BEHAVIOURAL SYNTHESIS OF ANALOGUE INTEGRATED CIRCUITS

by

SIMON JAMES PARISH

DOCTOR OF PHILOSOPHY

School of Electronic, Electrical and Computer Engineering
The University of Birmingham
B15 2TT
England

January 14, 2010.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION & MOTIVATION

The modern world is digital. The modern system-on-chip (SoC) application specific integrated circuit (ASIC) has permeated almost every aspect of life. The internet is ubiquitous, and the consumer is now able to choose from a plethora of digital gadgetry: iPods and personal digital music players, digital cameras, satellite navigation systems and portable video game systems are just a few. And, of course, mobile phones continue to merge all of these devices, and more, into one device. Home cinema systems, high definition digital televisions and digital BluRay players are just some of the other digital innovations that consumers now enjoy. Even more every day items like microwaves, washing machines and even cars now rely heavily on digital technology. The modern world is indeed digital.

With the exception, of course, that it is not. Not by any means. It is certainly true that many applications that were once analogue have now been replaced by more efficient, better, faster, smaller and cheaper digital alternatives and it is safe to say that this has brought about a digital revolution of sorts, but analogue systems and circuitry will *always* be needed. The reason for this is simple: the physical world is analogue. Digital systems must interface to the analogue world, and at its root, everything is analogue in nature. Even digital circuitry itself is in reality analogue circuitry designed to operate in only one of two allowable states. Digital is a simplified approximation of analogue. Without analogue circuits such as phase locked loops (PLLs), digital-to-analogue and analogue-to-digital converters (DACs/ADCs), filters, level shifters, amplifiers and voltage and power regulators, the digital world could not exist.

1.1 The Analogue Design Problem

The first electronic circuits were analogue. Although the concept of digital arithmetic may be traced back at least to the early 1800s[1], it was analogue circuits that dominated electronic system design during the early 20th century. Since then, approaches to analogue design have changed little, and modern analogue design is still a predominantly manual task that requires a great deal of skill and experience for all the but the most trivial of circuits, just as it did back

[1] This is the time when George Boole invented *Boolean Algebra* and Charles Babbage first conceived of the idea of the programmable computer.

then.

1.1.1 The 'Dark Arts'

Analogue engineers are far fewer in number than digital engineers, probably because analogue design is a talent which is far harder to acquire than digital design. It takes many years of experience to really become a proficient and competent analogue designer. Many tend to specialise in specific types of circuits, such as PLLs or filters. Indeed, analogue design is often perceived as something of a 'dark art', as something that is unfathomable to the uninitiated.

Entire careers can be spent becoming an expert in a given class of analogue circuit. Amplifiers are a good example. Over the decades, many different topologies have been developed for a wide variety of different applications. They make use of different techniques and strategies, designed to operate at extremely high frequencies, or dissipate very little power. Others may be designed to introduce very little distortion at audio frequencies. Filters are similar in this regard, and a good designer will gain an intuitive feel for how to tweak the circuits in order to optimise particular characteristics.

Analogue design is certainly a difficult task for the inexperienced, and in some ways it is indeed more like an art. Not only is experience important, but it also relies on intuition and to a certain extent, creativity. Unfortunately this makes it a very difficult process to automate, and is one of the reasons that it is still a predominantly manual process.

1.1.2 Design Automation

The technological domination of the modern world has been made possible in part by the emergence of design automation. Modern digital designs are huge, consisting of tens or hundreds of millions of transistors. The complexity of these chips are far beyond the capacity of any one individual to fully and exhaustively comprehend, the only way they can be designed in reasonable time frames is with the use tools which automate many parts of the design process. The stages involved in the physical design of an integrated circuit are usually referred to collectively as a *design flow*. The output of one stage forms the input of another.

As the level and sophistication of digital design automation increased, analogue design automation has almost stagnated. The gap between the two has now widened to a great extent, and only serves to highlight how under developed analogue design automation really is. The result is that digital circuits of a given level of functional complexity will typically take far less time to develop than analogue circuits of a similar complexity.

However, as technology has developed, there have certainly been *some* advances in analogue design techniques. New types of circuits have been developed, and the advent of the digital computer has made analogue circuit simulation a reality. This is probably the single most significant development in the field of analogue design automation to date. The first analogue simulator was CANCER[2], and although modern simulators, types of analysis available and the device models used have all greatly increased in sophistication[3] the fact remains that simulation, is still the only significant type of analogue design automation tool available, with the possible exception of optimisation.

Being able to predict the characteristics and behaviour of a given circuit topology is extremely useful and is no doubt a vital part of the modern analogue design process, which is generally an iterative procedure. However, one of the most significant differences between analogue and digital design automation is the ability to effectively do the opposite; that is, to take a required circuit behaviour and from that generate a suitable circuit topology. This process is circuit *synthesis*. Figure 1.1 illustrates this.

1.1.3 The Digital/Analogue Split

Analogue circuits and sub-systems are crucial parts of modern Systems-on-Chip (SoCs), whether mixed signal or ‘purely’ digital. Even digital SoCs will have *some* analogue circuitry, even if it is only to do with clock signal generation or power regulation/control. Purely analogue SoCs are also quite rare.

Mixed signal SoCs, however, are becoming increasingly common as more and more functions

[2] CANCER was developed in the 1960s; SPICE1 was in fact a derivative of this.

[3] Much of this could not have been possible without the increasing computational power of the digital computer.

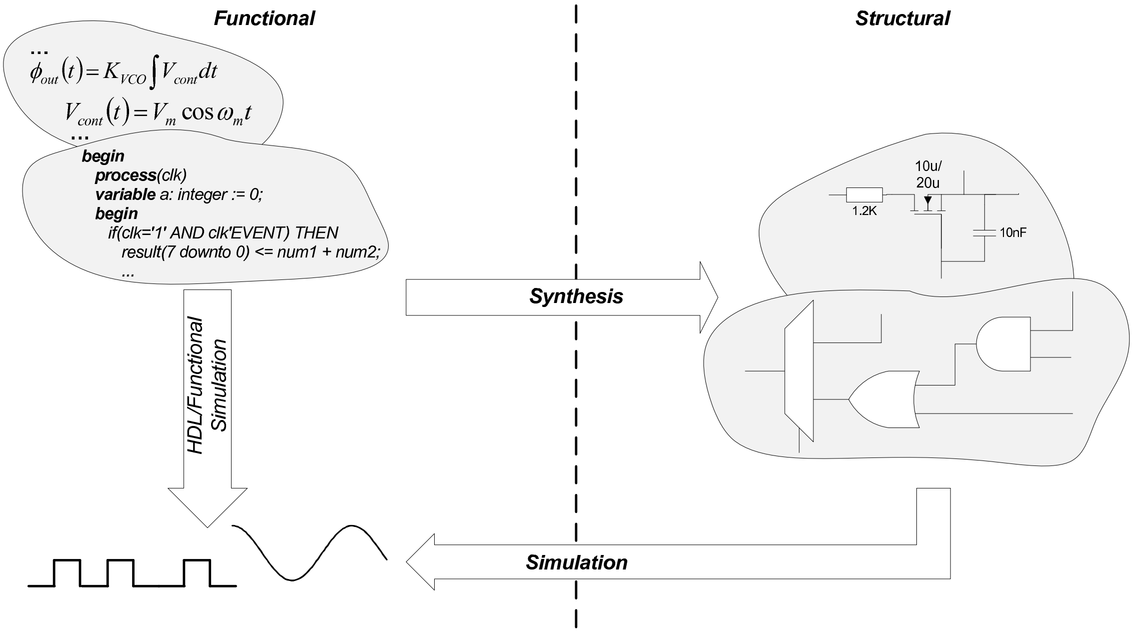

Functional
Structural
...
$\phi_{out}(t) = K_{VCO}\int V_{cont}dt$
$V_{cont}(t) = V_m \cos\omega_m t$
...
begin
process(clk)
variable a: integer := 0;
begin
if(clk='1' AND clk'EVENT) THEN
result(7 downto 0) <= num1 + num2;
...
HDL/Functional Simulation
Synthesis
10u/
20u
1.2K
10nF
Simulation

Figure 1.1: Relationship between synthesis & simulation

are integrated into modern Application Specific Integrated Circuits (ASICs). However, even SoCs which derive a significant part of their functionality from analogue circuits have a relatively small chip area dedicated to them. Despite this, the design time and effort of the analogue portion of the design is significantly higher than the digital portion [46].

This split is due to two reasons. The first is concerned with required chip area. In terms of number of components, analogue circuits are far, far smaller than digital. This is because analogue circuits tend to exploit the full range of physical properties of the transistors and other components used in them, whereas digital circuits are made up of huge numbers of transistors, with each one acting as a very simple switch. Therefore, a small number of devices can result in very complex behaviour in an analogue circuit, whereas a digital circuit will require many more devices to exhibit a similar level of functional complexity.

The second reason, to do with design time and effort, is that digital design automation is in a considerably more advanced state than analogue design automation. Analogue designs will usually absorb many more man-hours of design time than a comparable digital design.

1.1.4 Circuit Topology, Sizing & Layout

In the digital design flow, the function of the circuit is usually expressed in a highly abstracted form, and a hardware description language (HDL) is most often used as the medium. There are two clearly identifiable steps in converting the abstracted functional definition into a circuit layout ready to be manufactured. It is first synthesised in order to produce a *netlist*, and then that netlist is *laid out*; it goes through a *place and route* process.

In contrast, none of these steps exist for analogue in an automated capacity. There is, however, a possible exception to this in the form of circuit optimisation. Analogue circuits must be *sized*, the components must be given suitable values and any transistors must be given suitable dimensions. Optimisation tools do exist for analogue circuits. They usually require an initial guess of the component sizes to be provided. A search is then performed to find values that result in a closer fit to the specified circuit characteristics or behaviours. There are many such characteristics that may be optimised. This may include *design centering*, where components are sized such that the performance of the analogue circuit is *centered* in an allowable range, reducing the likelihood that variations in component value will push the circuit out of specification.

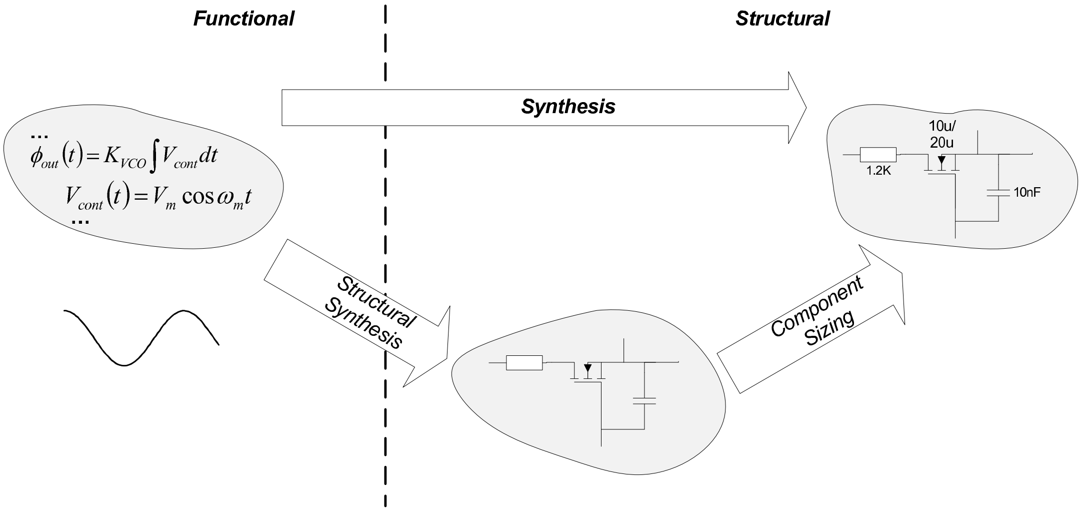


Figure 1.2: Component sizing can be considered to be part of the synthesis process for analogue circuits

There is no real digital equivalent to circuit sizing primarily because digital synthesis produces a netlist of predesigned *cells*. The component values in an analogue circuit have a direct impact on its functionality. An unsized netlist, which contains no component values would therefore be considered to be incomplete. In that sense it would seem natural to include circuit sizing as a part of the synthesis process, as shown in figure 1.2.

The only automated tools exist that assist in some aspect of the *design* of analogue circuits, then, are optimisation tools of various kinds. While useful, this leaves out the majority of the design process which must still be done manually. Simulation, while vital, assists with *verification* of a circuit rather than directly with its design.

1.2 The Analogue Synthesis Problem

No commercially available mature analogue circuit synthesis tools currently exist, despite digital circuit synthesis tools being ubiquitous in the modern design flow. Clearly, such a tool would be extremely useful, so there must be a good reason *why* this is so.

There have been a number of attempts at tackling this problem, with a range of different strategies being employed. However, to date the problem of analogue circuit synthesis has resisted all attempts to develop a satisfactory solution. The absence of any real form of synthesis from the modern analogue design flow and the corresponding lack of a commercial supply of such tools is clear evidence of this. Because synthesis is highly desirable from an engineering view point, there would be a significant market for a usable, reliable and robust analogue circuit synthesis tool. Analogue synthesis is difficult. *Extremely* difficult.

The existence of digital synthesis plays a role in creating the desire for an analogue counterpart. The success of digital design automation tools make analogue design automation conspicuous by its absence. The problems faced by the two tasks differ considerably. Digital synthesis tools use predefined rules to break down the specified digital behaviour into a collection of primitive digital operations which can be mapped directly onto digital cells [18]. These cells are sub-circuits which have a precisely defined functionality. The process of digital synthesis produces a topology of interconnected cells which is logically identical to the specified circuit function.

This approach is simply not applicable to analogue circuits. While digital behaviour is usually specified with the use a suitable HDL, there is no *commonly used* equivalent in the analogue world (although such HDLs do exist [10]). Even if there were, such complex behaviour cannot easily be broken down into simpler operations or behaviours. Analogue sub-circuits can never have such precisely defined behaviour as digital cells do, and there is no guarantee that any given collection of analogue sub-circuits would be able to produce the required functionality of the original specification. This means that any *truly* universal analogue synthesis tool would need to build a netlist not out of cells or sub-circuits, but out of transistors and other primitive components. This is another key difference between analogue and digital synthesis.

1.2.1 Previous Research

Due to the lack of any obvious best or systematic way to perform analogue synthesis, a variety of approaches have been employed in experimental synthesis systems. Some methods attempt to mimic the digital synthesis process as far as possible, some methods are search based and others try to mimic human behaviour when designing a circuit. The following summary is by no means exhaustive. All that have been tried, however, share at best limited success.

Search algorithms are the most common approach. This usually involves the repetition of a 'generate-and-test' sequence. Trial circuits are generated, measured in some way to determine how suitable a solution it is, and then those results are used to guide the search from that point. This is of course in sharp contrast to the inner workings of a digital synthesis tool.

Two popular generate-and-test search algorithms that have been applied to this problem are simulated annealing (SA) [25] and genetic algorithms (GA) [20]. These are two very different approaches to searching through a 'solution space' of possible circuits, although the overall structure of both algorithms is the same. Both typically require a great number of iterations and both will trial a great number of unsuitable circuits. The difference lies in how the search is guided, or in other words, how a new solution point is selected for evaluation and how the results of each circuit measurement influence this. There is a strong element of probabilistic decision making in both.

HDL-based approaches try to mimic digital synthesis tools to an extent, and accept analogue behavioural specification in the form of an analogue HDL. This HDL source code is then parsed. Structure is derived from the resulting parse tree and a high level topology of interconnected analogue cells is formed. These analogue cells are then mapped onto predefined transistor level netlists, and various optimisation algorithms are then applied to size these subcircuits. There is usually an optimisation step required in order to size the analogue cells. Also, only a small subset of the HDL used is usually accepted as synthesisable.

Artificial intelligence based approaches have also been applied to this problem in the form of expert systems. The idea behind this is to try to mimic the mental process a human analogue circuit designer uses and try to attack the problem hierarchically. A knowledge base of predefined sub-circuit topologies is used by these systems, which mimics the analogue knowledge an experienced designer would have.

Almost all of the experimental solutions, however, are in various ways really quite restricted, as discussed in section 3.2.2. They tend to have the ability to deal with a small number of topologies or behaviours. Most are either inherently limited, or are bound by their implementation and do not allow complete freedom in topology generation.

