowpass filter (albeit a very poor one). It is precisely the kind of circuit that one often encounters as a best-of-generation individual in an early generation of a run of the genetic algorithm, evolution strategy, or genetic programming. This circuit, consisting of a single T-section (whose three external points are at nodes 2, 0, and 3), does not, of course, satisfy the problem's requirements. However, the T-section is a useful building block and represents progress toward the eventual solution.

Table 2 shows one possible way of representing the circuit consisting of two T-sections shown in figure 3. The first, second, and third components in this chromosome represent the first T-section. Node 6 is the internal point of the first T-section. The three external points of this first T-section are nodes 2, 0, and 7. The fourth, fifth, and sixth components in this chromosome represent the second T-section. The three external points of the second T-section are nodes 7, 0, and 3 (with node 8 being the internal point of the T-section). The fourth component is a 105.5-millihenry (105,500-microhenry) inductor whose leads are connected to intermediate nodes 7 and 8. The fifth component is a 186-nanofarad capacitor whose leads are connected to nodes 8 and 0. The sixth component is a 105.5-millihenry inductor whose leads are connected to nodes 8 and 3 (the circuit's output point). Notice that the third component in table 2 is connected to node 7 (an intermediate point), as opposed to being connected to node 3 (the circuit's output point) as it was in table 1. This change reflects the fact that the output of the first T-section in table 2 will be fed into the second T-section. Also note that the final output of the cascade of two identical T-sections is connected to node 3 (the circuit's output point).

Although the genetic algorithm operating on fixed-length strings is capable of solving the stated problem, notice that its discovery of the first T-section will not aid in discovering the second T-section. The reasons are that each T-section in the chromosome is defined in terms of particular node numbers and that the three external node numbers associated with each new T-section are necessarily different from those associated with the previous T-section. In particular, after discovering the first T-section (with external points 2, 0, and 7), the genetic algorithm operating on fixed-length strings must separately discover the second T-section (with external points 7, 0, and 3).

In contrast, representations employed by genetic programming have the ability to leverage the once-learned T-section. Consider one way in which this can happen. Suppose that a particular individual in an early generation develops into a circuit with a single T-section. During the early generations of the run, this individual will be relatively fit (when compared to cohorts lacking the T-section or some equally good structure). Thus, this individual will probably be selected (and, indeed, reselected) to participate in crossover. On some of the occasions in which this first individual is selected to participate in crossover, the subtree that is responsible for the T-section will be included in the crossover fragment. Meanwhile, a relatively fit second individual will be selected to mate with the first individual. Possibly, the second individual may be reasonably fit because it too already contains one occurrence of a T-section. Note that because reselection is allowed, the first individual may occasionally mate with itself. Thus, the ordinary crossover operation will sometimes create an offspring with two T-sections. At the moment when this doubling-up occurs, the offspring will probably have distinctly better fitness than its cohorts in the population. The offspring with two T-sections will thus immediately start winning future tournaments in which it participates and will therefore frequently participate in future reproduction, mutation, and crossover operations.

Of course, this scenario does not occur on every crossover. However, the vast majority (about 90%) of the genetic operations on a typical run of genetic programming are crossovers. Moreover, a run of genetic programming involves a large population bred over many generations, so there are numerous opportunities for doubling-up of T-sections.

Figure 7 shows a circuit-constructing program tree that develops into the single T-section shown in figure 1. In this figure, the first and third arguments of the three-argument `THREE_GROUND` function (labeled 100) each create a 105,500-microhenry inductor. The second argument of the `THREE_GROUND` function creates a capacitor connected to ground.

Table 1 A first illustrative chromosome for the circuit shown in figure 1

L	106	2	6	**C**	186	6	0	**L**	106	6	3	**W**	0	0	7	**C**	100	0	7	**C**	200	8	0

Table 2 Chromosome for circuit consisting of two identical T-sections shown in figure 3

L	106	2	6	**C**	186	6	0	**L**	106	6	7	**L**	106	7	8	**C**	186	8	0	**L**	106	8	3

Table 3 Sub-string producing a T-section

L	106	2	6	**C**	186	6	0	**L**	106	6	3

Table 4: Chromosome of first parent

L	106	2	6	**W**	0	6	7	**C**	186	7	0	**W**	0	7	8	**L**	106	8	9	**W**	0	9	3

Table 5: Chromosome of second parent

W	0	8	6	**W**	0	9	3	**W**	0	7	8	**L**	106	6	9	**C**	186	8	0	**L**	106	2	7

Table 6: Result of a crossover

L	106	2	6	**W**	0	6	7	**C**	186	7	0	**L**	106	6	9	**C**	186	8	0	**L**	106	2	7

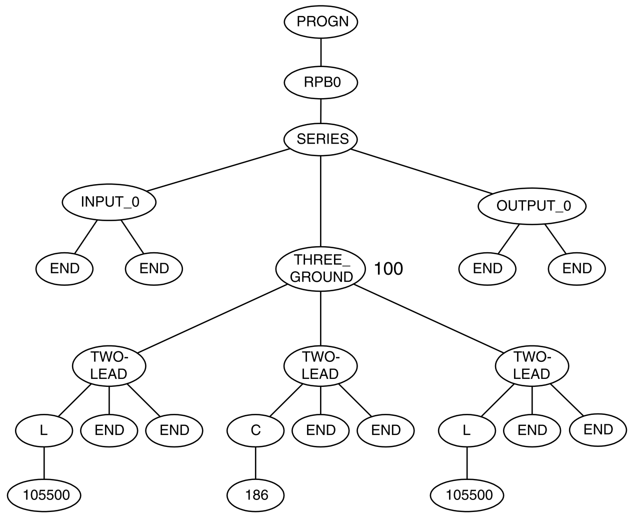

Figure 7 A circuit-constructing program tree that develops into the single T-section shown in figure 1

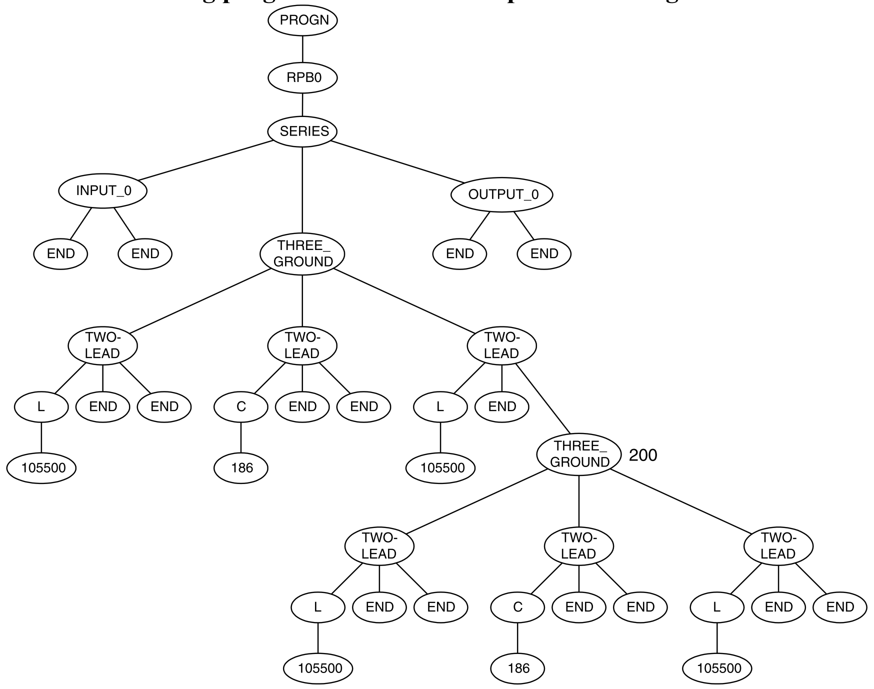

Figure 8 A circuit-constructing program tree that develops into the two T-sections shown in figure 3