1.3 Research Overview

As presented in Chapter 3, the majority of previous research has been rather limited in scope. With a few notable exceptions [30][50][36], none of it has investigated the generation of topologies with complete freedom. Most research has focused on synthesis methods which require the circuit topology to be provided, either implicitly or explicitly (see section 3.2). Essentially, they just perform circuit sizing and there is little real difference between these approaches and optimisation. Other methods have the ability to perform limited topology synthesis, but only at a high level. They arrange interconnections of subcircuits with predefined transistor-level topologies. They typically also size these subcircuits, but they are still inherently limited by the range and quality of these predefined subtopologies. Section 3.2.2 covers these methods in more detail.

The systems which genuinely allow significant freedom in topology generation are those based on fairly complex implementations of genetic algorithms, many of which are based on *genetic programming* (GP) [30] [50]. Section 3.2.2 presents examples of GP and GA-based synthesis systems. GP allows the generation of a circuit of unrestricted size, within limits, which is something that more traditional genetic algorithms do not allow. Other, non-GP GA implementations [36] also allow this level of freedom. Therefore, the research previously conducted would suggest that only certain GA implementations have the potential to fill the topology-generation gap in the analogue design flow illustrated in figure 1.2. Chapter 4 presents GAs and GP in detail.

1.3.1 Research Objectives

Despite a number of experimental analogue synthesis systems being developed, as already discussed, commercial tools are still not a reality. Analogue synthesis is still not a real, commonly used and accepted part of the modern design flow and analogue design remains a manually intensive task. Clearly, all previous experimental systems have been in some way unsatisfactory. It seems reasonable that any genuinely useful synthesis system would need satisfy the following criteria:

Automation. There is a minimum level of automation required in order to reduce a significant amount of human design effort. Identifiable levels of analogue design automation are rather coarse, and beyond the 'finishing off' step of circuit sizing (optimisation) which can already be automated, the most significant remaining task is topology generation, as shown in figure 1.2. Any useful synthesis system must automate topology generation, and ideally it must also perform circuit sizing.

Robustness. Digital synthesis tools reliably produce a netlist of cells. In general, they only fail to do this when fed with badly written HDL source code. Provided that a good quality behavioural specification is input, a corresponding netlist is produced. The digital design flow would become much more difficult to use if this were not the case. Analogue synthesis tools must achieve a similar level of reliability, they must be *robust*. Much of the previous research has produced experimental tools which are not guaranteed to produce any useful circuit at all. Not only can two separate runs using the same inputs produce quite different circuits, but they may produce no circuit at all.

Scope/Generality. Any tool which is restricted to only a very narrow range of circuits would obviously be of limited use. Digital synthesis tools can cope with virtually any required functionality. A useful analogue synthesis tool must be able to cope with as wide a range of circuit behaviours as possible.

Execution time. One of the primary goals of automation is reduce the required time to perform a particular task. While this is not the only benefit of automation, it is an important one. A synthesis tool would ultimately be of little practical use if it took significantly longer to produce a given circuit than a human designer.

Given the above criteria, the objectives of this research are, having implemented a genetic algorithm which achieves a high level of automation and generality, to:

- Investigate the robustness of a genetic algorithm in terms of its sensitivity to its control parameters and its corresponding ability to reliably generate a useful circuit.
- Identify any obvious weaknesses or issues in genetic algorithms which may prevent them from satisfying the above criteria.
- Determine whether genetic algorithms have the potential to form the basis of a *realistically usable* design tool.

In particular, a realistically usable design tool would be able to synthesise analogue circuits which may be found on a typical modern analogue SoC. Active and passive filters and amplifiers, operating at frequencies of a few KHz up to hundreds of MHz. Such a system would ideally also be able to synthesise more complex circuits such as PLLs, and the analogue portions of digital-to-analogue, and analogue-to-digital converters.

1.4 Thesis Contributions

1. An investigation into how *practically useful* Genetic Algorithms are for analogue circuit design. Some important characteristics which have a direct impact on this, of both analogue

circuits and Genetic Algorithms, are identified.

2. Developed a Genetic Algorithm implementation, using a tree encoding method similar to that used by Genetic Programming, specifically tailored for analogue synthesis. Unlike Genetic Programming, this implementation does not involve the overhead of parsing and executing LISP programs.

3. Developed a Genetic Algorithm implementation, specifically tailored for analogue synthesis, which can accept an arbitrary amount of predefined knowledge and can process arbitrarily constrained problems.

4. Developed a novel Genetic Algorithm fitness function based on pole-zero analysis.

5. An investigation into the mutability of both the topology and sizing of analogue circuits.

6. Developed a set of tools for analysing the applicability of SPICE in the role of circuit evaluator for Genetic Algorithm fitness functions.

1.5 Thesis Structure

- **Chapter 2 - Electronic Design Automation: A Brief Introduction.** This chapter provides an introduction to Electronic Design Automation (EDA), and includes a history of how it developed and explains why the state of the art is as it is. Digital and analogue EDA are compared and significant differences are highlighted.

- **Chapter 3 - Approaches to Analogue Synthesis: Attempts At Solving The Problem.** A look at previous attempts at creating an analogue synthesis tool. A comparison of the wide variety of approaches which have been employed in an attempt to tackle this difficult problem is presented. Includes a short survey of the literature.

- **Chapter 4 - Genetic Algorithms.** This chapter examines some of the finer details of Genetic Algorithms, and also takes a look at Genetic Programming.

- **Chapter 5 - SPICE Simulation & Other GA Issues.** SPICE is a well known simulation tool, something of an ‘industry standard’. It is also a very common element of many Genetic Algorithms which have been designed for analogue circuit synthesis. It is capable of producing highly accurate circuit simulations, but there are many hurdles which may prevent this. This chapter examines some of these hurdles, and what effect they may have on Genetic Algorithms. Issues related to circuit encoding are also considered.

- **Chapter 6 - A Genetic Algorithm System For Analogue Synthesis.** A detailed presentation of an analogue synthesis system based on Genetic Algorithms. A novel fitness function based on pole-zero analysis is also presented.

- **Chapter 7 - Case Studies.** This chapter contains the results of a series of experiments, including the synthesis of some analogue filters, the effect of varying control parameters of the Genetic Algorithm. Finally, an investigation is presented into the sensitivity of analogue circuits to changes in their sizing and topology.

- **Chapter 8 - Conclusions.** The central conclusions of this thesis are presented, and interesting directions of possible future work are discussed.

Chapter 2

ELECTRONIC DESIGN AUTOMATION: A BRIEF INTRODUCTION

Since the first integrated circuit (IC) was produced in 1958, development of ICs has come a long way. Their complexity, as measured by the number of transistors placed on a single die has increased by a factor of a million over the last forty years and is still increasing. From the early Intel 4004 containing just 2,300 transistors [1] to the Intel Itanium Tukwila containing a staggering 2 billion [2], the challenges associated with IC design have changed beyond all recognition. It simply would not be possible to produce modern ICs without heavily automating the design process, and as a result of the ever increasing difficulties associated with ASIC design the field of Electronic Design Automation (EDA) came into being.

This chapter discusses what EDA is, how and why it developed and examines the current state of the art. While this text is focused on high level analogue circuit synthesis, it is also useful to briefly examine digital EDA to serve as a reference point. Indeed, the question of analogue synthesis exists precisely because such a process exists for digital systems and has proved to be indispensable.

2.1 Introduction to EDA

Electronic Design Automation tools are anything that assists with the design specification, simulation, verification or realisation of integrated circuits, field programmable gate arrays (FPGAs), printed circuit boards or electronic systems. A modern design flow for electronic systems consists of a series of EDA stages. Such a flow takes a human-defined specification of some kind and ultimately produces a circuit representation that is suitable for direct implementation or manufacture.

A brief summary of design automation, focused mainly on tools for ICs is presented here. Not only has the development of ICs provided the main driving force in the development of EDA, this is also the area that EDA is most needed. The summary is divided into four parts. Section 2.1.1 explains the role of circuit abstraction in EDA, section 2.1.2 defines more concisely what EDA is, and gives some examples of EDA tools and the roles they perform. Sections 2.1.3 and 2.1.4 describe typical modern design methods and procedures for analogue and digital circuits, to serve as a comparison between the two.

2.1.1 Circuit Abstraction

Circuit abstraction is an important part of EDA. It allows circuit designs to be captured and represented in forms suitable for use at various stages in the design process. This includes the initial, human-input point of design capture, and also intermediate design representations at various stages in the design process.

Human designers can only cope with so much design complexity, and so when dealing with all but the most trivial of circuits it is necessary to abstract the design in some way. The amount of detail in a given level of abstraction is roughly inversely proportional to the size of the design, or portion of design being considered. The highest levels of abstraction are typically used for giving an overview at the top level of a design, maybe when describing the functionality of a complete SoC or microprocessor. Minute detail is not needed here and would be indigestible on such a large scale. Very low levels of abstraction, such as describing individual transistors is useful when designing a standard cell such as a logic gate or a flip-flop.

However, it is not only *levels* of abstraction that are of interest. Ashenden [11] also identifies three *domains* of abstraction - *functional*, *structural* and *geometric* (physical). Each domain has its own levels of abstraction. Abstraction in the functional domain describes the function and operations performed by the system. The structural domain is concerned with how the system is partitioned and connected between its partitions. The geometric domain describes how the circuit is implemented and physically laid out.

Figure 2.1 shows levels of abstraction used in a typical design flow. A behavioural system model usually specifies *what* is to be done, but not *how* it is to be done. Algorithms may often be used in a behavioural model. Register Transfer Level (RTL) modeling describes the system in terms of data storage elements, or registers. Data is transferred between registers through sections of digital logic which transform the data in some way. The next two lower levels of abstraction are in the structural domain. Gate/macro abstraction describes the design in terms of a list of sub-circuits or cells and their interconnections. The transistor level is similar, except that individual components are listed.

Despite the fact analogue and digital circuits often require different modeling techniques,

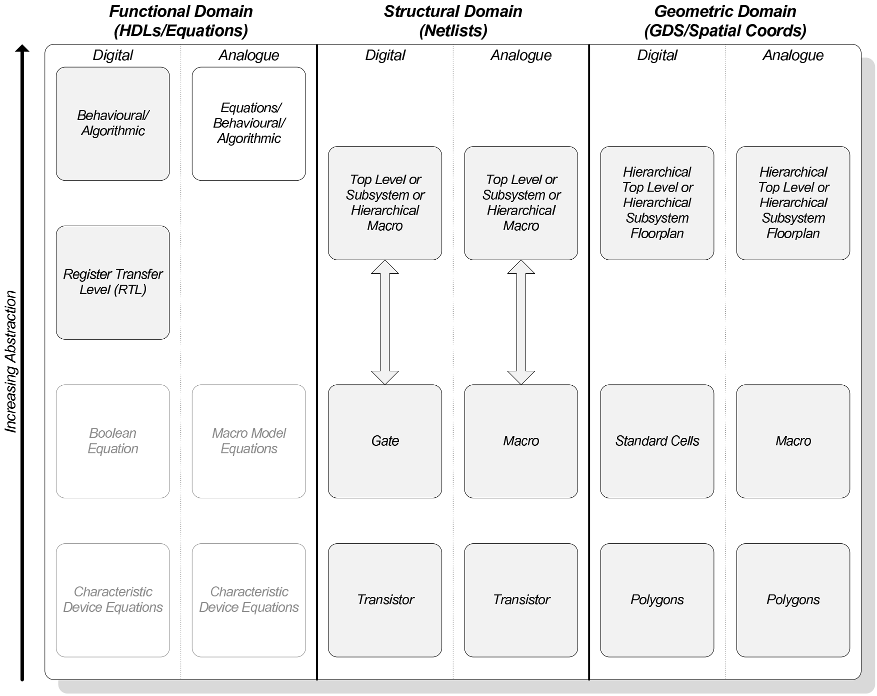

Functional Domain (HDLs/Equations)
Structural Domain (Netlists)
Geometric Domain (GDS/Spatial Coords)
Digital
Analogue
Digital
Analogue
Digital
Analogue
Increasing Abstraction
Behavioural/ Algorithmic
Equations/ Behavioural/ Algorithmic
Top Level or Subsystem or Hierarchical Macro
Top Level or Subsystem or Hierarchical Macro
Hierarchical Top Level or Hierarchical Subsystem Floorplan
Hierarchical Top Level or Hierarchical Subsystem Floorplan
Register Transfer Level (RTL)
Boolean Equation
Macro Model Equations
Gate
Macro
Standard Cells
Macro
Characteristic Device Equations
Characteristic Device Equations
Transistor
Transistor
Polygons
Polygons

Figure 2.1: Levels of Circuit Abstraction

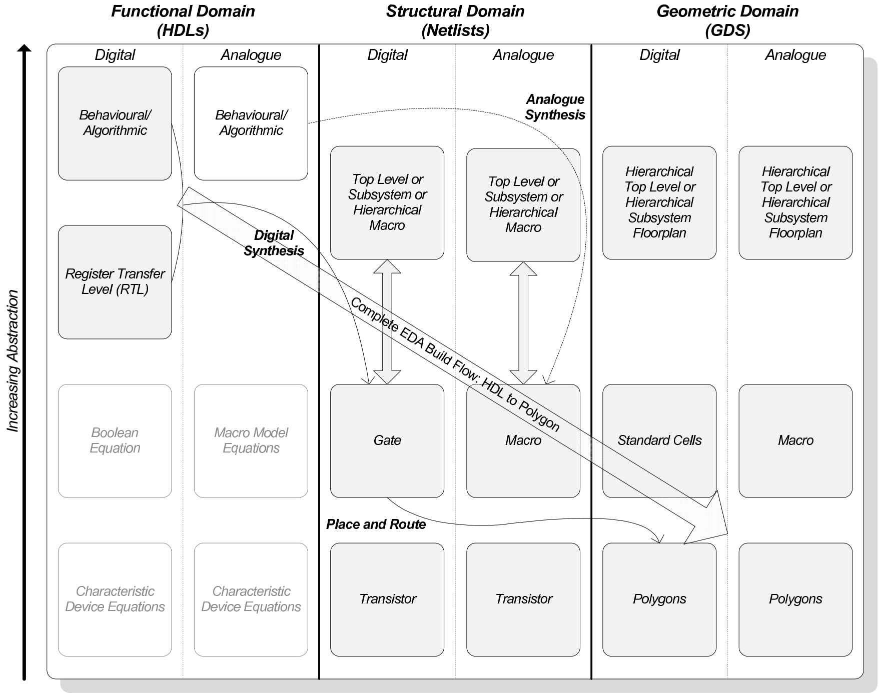
different abstraction levels are generally applicable to both, and will look very similar. Even a behavioural abstraction that might describe circuit function in terms of an algorithm, will allow both analogue and digital circuits to be modeled. It is important to note that there is no analogue equivalent of digital RTL abstraction. This has important implications for analogue EDA, as RTL bridges a wide gap between behavioural and gate level abstractions for digital systems.

2.1.2 EDA Concepts

Most EDA tools may be considered as belonging to one of two groups: some which perform *verification* of a circuit design, and others which perform *translation* of a circuit design.

The need for translation results from the ability, and often the necessity, to capture designs at a high level of abstraction. As discussed in section 2.1.1, highly abstracted design representations deliberately lack fine detail. This allows human designers to cope with bigger designs. However a circuit must ultimately be manufactured or implemented in some way - that is, it must be *realised*. The design must be translated to a level of abstraction suitable for direct implementation.

There are two important translation stages used in the design of modern ICs and FPGAs. *Synthesis* is the process of translating from the highest levels of abstraction to the macro, gate, or netlist level. An important feature of synthesis is that it not only translates to a lower abstraction level, it also translates between *abstraction domains*. The input is a high level functional description. The output is a lower level *structural* description. While this contains much more detail about how the circuit will be finally realised, it still cannot be directly implemented as it contains no physical or spatial information.