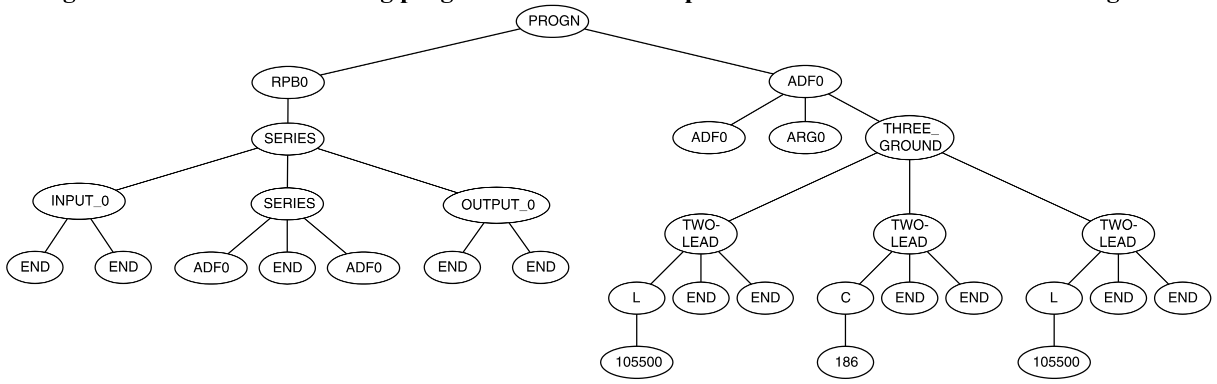

Figure 9 Circuit-constructing program tree containing automatically defined function `ADF0` that develops into the two T-sections shown in figure 3

The first argument of the three-argument SERIES function is an INPUT_0 function that causes a connection to be made between the incoming signal (node 2 in figure 1) and the first inductor. The third argument of the SERIES function is an OUPTUT_0 function that causes a connection to be made between the circuit's output point (node 3 in figure 1) and the second inductor. The second argument of the SERIES function is a THREE_GROUND function that causes a connection to be made between the ungrounded lead of the capacitor and the first and second inductor. Figure 8 shows a circuit-constructing program tree that develops into the two T-sections shown in figure 3. The upper left portion of figure 8 (i.e., everything above the point labeled 200) is identical to figure 7. As can be seen, the subtree rooted at (that is, below) the point labeled 200 in figure 8 is identical to the subtree rooted at the THREE_GROUND function labeled 100 in figure 7. That is, the subtree that produces the T-section that is responsible (during early generations of the run) for the filtering ability of the individual of figure 7 is embedded inside another individual that itself produces a single T-section. Figure 8 shows a way by which the ordinary crossover operation can create a circuit consisting of a cascade of two identical T-sections. The circuit consisting of a cascade of two identical T-sections has better fitness than the circuit consisting of just one T-section.

Automatically defined functions provide a second way by which genetic programming can leverage the already-learned utility of a T-section. Figure 9 shows a circuit-constructing program tree consisting of a result-producing branch (RPB0) on the left and an automatically defined function (ADF0) on the right. The entire program tree develops into the two T-sections shown in figure 3. Automatically defined function ADF0 develops into one T-section. The result-producing branch (RPB0) invokes ADF0 twice, thereby creating a circuit consisting of two identical T-sections.

The automatically defined copy (described in chapter 30 of Koza, Bennett, Andre, and Keane 1999) provides yet another way by which genetic programming can leverage on the already-learned utility of a T-section.

The genetic algorithm operating on fixed-length strings cannot readily leverage a previously discovered useful substructure in the just-described ways. In fact, if the (usual) homologous crossover is employed, there is no way to move a T-section attached to, say, nodes 2, 0, and 3 to another part of the chromosome. Thus, after discovering one T-section, the genetic algorithm operating on fixed-length strings must separately (and laboriously) discover the second one.

Note that the genetic algorithm operating on variable-length strings (Smith 1980) has the same difficulty. It can easily create two identical T-sections attached to the same set of nodes. For example, the sub-string producing a T-section attached to nodes 2, 0, and 3 (table 3) could easily be inserted, by crossover, into another chromosome that already contains a T-section attached to nodes 2, 0, and 3.

However, the result would be two T-sections attached to the same group of nodes (that is, 2, 0, and 3). The point is that the genetic algorithm operating on variable-length strings does not readily yield a *series* composition in which the one T-section is attached to nodes 2, 0, and 7 and the second T-section is attached to differently numbered nodes (say, nodes 7, 0, and 3). In contrast, genetic programming has the ability to easily produce, by crossover, either a series or parallel composition of T-sections (with the series composition being the more useful here).

2.1.2 The Number of Substructures

If discovering the number of substructures is an important part of the problem, genetic programming may be well suited to the problem.

Consider the problem of automatically synthesizing the double-bandpass filter described in chapter 36 of Koza, Bennett, Andre, and Keane 1999.

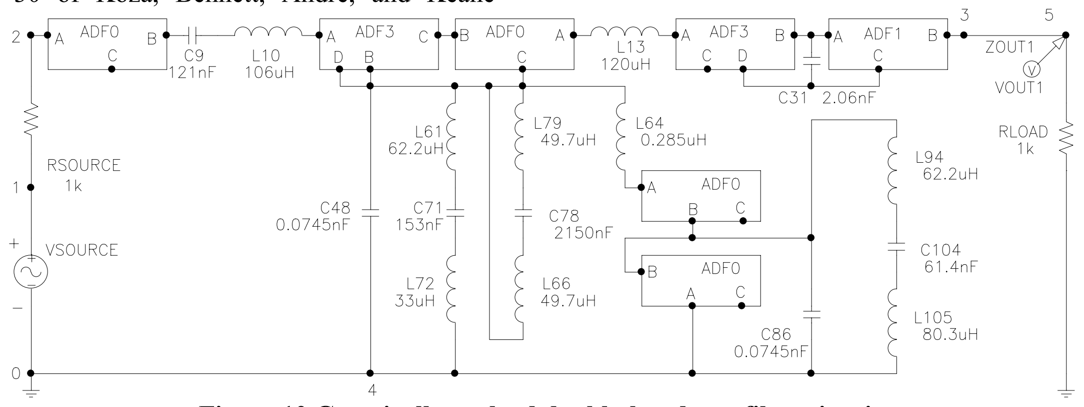

Figure 10 Genetically evolved double-bandpass filter circuit.

The architecture-altering operations enable genetic programming to automatically create the architecture of the overall program during the run.

The genetically evolved double-bandpass filter (figure 10) contains four occurrences of the three-ported automatically defined function ADF0, two occurrences of the four-ported automatically defined function ADF3, and one occurrence of the three-ported automatically defined function ADF1. The architecture-altering operations dynamically determined, during the run of genetic programming, the number of automatically defined functions that were used in solving this problem.

Automatically defined function ADF0 appears four times in the program that produced the genetically evolved double-bandpass filter (figure 10). Figure 11 shows the three-ported subcircuit produced by the execution of ADF0. Automatically defined function ADF3 appears twice in the this same program. Figure 12 shows the four-ported subcircuit produced by the execution of ADF3.