Functional Domain (HDLs)
Structural Domain (Netlists)
Geometric Domain (GDS)
Digital
Analogue
Digital
Analogue
Digital
Analogue
Increasing Abstraction
Behavioural/ Algorithmic
Behavioural/ Algorithmic
Analogue Synthesis
Top Level or Subsystem or Hierarchical Macro
Top Level or Subsystem or Hierarchical Macro
Hierarchical Top Level or Hierarchical Subsystem Floorplan
Hierarchical Top Level or Hierarchical Subsystem Floorplan
Register Transfer Level (RTL)
Digital Synthesis
Complete EDA Build Flow: HDL to Polygon
Boolean Equation
Macro Model Equations
Gate
Macro
Standard Cells
Macro
Place and Route
Characteristic Device Equations
Characteristic Device Equations
Transistor
Transistor
Polygons
Polygons

Figure 2.2: EDA Translation

Place and route (PNR), is the next translation stage, and translates from the netlist level into the polygon level (in the case of ASICs), or a configuration bit stream (in the case of an FPGA). It takes a list of cells and/or macros, and arranges them in physical space (the act of placing them). The design must then be *routed*, which connects the placed cells together in the manner described in the input netlist. Like synthesis, PNR translates between abstraction domains. In this case, from the structural to the geometric domain.

Translators provide the means to automatically convert a manually generated, highly abstracted circuit design into a form which can be directly implemented. In the case of modern digital ICs these tools are indispensable due to the sheer size of these designs. However, the correctness of these designs must also be guaranteed. This is especially important for ICs using modern fabrication processes, as the financial overhead of producing a mask set is typically in the millions of dollars for 65nm and 45nm processes. Correcting mistakes in the design, whether physical or functional, is therefore a very expensive and time consuming task. Verification forms an extremely important part of EDA.

Different abstraction domains and levels have different verification requirements, since each abstraction domain is concerned with different aspects of the design. The checkers and verifiers introduced in this section are elaborated on in sections 2.1.3 and 2.1.4. HDL simulators are used to help verify the functionality of the design. Most HDLs have features which allow *test benches* to be written. This allows tests to be written. Specific stimuli can be generated, and resulting outputs from the HDL code can be captured and evaluated. Sometimes programming languages can be used in conjunction with the HDL to allow more sophisticated tests to be written. There are often entire teams dedicated to verifying the functionality of a large design.

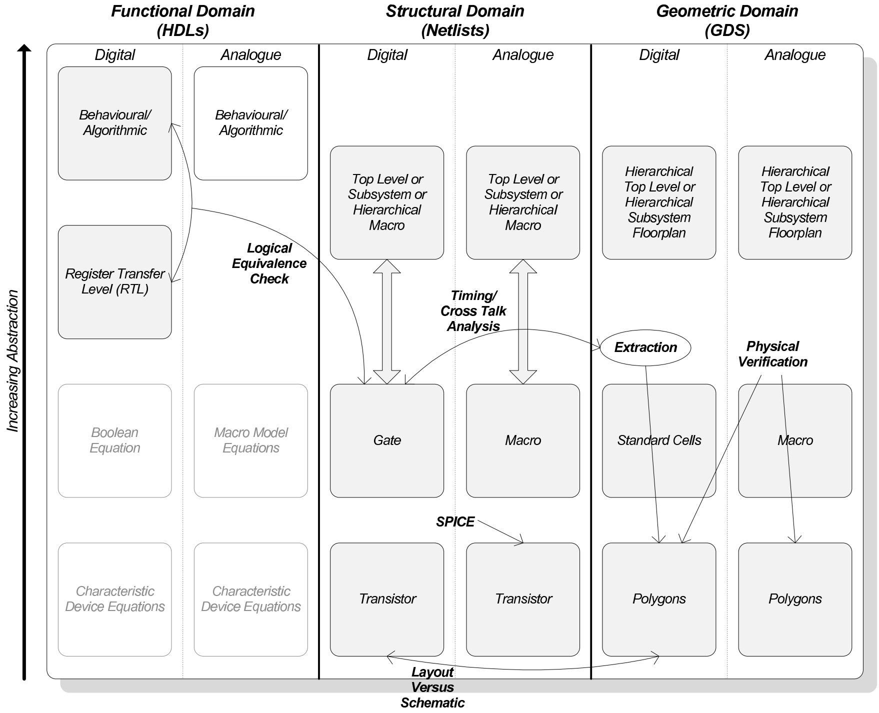

Functional Domain (HDLs)
Structural Domain (Netlists)
Geometric Domain (GDS)
Digital
Analogue
Digital
Analogue
Digital
Analogue
Increasing Abstraction
Behavioural/ Algorithmic
Behavioural/ Algorithmic
Top Level or Subsystem or Hierarchical Macro
Top Level or Subsystem or Hierarchical Macro
Hierarchical Top Level or Hierarchical Subsystem Floorplan
Hierarchical Top Level or Hierarchical Subsystem Floorplan
Register Transfer Level (RTL)
Logical Equivalence Check
Timing/ Cross Talk Analysis
Extraction
Physical Verification
Boolean Equation
Macro Model Equations
Gate
Macro
Standard Cells
Macro
SPICE
Characteristic Device Equations
Characteristic Device Equations
Transistor
Transistor
Polygons
Polygons
Layout Versus Schematic

Figure 2.3: EDA Verification

There are many different checks which need to be carried out in the geometric domain, and these are generally relevant only to ICs. Physical checks include analysis of the physical layout to check that the design rules published by the foundry are adhered to. Power simulations are also sometimes carried out to identify any areas of significant power dissipation or voltage drop within a circuit.

However, many checks are concerned with more than one abstraction domain. In digital designs, for example, logical equivalence checkers, compare representations of a design in the functional and structural domains to catch any errors that have been introduced during synthesis. A similar check, known as 'layout-versus-schematic' (LVS) compares the geometric and structural domains to catch any errors introduced during PNR.

2.1.3 A Typical Design Flow For Digital ICs

The design process, or *design flow* for digital systems is very well developed. Every stage of a typical digital design flow is capable of handling modern digital systems which often contain tens or even hundreds of millions of logic gates.

One of the first stages in designing a large digital system, after specifying what the design should do, is partitioning it into a number of *subsystems* or *blocks*, in order to make the design more manageable. There are different ways to accomplish this. The design may be partitioned with regards to the functionality of each subsystem, this may make the high level or *frontend* part of the design process easier. However, preference instead may be given to possibly important physical constraints and subsystems defined which will be easier to layout. When a system is partitioned, it is being abstracted at a high level in the structural domain. Therefore, the overall view of the system, (also known as the system's *top level*), contains instantiations of these subsystems but they are only empty *black boxes* at this level.

After the system has been partitioned, the exact functionality of each subsystem is defined with the use of an HDL - each subsystem is abstracted at a high level in the functional domain. Human designers write HDL code, usually RTL, which precisely defines what each block does. Functional verification goes hand in hand with this process, although the tasks of writing the RTL and verifying it are usually given to different engineers. These engineers will write test benches, often in a combination of the same HDL that the RTL was written with, and common programming languages like C++. These test benches specify logical stimuli for the HDL code, for use in logic simulation. They are also usually concerned with capturing and evaluating the output of these tests. These simulations must test the functionality defined in the RTL and show up any errors or unanticipated issues. At this point in the design cycle, the design exists in both high level structural and functional forms, but only the functional form must be verified. The partitioning of the system, or its structural form, cannot be said to be right or wrong, merely good or bad.

After the RTL has been completed and there is a reasonable level of confidence that it is correct, the frontend work is finished and the design cycle enters the *backend* stage, the target of which is to produce a *GDS database* (Graphic Data System). This is a database of polygons which a foundry will use to manufacture an integrated circuit.

The backend digital design flow makes use of pre-designed building blocks. These building blocks are usually classified as either *standard cells* or *macros*. Standard cells consist of basic logical units such as logic gates, flip flops, latches and so on. A typical cell library will not only contain cells of different functions, but a specific *type* of cell (which performs a specific function) will be further subdivided into versions of the cell which have different *drive strengths*. Macros refer to bigger, more complex cells such as memories, phase locked loops (PLLs) and digital to analogue, and analogue to digital convertors (DACs and ADCs). A library of standard cells and specialised macros will already have been designed and will be available to the tools used in a typical digital design flow.

Synthesis is performed automatically with the use of a dedicated tool - the RTL is translated into a netlist of standard cells and macros. Once this is done, the result is often verified by using a logical equivalence checker to confirm that no errors have been introduced during synthesis and the netlist represents exactly the same design as the source RTL.

The netlist must then be transformed into a GDS database, and so a PNR tool is used. The netlist specifies instances of standard cells and macros and also specifies how these are connected, so this is taken as an input by the PNR tool. The process is usually automated, although there is scope for the designer to manually specify the location of some (or, in small subsystems, all) of the standard cells, macros and the wire interconnects. In fact, it is quite common to manually specify the location of macros as these are not only usually much bigger than the standard cells, there are often far fewer of them. Manually placing standard cells will usually only be required for high performance designs, where specific sections of the design are required to have particularly low clock skew or low propagation times. An experienced human designer will usually make a better job of this than an automated tool, but at the cost of taking considerably more time and having a much lower capacity for coping with design size.

PNR usually produces a second, final netlist in addition to the GDS database. Additional buffers may be inserted into the design during PNR, and it is important to have an updated netlist for verification purposes. Additional logical equivalence checks are often carried out between the final netlist and the original RTL to once again prove that the functionality of the design has not been unintentionally altered during the digital design flow.

Once the GDS database is obtained, further checks must be carried out to ensure that other aspects of the design are correct. LVS is performed by comparing the GDS database to the final netlist, to prove that the correct design has been laid out - it is important here to first prove that the final netlist is correct (which has already been taken care of by the logical equivalence checks).

Much of the verification and design effort of laying out a digital system is concerned with *timing*. Digital signals take a finite time to propagate through standard cells and macros, and balancing the arrival times of signals and clock edges is crucial. Timing analysers will trace every logical path through the subsystem and check the propagation time against predefined, or automatically calculated constraints. The propagation time of a digital signal will depend on the number and type of cells it propagates through, the drive strengths of each of those cells, and the parasitic resistance and capacitance of the metal interconnects between them. It may also be affected by signal activity on other, capacitively coupled wires which are nearby. The parasitic resistance and capacitances of each metal wire, or *net* of a circuit layout is calculated by an *extraction tool*.

The activity of signals on nearby wires has the potential to not only cause additional timing delays, but it may actually corrupt a digital signal. Excessive noise injected from other wires may flip logic bits and lead to data corruption. *Signal integrity* analysers are designed to recognise situations where there is a risk of this happening.

A silicon foundry will have particular requirements and physical design rules which must be adhered to if the design is to be manufactured. These are physical layout rules, and define things like minimum wire width, maximum and minimum metal density, wire spacing rules and so on. These rules are becoming more complex with ever decreasing process geometries. Tools are available which check adherence to these rules. A *rule deck* is read in by the tool which describes how to check for the rules published by the foundry. This process is known by the rather vague term *physical verification*, although this could just as easily apply to any of the checks applied to the final layout.

Ideally, the PNR tool should produce a design which passes all these checks first time. Unfortunately, that is rarely the case for all but the simplest of designs. Human intervention is usually required, even if only to tweak the settings of the PNR tool. However, more direct intervention is usually required. Timing violations may be fixed by the human designer investigating the problem and picking new strengths or manually inserting buffers into violating paths. Signal integrity violations might require manual manipulation of specific wires, as might layout rule violations. Each time a change is made to the final layout, all of these checks are commonly run through again, to check that no new violations have been created by the changes. This process of fixing problems, and then re-checking the design is known as an engineering change order (ECO) loop.

2.1.4 A Typical Design Flow For Analogue & Mixed Signal ICs

The typical design process for analogue circuits is very different to that of digital. It is much less structured and less automated, and relies much more on human experience and intuition. Analogue circuits tend to be much smaller in terms of the number of components used, and it is quite rare to find an integrated circuit which is entirely analogue. It is far more usual for analogue circuits to form subsystems within a mixed-signal SOC. Typically, analogue systems will be designed by human designers who have extensive experience with particular types of analogue circuits.

As discussed in section 2.1.3, digital design flows can be roughly divided into *frontend* and *backend* processes, with the frontend tasks being primarily concerned with the functionality of the system and high level design abstractions, whereas the backend design process is concerned more with the physical implementation of the design and low level design abstractions.

This frontend/backend split is much less applicable to analogue circuits. When forming part of a mixed signal SOC, the design and functionality of any analogue subsystems it may contain are generally regarded as backend tasks.

When considering an analogue system or subsystem as a whole, there is very little which may be defined as being equivalent to the digital notion of a frontend design cycle. The functionality of the system must be specified - this usually takes the form of a simple list of requirements that the human designer will work to. The requirements of the circuit are simply stated, and no formal method or technique is generally used to capture circuit behaviour at a high level. Analogue circuits are not usually described using a high level HDL, and although this is possible there is currently no reliable way to automatically translate between the high level behavioural HDL description and a lower level structural netlist. This is discussed in more detail in 2.3. Analogue circuits cannot yet be automatically synthesised. There is little to no high level verification to do here, other than ensuring that the specified performance requirements are sufficient for the intended application.

The analogue design cycle really begins with the human designer creating a schematic of the circuit, based on knowledge and previous experience of the circuit type in question. Essentially what is happening is that netlist, a low level structural abstraction of the circuit, is created directly by the human designer. This netlist may be a combination of transistor and macro level abstractions. Put another way, it will likely contain individual, primitive components such as transistors but if a large system is being designed then the netlist may also contain analogue macros such as operational amplifiers or current mirrors. These may be considered analogous to digital cells.

Schematic capture tools are available that allow a transistor and/or macro level netlist to be captured graphically. However, for small systems a netlist may be manually written.

Once an initial design exists, the next step is to verify the function of the circuit using a simulator. This is almost always a spice-like simulator of some kind. If a schematic capture tool has been used to capture the design, a suitable netlist can be directly output from this. The simulator will be able to perform various types of simulation and report information on specific circuit characteristics back to the designer. Anything in the circuit's behaviour that does not conform to the stated specification can then be corrected. Thus, an iterative process begins during which the circuit functionality is refined. A comparison can be drawn here between verification of digital functionality at a high level using HDL simulators, and analogue functional verification at a low level using spice like transistor level simulators.

The analogue circuit must ultimately be physically laid out. Smaller, simpler circuits, or circuits with which the designer is more familiar may be laid out directly without going through the initial schematic capture stage. Analogue layout is again done by hand, in sharp contrast to digital PNR tools which almost, if not completely, automate the layout process. The initial schematic/netlist capture would effectively have assumed no parasitic wire resistance or capacitance. A circuit extractor can produce a purely transistor level netlist (no macros) from a physical layout. This will include modeled wire parasitics. Spice based simulators are again used to verify the circuit functionality after layout, and another iterative process can begin to refine and optimise the circuit.

It is here that perhaps one of the most important differences between analogue and digital circuit design can be highlighted. The correctness of analogue functionality is on a sliding scale. There is a point on that scale that marks *sufficient* correctness that guarantees that the circuit is *good enough* for its intended purpose. The functionality of digital circuits is simply right or wrong, which is why digital functional verification can take place at high level of abstraction, but analogue functional verification is still of concern at the lowest levels of abstraction. That is not to say that the physical implementation of a digital circuit cannot cause incorrect behaviour. Injected noise and timing issues are both a concern in digital systems, but these issues can be checked on a very localised basis. That is, an entire digital system need not be simulated to ensure that noise, timing or indeed voltage drop will not cause incorrect behaviour.

There are some checks that are exactly the same for both analogue and digital circuits.

Purely physical checks, such as adherence to the foundry's design rules are not concerned with whether the design is analogue or digital in nature. LVS checks are also useful here, and can catch errors introduced during manual layout of analogue circuits.

2.2 Evolution of EDA

In order to understand why the current state-of-the-art in EDA is as it is, it useful to examine how it has developed since it's beginning. Understanding not only how, but also *why* EDA has developed the way it has will help explain the differences in the current level of development between analogue and digital EDA. One of the primary applications of electronics is, and always has been, computing machines. As such the development of computers has been one of the main driving forces behind the increasing sophistication of EDA and therefore it is useful to examine how computers themselves have developed. This section contains a brief overview of the development of computers and a corresponding short history of EDA itself.

2.2.1 Early Computers And The First Integrated Circuits

Early computers were purely mechanical in nature, but during the first half of the 20th century the first *electronic* digital computers were built using thermionic valves. Famous examples include the *Colossus* machines built in secret by the British during World War II to help crack German communications encrypted by their *Enigma* machines.

These early machines were huge. The sheer size and complexity of these early digital computers demonstrates the first glimpse of what would become a very real problem for digital system and computer designers in the following decades. The problem is not so much the physical size of the machines, but rather the *size of the design.* As computers became more sophisticated, they would require a greater number of electronic switches and other components.

The invention of the transistor in 1948 by Brattain, Bardeen and Shockley [12] marked a critical turning point in the development of computers and electronic systems. Transistors are smaller, faster, use less power and are more reliable than thermionic valves. As a result, it was not long before computer designs made exclusive use of transistors. In the 1950s, IBM produced a series of transistorised computers which made use of its *Standard Modular System*[3]. These computers were made of a large number of printed circuit boards which contained a small number of logic gates or flip flops built from discrete transistors. Several of these boards could be connected together to form a CPU (central processing unit), for example.

Towards the end of the 1950s, the problem of ever-increasingly complex designs really began to make itself felt. Designs were starting to make use a huge number of components, and finding ways to connect all these components together was becoming a real problem. At the time, components were soldered together by hand and as designs become more complex it seemed as though newer designs would be composed primarily of wiring. This problem was known as the *'Tyranny Of Numbers'*[4]. One attempt to solve this problem was the 'Micro-Module' program [33], being developed at Texas Instruments (TI) and sponsored by the US Army. The idea was that by making all the components a uniform size and shape, and by including the wiring inside each component, that they could be simply snapped together, eliminating the need to hand solder them.

An engineer named Jack Kilby joined TI in 1958 and began working on the Micro-Module program. He was already familiar with the tyranny of numbers problem, but did not think the Micro-Module program offered a real solution to the problem as it did not address the huge numbers of components that were needed by complex designs. He began searching for an alternative and decided that the solution should be based purely on semiconductors. In 1976 Kilby wrote:

> *'Further thought led me to the conclusion that semiconductors were all that were really required that resistors and capacitors [passive devices], in particular, could be made from the same material as the active devices [transistors]. I also realized that, since all of the components could be made of a single material, they could also be made in situ interconnected to form a complete circuit'*[4]

On the 12th of September, 1958, Jack Kilby demonstrated the first integrated circuit. He connected a small piece of germanium to an oscilloscope, on connecting power to his IC it displayed a continuous sine wave. He had solved the 'Tyranny Of Numbers' problem.

Six months later, Robert Noyce, general manager of Fairchild Semiconductor (a company which he co-founded) independently came up with the same invention. Noyce's method solved some of the practical problems of Kilby's circuit. The way the metal interconnections, which wired together the individual components were laid down in Noyce's circuit made it much more suitable for mass production. Kilby and Noyce are usually credited as being co-inventers of the integrated circuit. Kilby received the Noble Prize for physics in 2000 for his invention. Noyce later went on to co-found Intel, along with Gordon Moore.

2.2.2 Early ICs & Design Methods

Jack Kilby's and Robert Noyce's invention was set to revolutionise the electronics industry and transform the world we live in. The Apollo space program provided one of the very first applications of the integrated circuit [5] [6] [7] - the Apollo guidance computer needed to be lightweight. The American space program in the 1960s therefore led the early development of the IC, but it was the use of ICs in the guidance system of the Minuteman missile that required their mass production [7].

From this point onwards, the number of transistors that could be squeezed onto a single IC increased exponentially. This was predicted by Gordon Moore in 1965. *Moore's Law*[40] states that the number of components that can fit onto a single IC will approximately double every 18 months. In 1966, the number of components that could be economically placed onto a single IC was approximately 20 [55]. This is known as *Small Scale Integration* (SSI), which describes ICs containing tens of components. The next step was to produce *Medium Scale Integration* ICs, which began during the late 1960s. These chips contained hundreds of components. Economics was a significant driving force here, as the production cost of MSI chips was not much more than that of SSI chips, more complex systems could be produced whilst requiring fewer components and therefore smaller circuit boards.

At the time, the design process for these ICs was very manually intensive. The circuit schematics were manually designed at the transistor level. These schematics were then laid out entirely by hand. Photolithographic mask creation followed a similar process to that used to create printed circuit boards (PCB). The hand drawn layout was manually transferred to rubylith. Rubylith is a transparent red film which was used to create a template for the mask. Polygons were manually cut into the rubylith using craft knives, the result was then optically reduced in size to make a mask.

Designs of this time were inextricably linked to the underlying physical process. This not only meant that the design of IC was tied to a particular manufacturing process at a particular silicon foundry, but the circuit designers had to take aspects of the physical process into account at most stages of the design process.

The next important milestone came with the development at Intel of the 4004, the world's first microprocessor, in 1971. This was a 4-bit device fabricated using a 10 micron process and contained 2,300 transistors[1]. This device was entirely hand designed, again using manually cut rubyliths in order to produce its masks.

2.2.3 The Mead-Conway Revolution

The size of IC designs continued to developed through out the 1970s, with the introduction of *Large Scale Integration* (LSI) chips in the middle of that decade driven by the same economic forces that led to the development of MSI chips. These chips could contain tens of thousands of components.

A few years before this, Carver Mead, a professor at Caltech whose field of expertise was in device physics, had made some predictions about fundamental transistor size limitations [38]. He realised that with the correct scaling, everything improved. Power went down, speed went up and it was possible to fit more transistors onto an IC. He then spotted the inevitable problem with ever increasingly complex circuit designs. Kilby and Noyce had solved the tyranny of numbers problem some years before with the invention of the IC. However, that had only addressed the problem of physically building circuits containing thousands of components. The removal of that constraint and the ensuing explosion in circuit complexity revealed what was effectively a new 'tyranny of numbers'. As Carver Mead himself wrote:

> *By 1969, it was very clear to me that someday we were going to be able to put millions of transistors on a chip. That meant that I was working on the wrong end of the problem. If we really could make a million transistor devices, the key issue wasn't about wafer fab or semiconductor device physics, it was about, "How the hell do you design something with a million working parts?"*[56]

In the late 1970s Mead began working with Lynn Conway, an engineer working at Xerox's Palo Alto Research Center (PARC). Conway had a background in system design and computer architecture. Their aim was to develop improved methods of designing LSI and VLSI (Very Large Scale Integration) ICs. They began 'designing the design methods' to address this problem. Each of their fields of expertise was well suited to this task, as each addressed each 'end' of the problem, both the physical and architectural aspects. This work culminated with the publication in 1980 of their book, *Introduction To VLSI Systems*[37]. This was a landmark text and marked the beginning of the *Mead-Conway Revolution.*

One of the main contributions Mead and Conway made was in calling for a clean separation of the architecture of the system from the underlying device physics, a separation of design from technology. This was in sharp contrast to how things were usually done at the time. In this proposed new organisation of industry and work methods, the circuit designers would concentrate on coordinating the operation of thousands (or more) of distinct parts of a machine, and the technologists and physicists would concentrate on reliably fabricating ICs and ever smaller transistors.

In order to facilitate this separation, Mead and Conway produced *simplified, scalable physical design rules.* These are also known as the *lambda(λ) design rules.* These rules greatly simplified the constraints on how a circuit could be laid out, constraints such as minimum feature size and spacing rules. Uncertainties like mask misalignment, over or underexposure of photoresist and over-etching were combined into a single, dimensionless length unit represented by the greek letter lambda(λ). This represented the fundmental resolution of the manufacturing process, and layout constraints were expressed in integer multiples of λ.

The other main contribution made by Mead and Conway was the advocation of maximising the amount of design automation used in the design process. They advocated a process known as *silicon compilation*, the idea being that system designers could design ASICs without getting involved with circuit layout issues. Mead wrote the worlds first silicon compiler. In his own words:

> *At Caltech, I built an artwork [layout] language that allowed you to enter feedback terms, minterms, and outputs and would automatically generate the layout. Then you could give it a truth-table for the microcode, and it would proudce the tooling for an IC. This was the first silicon compiler; it was great fun.*[56]

Mead and Conway were also responsible for a breakthough in education, and very soon many universities were teaching courses based on their book. Their methods not only meant that in general less people were needed to design a given IC, but the number of IC engineers increased dramatically as a result. They firmly established EDA as a discipline in its own right, and are the originators of EDA and IC design techniques as they would be recognised today.

2.2.4 HDLs and Modern EDA

During the early 1980s, the Mead-Conway revolution began to take effect. A number of EDA startups were founded with the aim of producing process-independent EDA systems. Three companies particular note were founded at this time, these were Daisy Systems, Mentor Graphics and Valid Logic. All three companies debuted at the Design Automation Conference in New Mexico in 1982. They were known in the industry at the time as the 'DMV'. They started producing the first commercially available schematic capture, logic simulation, automatic layout and test generation tools.

Throughout the decade such tools became more common place with more companies offering a wider selection of EDA tools of increasing sophistication. In 1984, a company named Automated Integrated Designs Systems produced the Verilog language. It began life as a proprietary language, and the company that created it later changed their name to Gateway Design Automation which was later acquired by Cadence. Cadence eventually transferred Verilog to the public domain, supervised by the Open Verilog International (OVI) organisation. That organisation is now known as Acellera. The language was eventually standardised in 1995 by the Institute of Electrical and Electronic Engineers (IEEE), IEEE Std 1364-1995 [22].

Another HDL, the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) was developed at around the same time. It is influenced heavily by the ADA programming language. The United States department of defence requested the development of the language to document components used in military equipment. It was standardised by the IEEE in 1987, IEEE Std 1076-1987 [21]. EDA companies began marketing simulators and synthesis tools for both of these languages throughout the late 1980s.

Analogue variants of both of these languages exist. Verilog-AMS is combination of Verilog-95 and *Verilog-A*, which is only capable of modelling continuous time analogue behaviour. Verilog-AMS is a mixed signal HDL, and was first standardised by OVI in late 1998. No IEEE standard exists for this language.

VHDL-AMS, however, was developed directly from VHDL-93 and no analogue-only variant of VHDL has ever existed. VHDL-AMS was standardised by the IEEE in 1999, IEEE Std 1076.1-1999. Simulators exist for analogue HDLs, but no commercial synthesis tools currently exist.

Modern EDA tools now consist of an extensive selection of automated tools that assist with every aspect of the digital design flow, including synthesis, place and route, high and low level simulators and many others.

2.3 Challenges Faced By Analogue EDA

Further development of analogue EDA poses some unique challenges. In terms of sophistication and level of automation, there is a gulf between the state of the art of analogue and digital EDA. In order to develop this field further it is important to understand why.

Some of the main driving forces that have contributed to the sophistication of digital EDA simply have not applied to analogue. As has already been discussed, the development of digital EDA, as well as the integrated circuit itself, has been driven by the development of the digital computer. Although analogue computers have successfully been used to solve complex problems, they simply cannot match the flexibility or adaptability of their digital counterparts. The rapid development in the power of digital computers, which is, ironically in part due to the development of digital EDA, has rendered them obsolete.

The key factor in this difference is the in the nature of analogue and digital circuits themselves. It is useful here to consider exactly what digital circuitry *is*. In reality of course, there is no such thing as a digital circuit. The real world is analogue in nature, and of course digital circuits are built from transistors. The signals that pass between them are not the precise, neat square waveforms typically displayed by HDL simulators but are smooth, curved waveforms characteristic of the charging and discharging of RC circuits. Digital behaviour is therefore an approximation of the analogue nature of the actual circuits. Being concerned with only two distinct states allows much of the complexity of analogue behaviour to be ignored.

This has extremely important implications for design automation. The majority of digital EDA tools and design techniques rely on the fact that the functionality of a digital system can be decomposed into functional primitives which can map directly onto physical circuits and have precisely predictable behaviours. What is more, each of these functional primitives may be realised in greatly differing physical implementations of arbitrary manufacturing processes and yet still yield exactly the same functionality. An AND gate on the Intel 4004, for example, will serve exactly the same function as an AND gate found on the most modern microprocessors. The difference in the physical size, switching speed and power dissipation (the analogue properties of the gates) will of course be considerable.

This easy decomposition of digital behaviour allows digital circuits to be modeled at a number of levels of abstraction, and fairly finely-grained abstraction levels at that. This in turn allows for easy, and therefore automated, translation between these levels. Analogue behaviour, however, cannot be easily decomposed. There are fewer abstraction levels which can describe analogue circuits and they are further apart, they are coarsely-grained. Translation between these levels is considerably more difficult.

The functionality of digital circuits, then, may be cleanly separated from the underlying device physics as Mead and Conway advocated. However, this is not possible with analogue circuits. The Mead-Conway revolution, which was probably the most important event in the development of digital EDA passed analogue circuits by, because it simply *cannot* apply to them. Analogue circuits are very much more dependent on their physical implementation, and as analogue behaviour cannot easily be decomposed into simpler functional units, the level of design automation that may be applied to them is inherently limited.

Despite these difficulties, the field of analogue EDA is not non-existent. EDA covers many varying techniques and methods and serves many different purposes, and one area where analogue EDA is well developed is in simulation. Although simulation does not *directly* contribute to the design of a circuit, it is never the less an extremely important tool in the design process of both analogue and digital circuits. It provides the designer with feedback and allows for an iterative process of refinement without the need to manufacture the circuit, which in many cases, and especially for ICs would be prohibitively expensive.

Analogue simulation has been around for a long time, and the first analogue simulators were developed in the 1960s. The majority of analogue EDA that does exist actually performs some form of simulation, and in some areas the distinction between analogue and digital EDA blurs. Extractors, timing, cross talk and power analysers are all very important when developing a digital system, but actually deal with analogue quantites. Many of these tools use some form of SPICE or similar analogue simulation.

Simulation provides an answer to the question 'I have this circuit, what is the behaviour?'. Synthesis, or perhaps more correctly *silicon compilation* provides an answer to the question 'I have this behvaiour, what is the circuit?' and is, and always has been more difficult for digital circuits, and so far virtually impossible for analogue circuits.

Chapter 3

APPROACHES TO ANALOGUE SYNTHESIS: ATTEMPTS AT SOLVING THE PROBLEM

While automatic high level synthesis of analogue circuits currently does not exist as an accepted, recognised part of a typical modern design flow, and certainly does not exist in a mature commercial capacity, nevertheless there have been many attempts in academia to produce such a process. These attempts vary considerably in their approach, flexibility, underlying methods and initial data input requirements. However they do all share at best limited success.

This chapter will examine the most common methods attempted. It will also describe what is commonly meant by the term 'synthesis' and provide a definition for this thesis, and also draw a distinction between synthesis and optimisation.

3.1 When Is Synthesis Not Synthesis?

Before examining possible solutions to problem of analogue circuit synthesis, it is important to define precisely what synthesis is. The process of synthesis is very well defined for digital circuits, and its role in a modern design flow is clear. As illustrated in figure 2.2, it is the process of translating a high level circuit abstraction in the functional domain, into a lower level abstraction in the structural domain. More specifically, a digital synthesis tool usually reads in either an RTL, or sometimes behavioural circuit description typically written in Verilog or VHDL and produces a netlist of gates, often in the same HDL as the input. The process is best defined in terms of both the level and *domain* of abstraction of its inputs and outputs.

The 'synthesis' concept appears to be much less well defined for analogue circuits. Almost all experimental analogue synthesis systems produce a similar form of output to that of the digital process, usually a transistor level netlist, or maybe a mixed transistor and macro level netlist. In this respect these systems are not altogether dissimilar to digital synthesis systems, both outputs are *structural* netlists of components, whether primitive in nature or not. However, these experimental systems vary considerably in their input requirements.

The reason for this lack of consistency is perhaps twofold. The form, and general use, of high level analogue abstraction is less well established than for digital. There are a number of reasons for this. Analogue circuit functional specification is often dependent on the particular *type* of circuit in question (and therefore dependent on the circuit's *application*). For an amplifier, things like frequency response, input and output impedance, phase response and power consumption may all be stated requirements. While these parameters may be considered to form a high level functional specification, it is a long way from the flexibility and sophistication of digital HDLs. Analogue HDLs do exist, however, but not only they are a relatively recent development, the fact that no mature commercial analogue synthesis tools exist limits their usefulness and as such are not widely used. Specifying the function of an already well understood circuit type such as an amplifier in an analogue HDL may even be overkill.

The second reason may simply be because high level analogue synthesis is such a difficult process. Automating analogue circuit design is certainly desirable, and in the absence of automation on a comparable level to digital, the idea of synthesising an analogue circuit may easily become a more nebulous idea of merely *assisting* the design process in some way.

It is here that care must be taken not to blur the distinction between synthesis and *optimisation*. Some of the synthesis systems examined in this chapter are, in the opinion of the author, essentially indistinguishable from circuit optimisation. For example, varying component values of a user-supplied topology to achieve a closer match to a given functional specification would certainly appear to be more like optimisation than synthesis.