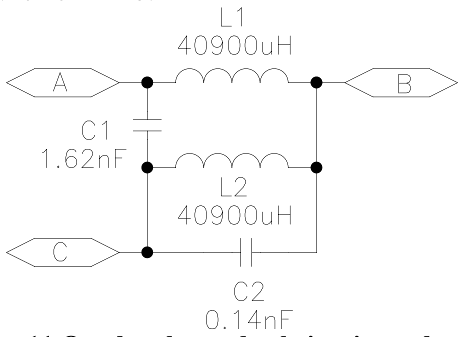

Figure 11 Quadruply-used subcircuit produced by automatically defined function ADF0

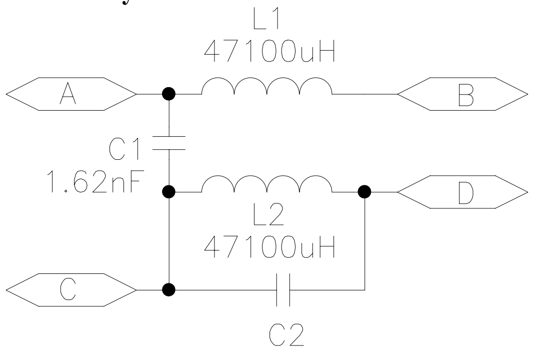

Figure 12 Twice-used four-ported subcircuit produced by automatically defined function ADF3

2.1.3 Hierarchical References among the Substructures

If discovering the nature of the hierarchical references among the substructures is a major part of the problem, genetic programming may be appropriate.

The architecture-altering operations enable genetic programming to automatically create the architecture of the overall program during the run, including the hierarchical arrangement of automatically defined functions.

Figure 13 shows the call tree for the genetically evolved crossover (woofer-tweeter) filter described in detail in chapter 33 of Koza, Bennett, Andre, and Keane 1999. This call tree shows that the circuit-constructing program tree for this filter consists of three result-producing branches (RPB0, RPB1, and RPB2). It also shows that RPB0 contains a reference to one-argument automatically defined function ADF3 (the number of arguments being shown inside the braces) and that ADF3 invokes one-argument automatically defined function ADF2. Similarly, the figure shows that RPB1 contains a reference to one-argument automatically defined function ADF3 and that ADF3 invokes two-argument automatically defined function ADF2. The figure also shows that RPB2 contains references to one-argument automatically defined functions ADF4 and ADF2 and that ADF4, in turn, invokes one-argument automatically defined function ADF2.

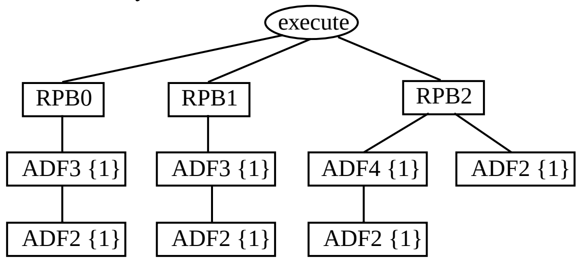

Figure 13 Call tree for a genetically evolved crossover (woofer-tweeter) filter.

2.1.4 Passing Parameters to Substructures

If passing parameters to a substructure is likely to be useful in solving the problem, genetic programming may be appropriate.

Continuing the discussion of the crossover filter from the previous section, consider the subcircuit produced by the three-ported automatically defined function ADF3 (figure 14) that appears in the genetically evolved crossover (woofer-tweeter) filter. In addition to the mundane matter of inserting an ordinary 5,130-nanofarad capacitor C112, the execution of automatically defined function ADF3 has the following two noteworthy consequences:

- ADF3 inserts a *parameterized capacitor* C39 whose component value is dependent on the dummy variable ARG0 that is passed into automatically defined function ADF3 from the program that calls ADF3 (namely the result-producing branch RPB0 shown in figure 13).
- ADF3 calls automatically defined function ADF2. A dummy variable ARG0 is passed along from ADF3 to ADF2. In turn, ADF2 creates a

parameterized inductor whose component value is dependent on this dummy variable.