3.1.1 A Definition Of Circuit Synthesis

A concise definition for synthesis, as used in this thesis, will now be stated: *Circuit synthesis is the process of translating from the functional abstraction domain to the structural abstraction domain.* This is true for both analogue and digital circuits. Synthesis is not necessarily the translation from higher abstractions to lower abstraction levels, the key point is that synthesis is the translation between abstraction *domains*. This is why it is useful.

It must be noted, however, that while synthesis does not necessarily need to translate from high abstraction levels to lower ones, in digital synthesis this is usually the case. Translating ‘sideways’; that is, translating from a given level in the functional domain to an equivalent level in the structural domain, would also qualify as circuit synthesis.

Abstraction levels in a given domain do not *necessarily* have direct equivalents in other domains. A high level algorithmic description in the functional domain is not necessarily equivalent to a top level system netlist. However, in some cases it would seem reasonable to draw direct comparisons, for example between boolean equations in the functional domain and the gate level in the structural domain. This is illustrated in figure 2.1.

3.1.2 A Definition Of Circuit Optimisation

A concise definition for optimisation, as used in this thesis, will now be stated: *Circuit optimisation is the process of improving the performance of a given circuit for a given application, with no translation between abstraction levels or domains taking place.* In other words, the input and output abstraction level and domain of an optimisation process is the same.

3.1.3 Knowledge Re-Use

Knowledge re-use is considered to be good practice in many different fields. The re-use of source code is strongly encouraged in the field of software engineering, and re-use of HDL source code can also be of use in digital system design. There is little point in re-inventing the wheel.

It makes sense for the same principle to apply in circuit design, and indeed *manual* analogue circuit design is dominated by it. The vast majority of analogue circuits are essentially some form of filter or amplifier, and over the years analogue designers have built up a very extensive library of known good topologies and sub-circuits such as current mirrors and darlington pairs. Human designers typically use one or more of these known topologies and refine it for the intended application.

If knowledge re-use makes good sense for manual circuit design, it reasonable to think that it might also make good sense for design automation. Digital synthesis tools implicitly re-use known good topologies by the use of the supplied cell library. By outputting a *gate* level netlist they are necessarily constrained to use the typically hand designed topologies of each cell. However, this form of knowledge re-use happens at quite a low level and has a very clear and well-defined boundary. Re-using known good topologies at the gate level is less common.

There are advantages to *not* re-using topologies, or at least in not relying too heavily on them. Because digital synthesis tools are quite capable of not using any known-good topologies at the gate level this means that they have complete freedom to produce novel designs.

Knowledge re-use therefore is very valuable when used correctly, but the situation is not so well defined for automated analogue synthesis. One of the reasons that human designers make such extensive use of known good topologies is because arbitrarily designing a novel, unique analogue circuit from scratch will typically be considerably more difficult than for digital. If the design process is automated, then maybe it might be productive to give the synthesis tool complete freedom in producing novel circuit topologies? The problem with doing this, of course, is that the synthesis tool is being asked to start off from a lower level than digital synthesis tools. The task is already more difficult.

This is actually just another way of looking at the fact analogue circuit function cannot easily be decomposed into functional primitives. It is this property that allows digital synthesis tools to make use of a strict, well-defined boundary of knowledge re-use. They do not have to work at the transistor level, and the task of piecing together the functional primitives (digital cells) is much easier.

It would seem advantageous to use some form of knowledge re-use in an analogue synthesis system, and the various experimental methods that have already been tried allow for greatly differing levels of topology flexibility and re-use of common analogue sub-circuits. As will be seen, the methods tend to follow an ‘all or nothing’ mentality. Either the topology tends to be completely, or almost completely restricted to a particular circuit type or topology, or the synthesis tool has completely free reign to wire up individual transistors and fails to make much use of the vast library of filter, amplifier and other analogue sub-circuits that already exist.

It is not at all clear what the correct level of knowledge re-use is for the automated design of analogue circuits, but it seems likely the correct level will be dependent on both the functional specification and application of the circuit.

3.2 Related Work

Analogue design automation is certainly a diverse area of research. Advancing the current state of analogue EDA is a desirable goal, however it is most certainly not an easy task. The many diverse methods that have been reported in the literature are all significantly limited in some way: they share limited success in what they set out to achieve, or they are limited in the range of circuits they can deal with or in the level of design automation that they achieve.

Despite the numerous claims of substantial success that exist throughout the literature, the absence of any form of mature, commercially available analogue synthesis tools is a strong indication that these methods are all in some way unsatisfactory. Any system which significantly automates the analogue design process, is realistically usable *and* consistently produces an acceptable quality of results (QOR) would surely be of considerable worth.

Perhaps one of the reasons for the number of claims of success in solving the problem of analogue synthesis is, as discussed in section 3.1, the lack of a precise, accepted definition of 'analogue synthesis'. The majority of synthesis tools reported in the literature are based on wildly differing concepts of what this actually means.

This literature survey first briefly examines common methods of circuit synthesis and optimisation. Each method is accompanied by examples of their use. After that, the following issues are discussed for each example: the level of design automation offered, the reported quality of results (QOR), the ability of the technique to explore new circuit designs, and finally any circuit simulators or analysers that are employed by the technique are examined.

3.2.1 Methods For Circuit Synthesis & Optimisation

The literature reports many different attempted approaches and techniques for the synthesis and optimisation of analogue circuits. This section describes some of the more common ones. However, there are still many different ways of implementing a given method.

Gradient Descent

Gradient descent, also called the *method of steepest descent* or the *gradient method* was introduced in 1847 by the French mathematician Augustin Louis Cauchy [31]. It is an optimisation method that works in spaces of any number of dimensions. It is designed to find local minima of the objective function, $f(x)$, where x is vector of optimisable parameters. It is a relatively simple method.

An initial point, x_0 in the solution space of $f(x)$ is required. In order to move to lower values of the objective function fastest, the *negative* gradient of x_0 must be followed. The new point, x_1 can therefore be found using equation 3.1, where γ represents the *step size* taken.

$$x_{n+1} = x_n - \gamma_n \nabla f(x_n) \tag{3.1}$$

The process is iterative, and should eventually converge on a local minimum. Note that γ may change on every iteration, and choosing an appropriate step size is one of the main parameters that determines the effectiveness of this method. The bigger the value, the fewer number of iterations will be required for convergence, at the risk of moving to a new point which is higher than the current point. The step size should be chosen such that equation 3.2 is satisfied.

$$f(x_0) \geq f(x_1) \geq f(x_2) \geq \ \ldots f(x_n) \tag{3.2}$$

Some implementations of gradient descent first check that the new point is indeed lower than the current one before moving to it. Methods also exist that attempt to pick an appropriate value of γ for each move. Simply using a fixed value may produce poor results, although this depends to an extent on the problem being solved.

Gradient descent also has other problems. Even if a sensible value of γ is picked for each iteration, convergence can be slow depending on the curvature of $f(x)$. But perhaps the most significant issue is the fact that gradient descent is virtually guaranteed to converge on local, rather than global minima. For this reason it is sometimes used in conjunction with other methods for circuit optimisation [39]. It is more likely to be used for *optimisation* rather than synthesis or topology modification.

Simulated Annealing

Simulated Annealing (SA) is an optimisation algorithm originally described by Kirkpatrick et al [25]. It attempts to find the *globally optimal* solution to a problem and is specifically designed to try to avoid local minima, and can optimise for an arbitrary number of variables. It takes an arbitrary, initial state of a system and attempts to 'freeze' the system into a lower state of energy.

It is modeled on the metallurgical process of *annealing*, which freezes atoms in a material into a crystalline structure. The material is heated almost to its melting point, and then goes through a slow, controlled cooling process. The heating causes individual atoms to be freed from their initial positions, and the cooling means they have more chance of finding a lower energy point than they started from, after randomly passing through higher energy states. A crystalline structure starts to form. Semiconductors are sometimes annealed after doping via ion implantation. This particular doping process can damage the silicon crystal lattice, and so the semiconductor is annealed in an attempt to repair the damage done.

The SA algorithm mimics this process. The exact details of an SA implementation will vary and to an extent will be dependent on its application. The system being modeled has an *objective function*, $f(s)$ which is analogous to the system's *internal energy*, which in turn is dependent on its internal state s. The internal 'state' of the system is a vector of the paramters being optimised. The main variable controlling the algorithm is the *annealing temperature*, T which will be a user defined starting value, T_i.

Figure 3.1 illustrates the SA algorithm. It begins with the system in an arbitrary state, s. A new trial solution, or *neighbouring state* is typically generated as shown in equation 3.3, M is a user specified vector of maximum allowed changes to each parameter and u is a vector of (possibly constrained) random numbers.

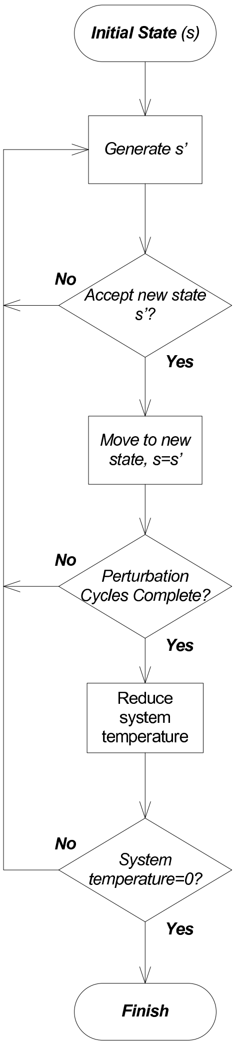


Figure 3.1: Simulated Annealing Algorithm

$$\acute{s} = s + Mu \tag{3.3}$$

The neighhbouring state is accepted as the new current state according to the *acceptance probability*, $P_a(s, \acute{s}, T)$ which is a function of the current and neighbouring states and the current system temperature. Equations 3.4 and 3.5 demonstrate how P_a might be calculated. The difference in the objective function between the two states (essentially the difference in system energy) is given by δf.

$$P_a = e^{\frac{-\delta f}{T}} \tag{3.4}$$

$$\delta f = f(\acute{s}) - f(s) \tag{3.5}$$

This loop of picking a neighbouring state, and moving to it if accepted is repeated a user specified number of times for a given temperature. When this loop has been exhausted, a new loop will begin with a lower system temperature. The manner in which the temperature is decreased is known as the *annealing schedule*. Again, there are various ways to implement this but a common method, as shown in equation 3.6 is to use a geometric method, where the new temperature, $\acute{T}$ is multiplied by a *decay factor*, α.

$$\acute{T} = \alpha T \tag{3.6}$$

The algorithm is usually terminated when the system temperature reaches or approaches zero. However, the algorithm may need to be stopped if an alloted computation time budget is exceeded. If this is the case, then it may indicate that the annealing schedule has been poorly chosen. It also may be stopped if a sufficiently good solution is found. However, this generally goes against the spirit of SA and it is more usual to let the algorithm continue until the system is completely frozen. This increases the chance of an even better solution being found.

Many of the parameters of an SA algorithm will depend on the application and therefore will be user defined. However, there are a few key points which must be taken into consideration.

- The neighbour state selection function should be skewed such that new states have an energy value close to the current state. Simulated annealing is designed to ‘freeze’ the system into a globally optimum, or at least almost globally optimum energy state. Therefore, the current state is expected to have a much lower energy than a random state. Moving to another state of similar energy will have a tendency to exclude both very good, and very bad states as compared to the current state. However given that very bad states are likely to be much more common than very good ones this heuristic tends to produce effective behaviour.

- The acceptance criteria of P_a have some critical requirements. P_a *must* be non zero when

$f(\acute{s}) > f(s)$. This allows transitions to states with a higher energy, in other words states which represent a *worse* solution. It is this property which allows the algorithm to avoid local minima. However, if $f(\acute{s}) < f(s)$ then in many SA implementations $P_a = 1$. Although this is how Kirkpatrick originally described SA, it is not essential for the algorithm to work. This criterion may be chosen at the users discretion. Finally, as $T \Rightarrow 0$, if $f(\acute{s}) > f(s)$ then it is a requirement that $P_a \Rightarrow 0$. The algorithm should favour downhill transitions, to a lower energy state as the system approaches 'freezing point'.

- The annealing schedule is also of crucial importance. The initial temperature T_i should chosen such that the system is completely 'melted'. However, the system should reach freezing point within an acceptable time frame.

Simulated annealing can be used to optimise component values in analogue circuits, with the set of component values making up the state vector s. In such a case, the vector M of maximum allowed changes would need to be tailored to the components contained within the circuit.

Using SA to optimise the topology of a circuit is also conceivable. In this case s would need to contain connectivity information for each of the components. However, getting SA to explore an *expanding* solution space, that is, to allow circuits with an arbitrary number of components is more difficult and therefore is not as well suited to synthesis as it is to optimisation.

In order to apply this method to circuit optimisation, a external simulator will usually be required in order to evaluate the neighbouring state. This will also require a function that compares the result of the simulation with a ideal, target circuit behaviour or characteristic.

SA forms the central part of the ASTRX/OBLX system [44] [43] [42]. This is one of the analogue synthesis systems reported in the literature. It is made up of two tools: *ASTRX* compiles the user supplied circuit topology and specifications, and translates them into constraints for the solution tool, *OBLX*, which generates the output circuit. OBLX is a search engine based on simulated annealing, and modifies the transistor sizes and component values.

Using SA for topology generation or optimisation is much less common than using it for component value optimisation, although it has been done [39]. It has also been used as part of a wiring algorithm [54].

Genetic Algorithms

Genetic Algorithms (GAs) were invented by John Holland [20]. They can be applied to all manner of different problems in many different fields, including circuit synthesis and optimisation. This section provides an overview of GAs and looks at examples of their use, in order to contrast them to the other methods examined in this chapter. These methods are expanded on in Chapter 4 in much greater detail, concentrating on the specifics.

Principles and ideas from the study of biological evolution are central to the GA, which may be thought of as *simulated evolution*. The algorithm maintains a population of candidate solutions to a problem, and applies a variety of operators to them over a number of iterations, or *generations*. If successful, the population will contain increasingly good solutions to the problem until an acceptably good solution is found. The solution is *evolved*.

Central to the operation of a GA are the structures used to represent the problem solutions. These structures must map in some way to a meaningful solution to a problem. A very common structure in GAs is the *string*. For example, if a GA was being applied to the problem of optimising an analogue filter, a string of numerical values might be used as the *encoding scheme*. Each value could represent the value of a particular component in the circuit topology.

The form of these structures is important because the GA will apply the *crossover* operation to them. Crossover is one of the defining characteristics of the GA and is designed to produce new solutions which have a combination of useful traits and features from other good solutions. Combining several characteristics of good solutions will ideally produce an even better solution.

Figure 3.2 shows the basics of how a GA works. Each point in the algorithm is described below. One complete execution of the loop results in a new population, or generation.

1. The initial population of solutions is randomly generated.
2. Each individual is then measured to determine how good a solution it is, and during this process it is assigned a *fitness score*. The method used to calculate this score will depend upon the application.
3. The best individuals are probabilistically *selected* for crossover. A given individual may

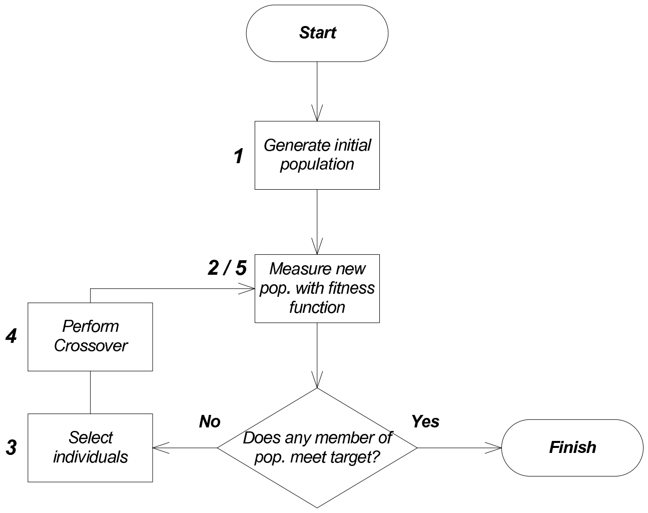

Start
1
Generate initial population
2 / 5
Measure new pop. with fitness function
4
Perform Crossover
3
Select individuals
No
Does any member of pop. meet target?
Yes
Finish

Figure 3.2: Overview of Genetic Algorithm

be selected for crossover multiple times.

4. Crossover is probabilistically performed on the selected individuals, and the 'offspring' resulting from crossover will go through to the new generation. Crossover usually only happens to a proportion of those selected for crossover, meaning that some individuals pass through unchanged into the new generation.

5. At the end of the GA loop, a new generation of solutions is obtained containing some of the best individuals from the previous generation, and some new solutions obtained from mixing good solutions of the previous generation.

In contrast to the methods previously described in this chapter, GAs are a blind search, and do not search through points in the solution space in a linear fashion. In both the gradient descent and simulated annealing methods, new solution points are close to the current solution point. New solution points in a GA search may be very far away from the current one.

There are several examples of GAs being applied to analogue circuit design in the literature. Zebulum [58] reports, amongst other things, attempts to optimise analogue circuits including CMOS OpAmps and a Miller CMOS operational transconductance amplifier (OTA). Zebulum introduces the concept of variable length strings (these are discussed in more detail in chapter 4), although some of the case studies reported, including the ones mentioned here, are based on the more common fixed length strings.

Koza uses a novel GA representation system evolving computer programs to address the issue. *Genetic Programming* [26][30][27][28][29] evolves LISP programs which, when executed, construct an electrical circuit. Detailed discussion of GP is deferred to chapter 4. The representation system used by GP allows a very dynamic sizing of the problem solution; it is not a fixed length system.

Sripramong [50] employs a GP system with the addition of a current-flow analysis technique in order to prune poor solutions early on in the progression of the GA. A variable population size is also employed here. The system is focused primarily on the synthesis of CMOS op-amps. This system uses a novel technique to perform pruning and correction on the GA's population. *Current Flow Analysis* allows easy identification of isolated or redundant components and can prune or correct them as necessary. This perhaps goes against the spirit of GAs, although poor circuits with extraneous or poorly connected components do tend to proliferate within the population.

Matiussi [36][35][34] uses a system based on variable length strings of alphabetic symbols. This system is very different to many other systems based on strings, the encoding scheme is much more complex. It can be used to evolve the of topology of various types of network. For example, both neural networks and analogue circuits can be evolved. The encoding scheme used by this system is perhaps the most similar to biological DNA sequences of all the encoding schemes reported in the literature. One of its most salient features is the interaction between different sections of these sequences of letters to produce a measure of 'interaction strength'. This is used as a basis of determining topology of a network. Certain markers within the sequence are used to define instances of devices within the network. Letters associated with subsequent 'terminal' markers of different device instances are used to determine the interaction strength between terminals of those devices. A network topology can be deduced this way.

Other Methods

The methods discussed so far are by no means the only ways of addressing this problem. Systems based on artificial intelligence (AI) have been tried, these systems and others also make use of predefined, existing design knowledge.

The BLADES system [17] automates analogue circuit design via the use of an expert system. It essentially attempts to mimic the process that human designers go through when designing an analogue circuit, and as such tackles the problem in a top-down fashion. The system attempts to deduce a high level topology based on the circuit constraints, made up of 'black-boxed' sub-circuit building blocks. It derives the specifications of each sub-block from the overall system specifications provided by the user. It then progressively breaks down these building blocks into smaller ones until it reaches a level where the sub-blocks can be mapped onto a discrete sub-circuit topology. BLADES employs a predefined knowledge base of formal knowledge and heuristic knowledge, taken from design experts.

OASYS [19] is based on similar concepts. It is not based on an expert system, however it does depend on predefined, mature design knowledge. It also tackles the problem hierarchically, using a top-down approach, again in an attempt to mimic the design process typically followed by human designers. A selection of circuit topologies are predefined. These topologies, however, are specified hierarchically, as an interconnection of sub-blocks. The overall topology of the solution is selected at a very high level, this then provides a framework in which to size each sub-circuit rather than focus on designing the interconnections between each sub-block.

There have also been attempts to synthesise analogue circuits using a high level HDL description as a starting point. Both the VASE system [13] [14], and the system reported by Kazmierski [23] [16] take the circuit specification in the form of behavioural VHDL-AMS. Both of these systems also rely on predefined knowledge. They use the parse tree of the supplied source code to derive structure at a high-level. This leaves a netlist of black boxes which each perform a given function. These black boxes are then mapped onto transistor level netlists. There have also been attempts to define a standard, synthesisable subset of the VHDL-AMS language [15].

3.2.2 Level Of Design Automation Achieved

Central to the question of how successful a circuit synthesis tool, or indeed any EDA tool is, is how much of the design process it has succeeded in automating. It is crucial to take this into account when judging the success of the synthesis methods reported in the literature. A synthesis system which requires the user to do the majority of the design work may not offer much advantage over simply performing the entire task manually, even if the system does produce good QOR.

Some measure of the level of design automation achieved can be obtained by comparing the input requirements of a system to its output. This can identify them as being optimisation rather than synthesis systems, although as already stated there is no universally accepted, clear definition of analogue synthesis and very different concepts of this may be found in the literature. However, in this survey the definitions stated in sections 3.1.1 & 3.1.2 will be used as a yard stick.

One thing that is quite apparent from the literature is that the various systems that exist require very different amounts of *preparatory effort*. This is the amount of effort needed in order to set up the system for generation of the desired circuit, and is a factor when judging how much of the design process has been automated. However, this is by no means the most important consideration in design automation and it has no bearing on the definition of synthesis given in section 3.1.1.

As discussed in section 2.1.1, human designers necessarily work at high level of abstraction when dealing with complex systems, and the most natural starting point for the majority of design problems is in defining *what* that system should do; in other words, the behavioural abstraction domain. Translation between abstraction domains is one of the most important tasks that EDA tools perform. A system which nominally requires little preparatory effort but requires the user to supply the circuit topology is in fact leaving a very large part of the design process to the user.

A good example of this is the ASTRX/OBLX system. Much emphasis was placed on reducing preparatory effort in the design of this system. This is reflected in the definition of automation used by the authors. It is defined as being the ratio of the time it takes to manually design a new circuit to the time taken by an automated system. The time taken with an automated system is considered here to be the combination of the time needed for the synthesis process itself and the preparatory time.

Method/ Authors	**Input Requirements**	**Setup Effort**	**Automation Achieved**
SA Methods			
ASTRX/OBLX	Circuit topology & Constraints	Other than supplying the topology & constraints, the user must also supply a 'test jig' for the circuit.	The supplied input abstraction is structural, and the output is also structural. This is optimisation.
GA Methods			
Zebulum	Circuit constraints, the circuit topology is implicitly input via the GA implementation	Minimal, the user must provide the performance specifications.	Both the input and output abstractions are structural. This is optimisation.
Koza (GP)	Circuit performance specifications and/or behaviour characteristic. Also requires an embryonic circuit to be provided.	Depends on difficulty of specifying or generating specifications or behaviour of required circuit.	Input abstraction is functional, output is structural. Translation across domains takes place. This is synthesis.
Sripramong (GP)	Circuit performance specifications and/or behaviour characteristic. Also requires an embryonic circuit to be provided.	Depends on difficulty of specifying or generating specifications or behaviour of required circuit. However, this system is aimed primarily at CMOS amplifiers.	Translation from functional to structural abstraction domains takes place. This is synthesis.

Table 3.1: Level of Design Automation Achieved by Reviewed SA & GA Methods

Method/ Authors	**Input Requirements**	**Setup Effort**	**Automation Achieved**
BLADES	User provides circuit constraints to expert system. However, a selection of circuit & sub-circuit topologies will already exist in the systems knowledge base.	Minimal, if the knowledge base is already populated (user must then only answer expert systems questions).	Translation across abstraction domains only takes place at a high level. This is a limited form of synthesis.
OASYS	Circuit performance specification. However, a selection of fixed circuit topologies will already be known to the system.	Minimal, if the knowledge base is already populated	Translation across abstraction domains only takes place at a high level. This is a limited form of synthesis.
VASE	VHDL-AMS circuit specification. However, a selection of fixed circuit topologies will already be known to the system.	Depends on complexity of behaviour of required circuit.	Translation across abstraction domains only takes place at a high level. This is a limited form of synthesis.
Kazmierski	VHDL-AMS circuit specification. However, a selection of fixed circuit topologies will already be known to the system.	Depends on complexity of behaviour of required circuit.	Translation across abstraction domains only takes place at a high level. This is a limited form of synthesis.

Table 3.2: Level of Design Automation Achieved by Reviewed Miscellaneous Methods

While this is a good definition of automation in some respects,[1] it leaves out some of the other important benefits of automisation. Removing or reducing human design effort usually does not *only* reduce design time. Automating analogue design would also ideally allow less experienced, or even totally inexperienced designers in the field of analogue design to produce complex analogue circuits. This would mean 'designing', or specifying analogue circuits at the behavioural level which brings us back to the concept of synthesis as a translation process.

The user of the ASTRX/OBLX system is required to input an unsized circuit topology, the corresponding constraints and a corresponding test harness. The system generates the component values and sizes. While the circuit constraints would necessarily specify behaviour in some way, the user must also provide a structural representation of the circuit. Both the input and output, then, are *structural* in nature. The system performs no translation between abstraction domains. It is an optimisation system. However, the creators of the ASTRX/OBLX system emphatically describe it as a synthesis system, and state a clear distinction between synthesis and optimisation (quote below is referring to previous attempts at analogue synthesis):

> *The key hurdle that has not been overcome to make this transition from* ***optimisation*** *to* ***synthesis*** *is that optimisation requires a good initial starting point to find an excellent answer, while synthesis requires no special starting point information.*[44]

In the context of this system, *'good initial starting point'* means providing a good initial guess of the component values and sizes. This definition has no concept of moving from behavioural input to structural output, merely how close the initial trial values of components must be to a good answer. Both the input and output of this system are transistor level netlists, even if the input netlist is not sized. While this system does provide some of level design automation, and indeed would appear to be quite an effective optimisation system (since it can start from a poor initial guess) the user must still provide *a lot* of design information.

The system reported by Zebulum offers a similar level of design automation. Based on a genetic algorithm, the *user* is only required to input circuit constraints. However, any given

[1] One the primary reasons for automating a task is to increase the speed with which it is done.

implementation of this system is restricted to only ever being able to deal with the topology determined by the GAs encoding method. The circuit topology is implicitly input into this system via its implementation.

Again, both the input and output are structural, so this is an optimisation system. However, it is explicitly described as such in the literature, which goes on to make a rather bold claim about the system's abilities during a case study of the optimisation of an OpAmp (emphasis added):

> *The GA arrived at these design strategies* ***without any kind of previous design knowledge being supplied to the system****, thus illustrating the GA's potential to rediscover human design rules.*[58]

While it is true that no previous design knowledge about component sizes or design rules had been supplied to the system,[2] the circuit topology represents a very significant amount of design knowledge.

The same is not true of the other genetic algorithm based systems reported by Koza and Sripramong. The user supplies essentially no structural information at all about the desired circuit, other than an embryonic circuit. While it is possible for the user to add a limited amount of structural information into the circuit embryo, this is certainly not necessary. The only input requirements for both systems are the embryo, allowable component types and values, and behavioural circuit characteristics. The topology of the output circuit can be generated completely automatically.

As the system reported by Sripramong is focused primarily on the synthesis of CMOS amplifiers, the required constraints are typical parameters of an amplifier such as gain-bandwidth product, slew rate, etc. The preparatory effort is therefore very low for amplifier generation.

Koza's system, however, is designed to be much more generic and therefore the generation of appropriate constraints or target characteristics may require more effort depending on the

[2] Other than maximum and minimum allowable component values, which are again implicit to the systems implementation.

desired type of circuit. Both of these systems accept behavioural input and produce structural output, they are analogue circuit synthesis tools in the truest sense. They both require relatively little effort to setup.

The BLADES and OASYS systems are quite similar to Zebulum's system in their input requirements. The user is required only to input circuit specifications. In the case of BLADES, the user must answer questions posed by an expert system. The output from both systems is a structural netlist. However, both systems rely on a detailed knowledge base of building blocks, defined at the transistor level. From the user's point of view, these systems do indeed perform translation. The knowledge base is also effectively an input into the system. Any topology generation performed by these systems only takes place at a very high level. Connections between 'black box' building blocks are formed, these blocks must be later replaced with transistors from the knowledge base. The majority of the structure in the generated circuit ultimately needs to be input into this system. Some generation of topology may occur for some circuits, however the detailed topology is not generated. Therefore, both of these systems offer a limited form of synthesis.

A similar level of synthesis is offered by the HDL based systems VASE, and as reported by Kazmierski. Both systems take a VHDL-AMS description of the circuit as an input. Only a limited subset of VHDL-AMS may be used in either system. After the source HDL code has been read in and compiled, the resulting parse tree of the input code is used to derive a high level topology of interconnected black boxes that perform particular functions. These black boxes are ultimately mapped onto predefined transistor level netlists. The input, then, is representative of circuit behaviour and will implicitly contain circuit constraints. Like the BLADES and OASYS systems, VASE and Kazmierski's system perform a limited form of synthesis.

3.2.3 Reported QOR

The quality of results generated by the methods reported in the literature are obviously also central to their worth. However, the QOR is in many ways linked to the level of design automation that a method offers. Automating analogue circuit design is difficult, and it is reasonable to expect that the greater the level of design automation achieved, the lower the QOR will be.

It may be acceptable to trade automation for QOR *to an extent*, but for any synthesis system to be usable the circuit designs produced must still be *good enough* for their intended purpose. Extremely high performance circuits are not always required.

Method/Authors	Circuits Attempted	Quality of Results	Comments
SA Methods			
ASTRX/OBLX	OTA, Simple OTA, Comparator	Good quality results which met or exceeded human designs.	Synthesised circuits were based on a user supplied topology.
GA Methods			
Zebulum	CMOS OpAmp, Class A Amp, Simple & Cascade OTA.	All generated circuits have good characteristics, most close to comparable human designs.	Evolved circuits used a human designed topology.
Koza (GP)	Lowpass filter, crossover filter, analogue computational circuit.	The evolved filters were based on recognisable topologies and had performance similar to human designed circuits.	The evolved circuits tended to be quite large and consist of many components.
Sripramong (GP)	Operational Amplifier	Synthesised circuits met required specifications.	Evolved topologies were large, although were largely free of redundant components.
Misc Methods			
BLADES	Operational amplifier.	Systems were comparable to human designed circuits.	Synthesised circuits were comprised of human design subtopologies.
OASYS	Simple 2-stage OpAmp, Cascode 2-stage OpAmp, OTA	Broadly satisfied design requirements.	Synthesised circuits were comprised of human design subtopologies.
VASE	'Telephone set' of audio transmitters & receivers.	Synthesised circuits met required specifications.	Topologies varied at high-level (made up of supplied subcircuit topologies).
Kazmierski	Colpitts Oscillator, Active 4$^{\text{th}}$-order low pass filter.	Synthesised circuits met required specifications.	Topologies varied at high-level (made up of supplied subcircuit topologies).

Table 3.3: Quality of Results Achieved by Reviewed Methods

Almost all of the analogue synthesis systems reported in the literature report are presented as having good QOR. However, the metrics used to measure a circuit's performance vary from system to system. This is due in part to the differences in the systems themselves. For example, both Zebulum and Sripramong use very detailed fitness functions which measure many aspects of the operational amplifiers they are evolving, such as the gain-bandwidth product, DC offset and power dissipation. Both of these systems are designed specifically to synthesise operational amplifiers. The system implemented by Koza uses more 'traditional' shape-fitting, circuit response based fitness function, and might typically take a frequency response as a target. This helps to achieve a much higher level of generality but it also means that fewer aspects of the function or implementation are optimised.

This does not prevent the circuits produced by Koza's system being subjected to the same set of measures used by Zebulum, but if these constraints have played no part whatsoever in the synthesis of that circuit it would be quite unreasonable to expect the system to have satisfied them. The reported QOR is usually stated in terms of what the various synthesis systems were asked to produce.

Of the three GA based systems reviewed here, Zebulum and Sripramong produced the circuits most likely to be useful in an actual, real-world application. Koza's very 'open' system (open in terms of generality) produced large circuits which contained many redundant and poorly connected components. The circuits from Zebulum's very 'closed' system were based on a human designed topology and would in general be considered better quality.