Once a dummy variable (formal parameter) is passed from one part of a genetically evolved hierarchy into another, different instantiations of the

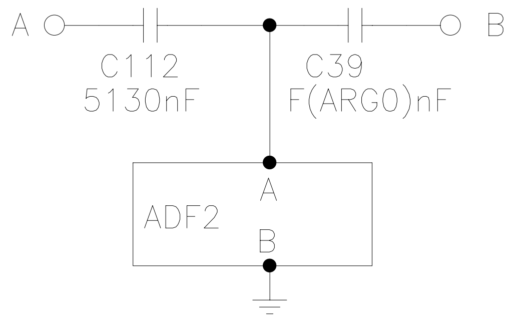

Figure 14 Subcircuit produced by the execution of three-ported automatically defined function `ADF3` of the genetically evolved crossover (woofer-tweeter) filter.

2.1.5 Number of Arguments Possessed by Substructures

If discovering the dimensionality of a substructure (that is, the number of arguments possessed by the substructure) is an important part of the problem, genetic programming may be appropriate.

The architecture-altering operations of argument duplication, argument creation, and argument deletion can automatically and dynamically determine, during a run of genetic programming, the number of arguments possessed by a substructure (Koza, Bennett, Andre, and Keane 1999).

3 The Developmental Process

If a developmental process is a desirable way to represent a solution to the problem, genetic programming may be appropriate.

The developmental approach has several advantages in automatically synthesizing complex structures such as circuits.

For one thing, the developmental approach has the specific advantage of preserving locality. Because most of the component-creating, topology-modifying, and development-controlling functions operate on a small local area of the circuit, the subtrees that are transplanted by the crossover operation generally operate locally. Thus, when a crossover replaces a subtree in one individual with a subtree from another individual, it (usually) replaces a local structure in the circuit created by the first individual with a local structure in the circuit created by the second individual. The developmental process works in conjunction with the crossover operation in preserving locality.

The developmental process has the additional advantage of preserving electrical connectivity. There are no unconnected leads in the initial circuit. Each component-creating, topology-modifying, and development-controlling function preserves connectivity at each stage of the developmental process. The result is that there are no unconnected leads in the fully developed circuit.

Also, the developmental approach enables useful parts of a circuit-constructing program tree to be reused (as already illustrated by figures 7, 8, and 9 and tables 1 and 2). Reuse eliminates the need to “reinvent the wheel” on each occasion when a particular sequence of steps may be useful. Reuse makes it possible to exploit a problem’s modularities, symmetries, and regularities (and thereby potentially accelerate the problem-solving process).

The preservation of syntactic validity and executablity of the circuit-constructing program trees during all genetic operations, the preservation of the constrained syntactic structure during all genetic operations, the preservation of electrical connectivity, the preservation of locality during crossover, and the facilitation of reuse together contribute to the efficiency by which developmental genetic programming is able to synthesize circuits.

In contrast, the ordinary crossover operation in the genetic algorithm operating on fixed-length character strings rarely preserves syntactic validity when chromosome strings are used to represent complex structures (e.g., electrical circuits).

When syntactic validity is not preserved, the user is usually faced with three choices concerning the syntactically invalid offspring:

- deletion,
- penalization, or
- repair.

For complex structures such as circuits, deletion is rarely a practical option because the percentage of syntactically valid offspring after a crossover is so low that virtually every individual would be deleted. Penalization is rarely practical for the same reason. Thus, in practice, repair is the most viable option.

The ordinary crossover operation used in the genetic algorithm has the specific advantage that all genetic material in the offspring comes from one or the other parent. This is usually not the case when repair is undertaken. That is, the offspring’s genetic material does not come exclusively from its parents.

The parents come from the current generation of the run. As such, the parents are likely to be relatively fit individuals, and their genetic material is likely to contribute to the solution of the problem.

The repair process does not confer this benefit on the resultant offspring.

As an example of a crossover operation whose offspring requires repair, consider the circuits defined by the chromosomes in tables 4 and 5.