The system implemented by Sripramong is, due to its implementation as a GP-based GA, a very general, open system. However the way in which it is used is rather closed. It has the ability to produce novel, arbitrary circuit topologies but is actually intended to synthesise operational amplifiers. This is reflected in its fitness function which measures OpAmp specific circuit characteristics. There are essentially two sets of results from this system - circuits which are very topologically unconstrained, and those which have been given a good 'starting point' with a circuit embryo that contains common OpAmp subcircuit structures. The topologically unconstrained circuits were more like those produced by Koza's, in that they were generally very big circuits with a high component count. They contained few or no redundant components as this system actively removes these. The circuits based on a good starting point were much more like those produced by Zebulum's GA.

The results reported for the ASTRX/OBLX system are presented in terms comparing the accuracy of the solution (how close the final circuit is to the stated requirements) to the time taken to produce the circuit. The time taken is a combination of computational time and preparatory effort. A set of results is also presented which compares the results of the system to manual designs. The system is reported to produce good results in comparison to the previously published results from other synthesis systems.

The other four reviewed methods, BLADES, OASYS, VASE and Kazmierski, all produced results somewhere between the topologically entirely constrained or almost entirely unconstrained systems. They were made up of building blocks of set, known good analogue topologies. However, the level of topology at which these building blocks themselves are interconnected was broadly unconstrained in these systems.

In general, the more ‘closed’ the system (the lower the generality), whether in terms of how constrained the topology is, the number or type of required circuit characteristics which can be specified or anything else which may constrain the resulting circuit, the higher or more predictable the quality of results. The same is true of level of design automation offered - the lower this is, the higher the quality of results tend to be.

3.2.4 Ability To Discover New or Novel Circuits

The scope, or *generality* of the methods reported in the literature varies enormously. Like QOR, the generality of a synthesis system may also be linked to the level of design automation offered. The range of circuits a synthesis system can cope with may have a direct bearing on its usefulness. Some of the methods reported are focused on only one type of circuit, whereas others claim to be able to synthesise any type of circuit that the user requests. Moreover, some systems are claimed to have the potential to generate completely new circuit architectures. Others are constrained by their implementation to use known good topologies. The greater the generality of a synthesis system, the less constrained it is likely to be, and therefore is more likely to have a lower QOR. A summary of the generality of the reviewed systems is given in table 3.4.

There is a strong relationship in the reviewed methods between the level of design automation offered and the generality of the system. Those that used fixed topologies, such as Zebulum’s system, obviously possess very low generality. The implementation of the system would have to be changed to allow different topologies to be tackled. ASTRX/OBLX is similar in that it is restricted to only ever produce circuits that use the user-supplied topology, although far less effort is involved in applying the tool to a different circuit type. These are both optimisation systems and offer both quite low design automation and generality.

Most of the reviewed systems, such as BLADES, VASE, Kazmierski and OASYS all offer limited forms of synthesis. However, they do have the ability, albeit limited, to vary the circuit topology based on user requirements. They can all produce a range of very different types of circuits. Ultimately, though, they are still very dependent on predefined topologies of building blocks being available to them.

Those systems with the highest generality are those which offer the highest level of design automation, those which can be regarded as performing full synthesis and have completely free reign to produce *any* circuit topology. Koza's GP system is an example of this. The other GP-based system reviewed here, by Sripramong also has the potential to evolve any type of circuit although this system is actually targeted specifically at CMOS amplifiers. It has particular specialisations to assist in this, such as a circuit constructing function which creates an embryonic gain stage. Nevertheless, it could easily be re-targeted to other circuit types.

Any system which has complete freedom to configure topology at the component level will not only typically offer a high level of design automation, but it also has the potential to generate a completely new circuit topology.

The question that naturally arises from this is whether the ability to generate novel topologies is really that important. In many cases, it is not, especially for typical engineering applications. Circuit topologies exist that offer good solutions to the majority of engineering problems. A system based on these predefined topologies would still potentially be useful in many situations. However, this also makes them inherently limited in some way. Not only do these systems need to be preprogrammed with a vast library of known topologies to really give them a high level of generality, but they would only ever be able to produce solutions as good as the best topologies within that library.

Method/Authors	Restrictions Determined by Input	Comments
SA Methods		
ASTRX/OBLX	Topology is fixed by user	Method can only explore variations in component values, system cannot produce novel topology.
GA Methods		
Zebulum	Topology is fixed by GA implementation	Method can only explore variations in component values, system cannot produce novel topology.
Koza (GP)	Topology is almost completely unrestricted	Only restrictions are determined by user configuration. System is capable of producing any topology.
Sripramong (GP)	Topology is almost completely unrestricted	Only restrictions are determined by user configuration. System is capable of producing any topology.
Misc Methods		
BLADES	Topology can be varied to an extent	Ultimately, topology is dependent on embedded knowledge & system cannot produce novel topology.
OASYS	Topology can be varied to an extent	Ultimately, topology is dependent on embedded knowledge & system cannot produce novel topology.
VASE	Topology is dependent on structure of input VHDL-AMS code	Ultimately, topology can only be novel at high-level, with subcircuits using known topologies.
Kazmierski	Topology is dependent on structure of input VHDL-AMS code	Ultimately, topology can only be novel at high-level, with subcircuits using known topologies.

Table 3.4: Ability of Reviewed Methods to Discover Novel Circuits

Unconstrained Evolutionary Electronics

The synthesis of completely novel circuits is investigated by Thompson [53][52][51]. The motivation of this research is not so much to produce a synthesis system that can be used in a real design flow, but to investigate the potential of GAs to produce completely new circuit topologies not previously seen or indeed even conceivable by human designers. Thompson states three hypotheses which are then investigated:

> 1. *Conventional design methods can only work within constrained regions of design space. Most of the whole design space is never considered.*
> 2. *Evolutionary algorithms can explore some of the regions in design space that are beyond the scope of conventional methods. In principle, this raises the possibility that designs can be found that are in some sense better.*
> 3. *Evolutionary algorithms* ***in practice*** *can produce designs that are beyond the scope of conventional methods and are, in some sense, better.*
>
> [53]

Thompson makes use of *intrinsic evolution* [58]. This is rather different to the GA systems already examined which come under the heading of *extrinsic evolution*. Intrinsic evolution involves the measurement of candidate circuits by measuring *real* circuits rather than running a simulation. This is usually achieved via the use of some kind of reconfigurable circuit such as an FPGA or Field Programmable Analogue Array (FPAA). A circuit configuration is downloaded onto the reconfigurable device, the output of which is measured and fed back to the GA.

The synthesis of completely novel analogue and digital circuits is investigated. However, Thompson also advocates eliminating the distinction between analogue and digital circuits in order to increase the size of the design space. Choosing either a specifically analogue or digital circuit as the solution to a problem inherently limits the range of circuits which may be used.

Thompson goes on to show data to support the stated hypotheses. It is difficult to explain how some of the evolved circuits work, and they clearly would not have been produced by human designers following conventional design methods.

3.2.5 Circuit Evaluators

Almost all the synthesis systems in the literature employ some means to evaluate circuit performance. Most systems are iterative, and use some kind of loop which feeds back data about circuit performance to the synthesis engine. Therefore these circuit evaluators form an inherent part of the system itself. The most common evaluator employed is SPICE or a SPICE variant, however the literature reports some alternative ways of analysing circuits, designed primarily to reduce computational effort. The result is almost always less accurate, although the proponents of such techniques claim this is not an issue for the circumstances in which they are used.

Method/Authors	Circuit Evaluators / Analysers	Comments
SA Methods		
ASTRX/OBLX	Asymptotic Waveform Evaluation, Relaxed DC formulation & Encapsulated Device Evaluators	Much emphasis has been placed on speed, in particular being faster than SPICE
GA Methods		
Zebulum	SMASH	SPICE-like simulator
Koza (GP)	SPICE	Author had source code access, likely to have been Berkeley SPICE
Sripramong (GP)	PSPICE	Commercial SPICE simulator
Misc Methods		
BLADES	ADVICE	SPICE-like simulator developed for internal use at AT&T Bell Labs
OASYS	Integral to synthesis system	Embedded knowledge of each topology and sub-circuit includes relevant method of evaluation
VASE	Analog Performance Estimator (APE)	Is an integral part of synthesis system and plays a part in sizing transistors
Kazmierski	HSPICE	Commercial SPICE simulator

Table 3.5: Circuit Evaluation & Measurement Strategies of Reviewed Methods

The majority of the reviewed systems use a SPICE-based simulator, as summarised in table 3.5. SPICE is sometimes described as a *detailed* circuit simulator, and it simulates circuits at the lowest possible level using extremely detailed device models. It always analyses the entire circuit and is widely accepted as producing the most accurate and reliable circuit simulations.

There are, however, certain issues with using SPICE in an iterative, trial-and-error algorithm. Because SPICE performs such a complete circuit analysis there is a computational penalty to pay. Peforming one or two analyses at a time, as might be carried out by a human analogue circuit designer is generally manageable, but iterative synthesis algorithms will usually require hundreds if not thousands of circuit analyses in a given synthesis run. The accuracy of SPICE is not in question, it is simply the run time required by these simulations that is at issue. However there are other approaches to analysing a circuit that have been used in experimental analogue synthesis systems, the primary motivation being to reduce required the computational effort. The notable exceptions are ASTRX/OBLX, VASE and OASYS.

The ASTRX/OBLX system is a particularly interesting case as it uses different approaches to solve different aspects of circuit analysis. *Asymptotic Waveform Evaluation* (AWE)[47][45] is way of approximating a circuit response by matching initial boundary conditions and the first $2q - 1$ moments of the exact response to a lower order q-pole model. It does not perform as complete or as exact an analysis as SPICE, and trades accuracy for reduced computational effort. In ASTRX/OBLX it is used to perform AC analysis. This synthesis system also employs a novel technique to calculate the DC bias point of the circuit which the authors call *relaxed-DC formulation*. The user must include a bias circuit as part of the input to the system, as a result the cost function that system optimises for, $C(\underline{x})$, contains DC voltage variables. The DC bias point of the circuit is solved analytically (SPICE would find the bias point numerically) and as a result is much faster.

A different approach is taken by the *Analog Performance Estimator* (APE) used by the VASE system. Rather than performing an actual simulation, it is designed to produce an estimate of specific performance parameters, such as the open loop gain of a system. It works hierarchically, with a total of four levels. At the lowest level it uses SPICE CMOS transistor models. The next level of abstraction consists of equations that describe basic analogue components, such as current mirrors and differential amplifiers. The level above that consists of op-amp models, and finally the highest level of abstraction is concerned with so-called *analogue modules*. It is made up of a library of circuits such as comparators, filters, DACs, etc. These circuits are constructed of objects from the three lower abstraction levels: op-amps, current sources, transistors, resistors, etc.

When the APE is invoked, it is given a netlist of these analogue modules and a set of system requirements. Working in a top-down fashion, the set of requirements is decomposed and passed down the hierarchy, with each level getting its requirements from its parent. At the lowest level, the transistors are sized based on these requirements and then performance estimates are generated. Now working in a bottom-up fashion, these estimates are propagated upwards, with each level using its performance equation and estimates from the level below to generate its own performance estimates. Finally, when this process reaches the top level, all the necessary performance parameters have been estimated and each primitive component has been sized.

The APE methodology, then, is not just a way to evaluate aspects of circuit performance. Because it also performs component sizing, it plays a part in circuit synthesis, it is an integral part of the VASE system. The literature reports that the measurements made by APE are reasonably accurate when compared to measurements taken from SPICE simulations [41].

OASYS does not employ a circuit evaluator that exists as a distinct, separate entity as most of the other methods do. This system has attempted to codify human design knowledge and experience, and attacks the problem hierarchically in a top-down fashion. A suitable topology is selected from a number of predefined alternatives, all of which consist of a netlist of sub-topologies (sub-blocks). The user provided constraints are *translated* to lower level blocks, until finally the transistor level is reached. This process of translation, and also transistor sizing relies on a number of design rules associated with each sub-block. Once the transistors in a given sub-block have been sized, there are rules to determine performance parameters from the sized transistors and other components. These are then compared to relevant constraints. The developing circuit can then be continuously checked at each stage to make sure it meets the performance requirements. If any performance parameters are found which violate the constraints, the system can back-track and attempt to re-design the sub-block.

Chapter 4

GENETIC ALGORITHMS

Essentially, evolutionary methods seek the solution to a problem through the application of principles and ideas derived from the process of biological evolution. Evolutionary methods are search algorithms, a characteristic property of them is the fact that they are a *blind search*. They do not rely on gradient data or any knowledge of 'nearby' solutions in the search space, nor are other solutions sampled randomly. Rather, they are driven by the evolutionary processes inspired by nature. Evolutionary methods may be applied to a considerable variety of problems, although here the only application considered will be that of circuit optimisation and synthesis.

This chapter looks in detail at two forms of evolutionary method. The *Genetic Algorithm* is probably the most popular form of evolutionary method employed in academia, and the 'traditional' version is examined here, although many variants of this exist. A second evolutionary method, *Genetic Programming* is examined and is, indeed, a variant of the classic GA, although it is different enough to warrant a separate classification of its own. This is in part due to its application, as it is focused primarily on producing novel structures of *computer programs* although it ultimately finds applications in many different fields.

4.1 The Classic GA

The Genetic Algorithm (GA) was invented by John Holland [20] of Michigan University in the 1970s. This was not the first time that principles of natural evolution had been applied to difficult problems, but it was Holland that really popularised it.

An overview of the workings of a genetic algorithm and how it might typically be used is now presented. This is to quickly familiarise the reader with the subject and give a quick grounding for the more detailed explanation of the component parts of the algorithm that follow. This overview is presented in generic terms, and is not just focused on the problem of analogue circuit design.

The user usually provides a *target characteristic* which essentially tells the GA what to look for. The target characteristic is way of *stating the problem to be solved*; it is a way of stating what the requirements are of a satisfactory solution. When optimising an analogue circuit such as a filter, for example, the target characteristic might be the required cut off frequency, the pass band attenuation or maybe a collection of several parameters.

Genetic algorithms maintain a *population* of candidate solutions to a specified problem. These candidate solutions are typically *encoded* in a form suitable for representing the problem solution. Each individual in this population is measured in some way to determine its *fitness score* which is a result of the *fitness function* employed. This score is a measure of how well a particular individual solves the specified problem.

The population is subject to processing by a number of *genetic operators*, of which the classic GA defines three types. The *selection operator* picks a number of individuals based on their fitness score, to go through to the next *generation*. Some of these individuals are then picked to undergo *crossover*, which is another of the genetic operators. Crossover is a key function of GAs, and is the primary means of directing the search process. Two individuals are picked and sectioned in some way. The resulting portions of each one are recombined with the other in order to produce two new offspring, both of which share characteristics of both parents. The offspring of two fit parents will inherit traits from both of them. If those traits are useful, the offspring will be even fitter. If not, the unfit offspring will eventually be eliminated from the population. Crossover therefore is the primary way of guiding the blind search.

After crossover, the final operator is applied. The *mutation operator* randomly picks individuals and then randomly changes part of them. Mutation is important as it can introduce 'fresh blood' into the population. A flow diagram of the classic GA is shown in figure 4.1.

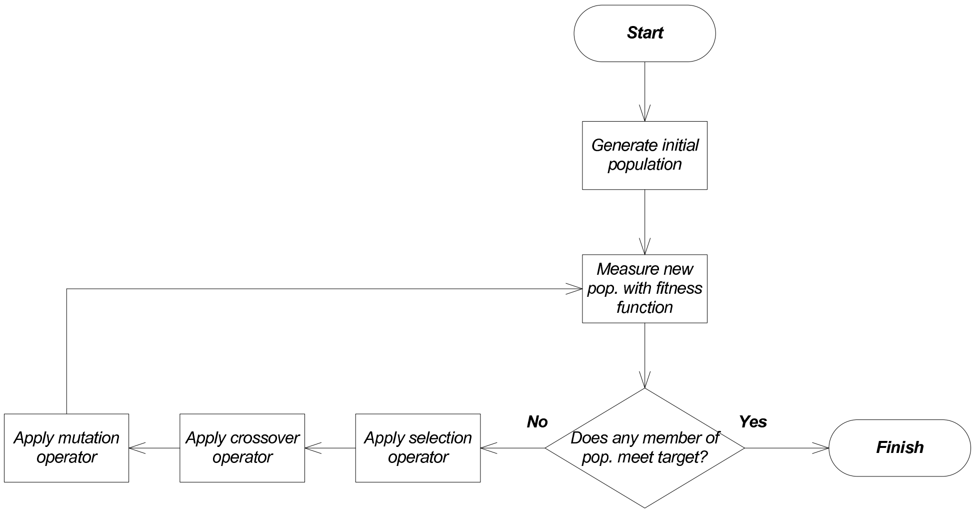

Start
Generate initial population
Measure new pop. with fitness function
Does any member of pop. meet target?
No
Yes
Finish
Apply selection operator
Apply crossover operator
Apply mutation operator

Figure 4.1: Classic Genetic Algorithm

The result of applying the three operators is a new population. This will be made up primarily of either the fitter individuals of the original population, or the offspring of crossing over primarily fitter individuals. Some members of the new population will have been mutated.