The chromosome of the first parent (table 4) codes for the single T-section depicted in figure 15. This T-section contains two inductors (L1 and L2) and one capacitor (C1). This individual is functionally equivalent to the circuit of figure 1 and therefore has the same fitness and the same frequency-domain behavior as shown in figure 2. As previously mentioned, a single T-section acts as a (very poor) lowpass filter; however, in an early generation of the run, a single T-section may be among the best individuals in the population at the time.

The chromosome of the second parent (table 5) codes for the single T-section depicted in figure 16. This T-section contains two inductors (L3 and L4) and one capacitor (C2). This individual is functionally equivalent to the first parent.

Now consider a crossover operation whose offspring requires repair. In particular, consider a crossover point at the right of the 12th gene. That is, genes 1 through 12 come from the chromosome in table 4 and genes 13 through 24 come from the chromosome in table 5.

Table 6 shows the chromosome of the offspring resulting from this crossover.

Figure 17 shows the offspring. The offspring circuit contains three inductors and two capacitors. Components L6 and C3 originate from the first parent (where they were called L1 and C1, respectively, in figure 15). Components L5, L7, and C4 originate from the second parent (where they were called L3, L4, and C2, respectively, in figure 16). After the crossover, one end (node 9) of the inductor L7 is left dangling. Similarly, one end (node 8) of the capacitor C4 is also dangling. Moreover, the offspring has no connection to the output probe point VOUT. Because the offspring is syntactically invalid, a mechanism must now be invoked to repair it.

There are numerous methods in the literature for repairing a syntactically invalid offspring. Some overtly employ random steps. The ones that do not employ random steps typically employ steps that are so arbitrary that they might as well be random. The point is not that these repair mechanisms fail to create a syntactically valid offspring, but that they succeed in a manner that resembles mutation.

When the genetic algorithm operating on fixed-length character strings is being used for problems involving complex structures, the probability is exceedingly small that a syntactically valid chromosome will result from a crossover that is applied directly to the structure. Thus, virtually every crossover requires repair. To the extent that the repair mechanism is either random or arbitrary, virtually every crossover becomes mutational in character.

In summary, the benefits of the developmental process include

- reuse,
- preservation of locality,
- preservation of electrical validity (for circuits),
- preservation of syntactic validity and executability (thereby alleviating the need for repair and the consequent introduction of genetic material from sources other than the two parents from the current generation of the run).

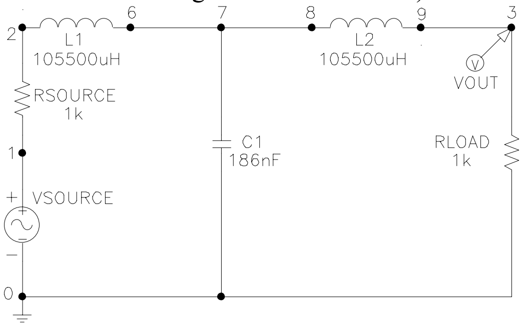

Figure 15 First parent

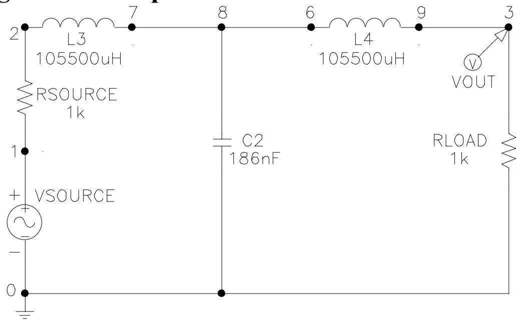

Figure 16 Second parent

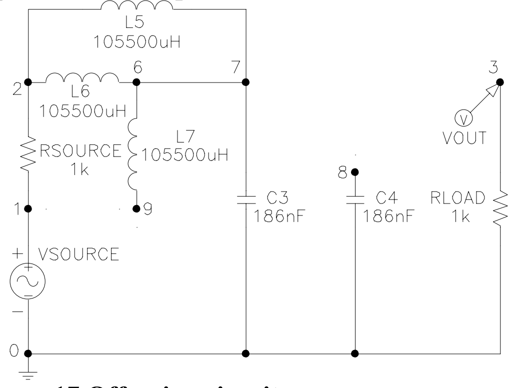

Figure 17 Offspring circuit

4 Conclusion

This paper has described the ways by which genetic programming can facilitate reuse and the associated advantages of using developmental processes.

References

Drechsler, Rolf. 1998. *Evolutionary Algorithms for VLSI CAD*. Boston: Kluwer Academic Publishers.

Grimbleby, J. B. 1995. Automatic analogue network synthesis using genetic algorithms. *Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA)*. London: Institution of Electrical Engineers. Pages 53–58.

Higuchi, Tetsuya, Iwata, Masaya, and Lui, Weixin (editors). 1997. *Evolvable Systems: From Biology to Hardware: First International Conference, ICES-96, Tsukuba, Japan, October 1996 Proceedings*. Lecture Notes in Computer Science, Volume 1259. Berlin: Springer-Verlag.

Higuchi, Tetsuya, Niwa, Tatsuya, Tanaka, Toshio, Iba, Hitoshi, de Garis, Hugo, and Furuya, Tatsumi. 1993. Evolving hardware with genetic learning: A first step towards building a Darwin machine. 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.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press. Second edition. Cambridge, MA: The MIT Press 1992.

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

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

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Keane, Martin A., Streeter, Matthew J., Mydlowec, William, Yu, Jessen, and Lanza, Guido. 2003. *Genetic Programming IV. Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996a. 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 Academic Publishers. Pages 151–170.

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996b. Reuse, parameterized reuse, and hierarchical reuse of substructures in evolving electrical circuits using genetic programming. In Higuchi, Tetsuya, Iwata, Masaya, and Liu, Weixin (editors). 1997. *Proceedings of International Conference on Evolvable Systems: From Biology to Hardware (ICES-96)*. Lecture Notes in Computer Science, Volume 1259. Berlin: Springer-Verlag. Berlin: Springer-Verlag. Pages 312–326.

Kruiskamp, Marinum Wilhelmus. 1996. *Analog Design Automation using Genetic Algorithms and Polytopes*. Eindhoven, The Netherlands: Data Library Technische Universiteit Eindhoven.

Mazumder, Pinaki and Rudnick, Elizabeth M. (editors). 1999. *Genetic Algorithms for VLSI Design, Layout and Test Automation*. Upper Saddle River, NJ: Prentice Hall.

Rechenberg, Ingo. 1973. *Evolutionsstrategie*: *Optimierung Technischer Systeme nach Prinzipien der Biolgischen Evolution*. Stuttgart-Bad Cannstatt: Verlag Frommann-Holzboog.

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

Smith, Steven F. 1980. *A Learning System Based on Genetic Adaptive Algorithms*. Ph.D. dissertation. Pittsburgh, PA: University of Pittsburgh.

Stoica, Adrian, Keymeulen, Didier, and Lohn, Jason (editors). 1999. *Proceedings of the First NASA/DOD Workshop on Evolvable Hardware, Pasadena, California, July 19–21, 1999*. Los Alamitos, CA. IEEE Computer Society.

Stoica, Adrian, Zebulum, Ricardo, and Keymeulen, Didier. 2001. Polymorphic electronics. In Liu, Yong, Tanaka, Kiyoshi, Iwata, Masaya, Higuchi, Tetsuya, and Yasunaga, Moritoshi (editors). *Evolvable Systems: From Biology to Hardware, 4th International Conference, ICES 2001, Tokyo, Japan, October 2001 Proceedings.* Lecture Notes in Computer Science, Volume 2210. Berlin: Springer-Verlag. Pages 291–302.

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. Pages 444–452.

Zebulum, Ricardo Salem, Pacheco, Marco Aurelio C., and Vellasco, Marley Maria B. R. 2002. *Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms*. Boca Raton, FL: CRC Press.