This loop is then repeated, and the aim is for overall fitness of each generation should improve. This continues until either a maximum number of generations has been executed or until an individual has been found that provides a satisfactory solution to the original problem. The initial population is randomly generated, although it may be subject to certain constraints depending upon the application.

4.2 Encoding Schemes

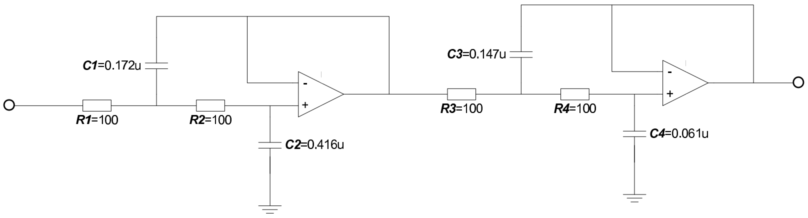
The choice and design of encoding scheme used by a GA is of crucial importance and can have a dramatic impact on the effectiveness and behaviour of the algorithm. Due to the way that GAs work, it is often not possible to represent candidate solutions to the target problem in their most natural form. A variety of genetic operators must be applied to the population, and this mandates that a solution representation is used that is compatible with these operators.

The crossover operator in particular imposes certain requirements. A encoding structure must be used that may be cut at any arbitrary point, but that when spliced with a portion of another structure will still represent a meaningful candidate solution. Such schemes may be classified as either fixed or variable in length or size.

Holland's original GA made use of fixed length, binary strings. In order to apply the GA to a specific problem, a mapping would need to be formulated in order to represent the problem as a binary string.

The *Schema Theory*, defined in section 4.5 suggests that binary representation will provide the highest number of schemata (defined in section 4.3). However, this binary representation is often far from the most natural or useful way to encode solutions. A much more direct mapping is often achieved through the use of integer or real-numbered strings (figure 4.2).

The number of different symbols used in an encoding scheme is known as the *cardinality of*



R1	C1	R2	C2	R3	C3	R4	C4
100	0.172u	100	0.416u	100	0.147u	100	0.061u

Figure 4.2: An example of how circuit may be represented by a real valued fixed length string

the alphabet.

4.3 Schemata

Now that a description of typical string encoding schemes has been given, the concept of *schemata* can now be defined. This is needed so that the *schema theory* can be introduced in section 4.5. The schema theorem helps to explain the behaviour of GAs and leads to some important results.

The size of the alphabet cardinality of the encoding scheme has a direct impact on the size of the search space. For a cardinality K, and string length L, the total number of points that exist in the search space is given by K^L [59]. A *schema* is a collection of a subset of these points, and may be represented by the addition of another symbol to the alphabet, $*$ which means '*don't care*'. A schema, then, is a pattern which can match a number of points in the search space. Table 4.1 shows example schemata for a binary encoding scheme.

Schema	Corresponding Points
$0\ 1\ *$	0 1 0, 0 1 1
$*\ *\ 0$	0 0 0, 0 1 0, 1 0 0, 1 1 0
$*\ 0\ *$	0 0 0, 1 0 0, 0 0 1, 1 0 1

Table 4.1: Example Schemata For Binary Encoding, $L = 3$

Any given point may belong to multiple schemata. Given that the size of the search space is given by K^L, the number of schemata for a given cardinality is $(K + 1)^L$ as there is an extra 'don't care' symbol.

The *order* of a schema, $O(H)$ is defined by the number of alphabet symbols it contains. Equations 4.1 to 4.3 demonstrate this.

$$H = 1\ 1\ 1\ 0\ *\ 1\ *\ * \Longrightarrow O(H) = 5 \tag{4.1}$$

$$H = *\ *\ 0\ 0\ *\ *\ 1\ * \Longrightarrow O(H) = 3 \tag{4.2}$$

$$H = *\ *\ *\ *\ 0\ *\ *\ * \Longrightarrow O(H) = 1 \tag{4.3}$$

The defined length of a schema, $\ell(H)$ is defined by the distance between the first and last defined symbols ($I_{end} - I_{start}$), as illustrated by equations 4.4 to 4.6.

$$H = 1\ 1\ 1\ 0\ *\ 1\ *\ * \Longrightarrow \ell(H) = 5 \tag{4.4}$$

$$H = *\ *\ 0\ 0\ *\ *\ 1\ * \Longrightarrow \ell(H) = 4 \tag{4.5}$$

$$H = *\ *\ *\ *\ 0\ *\ *\ * \Longrightarrow \ell(H) = 0 \tag{4.6}$$

4.4 Genetic Operators

4.4.1 Selection

The selection operator determines which individuals are permitted to go through to the next generation, and therefore which are available to be picked for crossover. Selection in GAs is probabilistic. Holland's original selection operator was quite simple. The probability of selecting a particular individual, p_i simply depends on the fitness of the individual f_i, and the total fitness of the population f_T, as shown in equation 4.7.

$$p_i = \frac{f_i}{f_T} \tag{4.7}$$

Depending upon the method used to produce the fitness score, f_i may need to be adjusted or normalised in some way. For example, higher fitness may indicate either a more or less fit individual.

The probability p_s that the schema H will pass through the selection operator is given by equation 4.8, where $f(H)$ is the average fitness of the members of schema H, and $\overline{f_m}$ is the average fitness of the whole population.

$$p_s = \frac{f(H)}{\overline{f_m}} \tag{4.8}$$

4.4.2 Crossover

The crossover operation is central to how the GA works. It provides the primary means of guidance to the GA in the search space. Crossover provides a means of creating new *offspring* solutions from a pair of *parent* solutions. The parent solutions are sectioned, and the resulting fragments are then recombined to create new offspring. The hope is that the new solutions will contain useful characteristics of both parents and be a better solution than either. The cutting point is randomly picked. The first section of parent A is then spliced to the second section of parent B to create the first new string. Vice versa for the second new string. This is illustrated in figure 4.3, using simple multiplexer Look-Up-Tables (LUTs) as an example. These circuits may be found as part of Configurable Logic Blocks (CLBs) on FPGAs, and allow logic functions to be programmed into what are effectively 'blank' logic gates.

It is important that the cutting point is the same for both strings. This is required to preserve the length of the strings. In this basic crossover operation, two parents are always required and two new offspring are always created.

Crossover is applied to a set of parent solutions probabilistically, the chance that crossover will be applied to parents is known as the *crossover rate*. This parameter is usually set to high

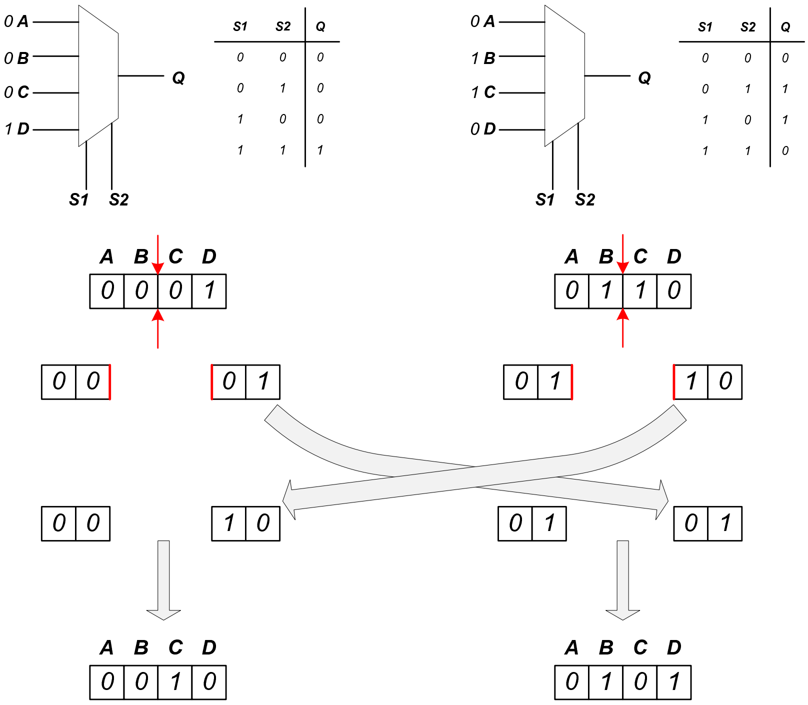


Figure 4.3: An example of crossover of simple 4-bit binary strings.

levels, 60% isn't unusual [58] [26].

The probability p_c that the schema H will pass through the crossover operator unchanged is given by equation 4.9, where r_c is the crossover rate, $\ell(H)$ is the defined length of the schema, and L is the length of the string. It is simply the ratio of the defined schema length to the string length, multiplied by the crossover rate. Short and low order schemata are more likely to pass through the crossover operator unchanged.

$$p_c = 1 - r_c \cdot \frac{\ell(H)}{L-1} \tag{4.9}$$

4.4.3 Mutation

While the crossover operator tends to guide the GA through the solution space, the mutation operator allows the search to be widened slightly. Simply put, the operator randomly changes randomly chosen parts of random strings. It is applied after crossover and plays an important role in that it injects new 'blood' or 'genetic material' into the population. This allows the GA to test out new solutions.

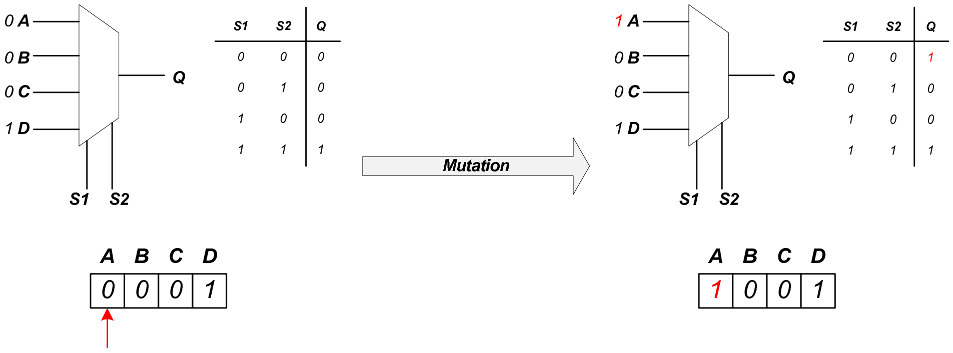


Figure 4.4: An example of mutation of simple 4-bit binary strings.

Figure 4.4 shows strings being mutated. It is usually applied at quite a low rate, often 5% or less. Too low, and particularly fit individuals can come to dominate the population and therefore the GA may get stuck in a local minimum. Too high, and the GA will start to behave like a random search.

The probability p_m that the schema H will propagate through the mutation operator unchanged is given by equation 4.10, where r_m is the mutation rate and $O(H)$ is the order of the schema. It is given by the probability that mutation *will not* happen $(1 - r_m)$ raised to the power of the schema order. The schema will remain intact if a 'don't care' symbol is picked for mutation rather than one of the defined symbols. As with crossover, short and low order schemata are more likely to pass through the mutation operator unchanged.

$$p_m = (1 - r_m)^{O(H)} \tag{4.10}$$

4.5 Schema Theory

The schema theory, also called the *fundamental theorem of genetic algorithms* was originally developed by John Holland [20] in an attempt to produce a precise mathematical model of genetic algorithms, it is the most accepted model. It assists in the understanding of how GAs work, and it leads to some important results. It attempts to answer the question: if a particular schema is present in a population, will it propagate to the next generation? However, it most certainly cannot explain all aspects of a GA's behaviour; it is most certainly not a complete model. GAs have many parameters, and the effect of varying them on its behaviour is extremely difficult to predict. Certainly, obtaining good parameters to solve a particular problem is often considered something of a black art.

The number of schema present in generation $g+1$ may be expressed as in equation 4.11, where $m(H,g)$ is the number of instances of schema H at generation g, and p_s, p_c and p_m are the probabilities of schema H passing unchanged through the selection, crossover and mutation operators respectively.

$$m(H,g+1) \geq m(H,g) \cdot p_s \cdot p_c \cdot p_m \tag{4.11}$$

$$m(H,g+1) \geq m(H,g) \cdot \frac{f(H)}{\overline{f_m}} \cdot \left[1 - r_c \cdot \frac{\ell(H)}{L-1}\right] \cdot (1-r_m)^{O(H)} \tag{4.12}$$

Replacing p_s, p_c and p_m with equations 4.8, 4.9 and 4.10 results in equation 4.12 which is the result derived by Holland [20].

Some important conclusions can be drawn from this. The first is that *short, low order schemata will exponentially increase or decrease depending upon their average fitness.* This is predominantly due to the second term in equation 4.12 which expresses how likely a particular member of a schema is to be selected for crossover. If the average fitness of the members of a schema is greater than the average fitness of the population, this term will be > 1, otherwise it will be < 1. The last two terms of the equation, those that express the probability of members of a schema surviving crossover and mutation, can only ever be ≤ 1. However this is still a very limited result due to the following factors, as described by Zebulum [58]:

- *It takes only into account the destructive behavior of the three GA operators, selection, crossover, and mutation. But, what about the role of these operators in building good schemata?*
- *This theorem suggests that we should use representations where good chromo-*

> *somes belong to short and low-order schemata. This guideline is usually not followed by GA users, because it is often difficult to have knowledge about how fit a schema will be before an extensive study of the particular representation is performed. More importantly, there are other effects of the genetic operators, as stated in the item above, that may hide the beneficial effects of this suggested compact representation.*

Another important result that can be derived from the field of schema theory is that the most efficient alphabet cardinality to use is $K = 2$. To expand on this, consider a problem which has at most 8 solutions. A binary representation ($K = 2$) would need 3 symbols ($L = 3$), therefore there would be $(2 + 1)^3 = 27$ schemata. At the other extreme, we could use an alphabet of 8 ($K = 8$) with a string just 1 symbol long ($L = 1$), resulting in $(8 + 1)^1 = 9$ schemata. Binary representation allows a greater number of schemata.

This is very important, as a greater number of schemata increase what is known as the *implicit parallelism* of GAs. As a GA works, it samples points in the solution space of the problem. While doing this it also implicitly processes several schemata.

However, the advantage of using a binary alphabet should not be overstated. The benefits of a binary alphabet have not been proved experimentally [58] and a binary alphabet does not provide for a natural encoding scheme for a great many problems, resulting in an overly complex mapping between representation and problem solution.

4.6 Alternative GA Implementations

There are many different ways to implement a GA. Almost every aspect of it can be tailored for a specific problem. However there are still some common elements to every GA. They all maintain a population of candidate solutions, and there are two essential genetic operators; crossover and selection. The actual form these take can vary although in GAs selection is probabilistic.

The parts of a GA that tend to vary the most across different implementations are the encoding scheme and fitness function, as it is these parts that are most concerned with the problem to be solved. In fact it usually a requirement that these are designed to suit the problem in question.

Different implementations may even have different operators. Mutation is common, although not essential, but it is possible to conceive new operators depending upon the problem to be solved and encoding structures used.

4.6.1 Alphabet Cardinalities

In practice, many GAs make use of strings of integer or real values. This often provides a more natural representation, and for this reason many GA users do not employ binary representation.

4.6.2 Variable Length Encoding Schemes

In implementations that closely follow this traditional GA, only solutions of a fixed size are sampled. Zebulum [58] introduces the concept of variable length strings, although to be more accurate it is a way of deactivating portions of the string. This is achieved by means of a second string, known as the *activation mask*, itself a binary string. The *main string* and the activation mask are the same length, they each contain the same number of symbols. Each bit in the activation mask controls whether the corresponding symbol in the main string expresses itself in the problem solution. Figure 4.5 illustrates the principle.

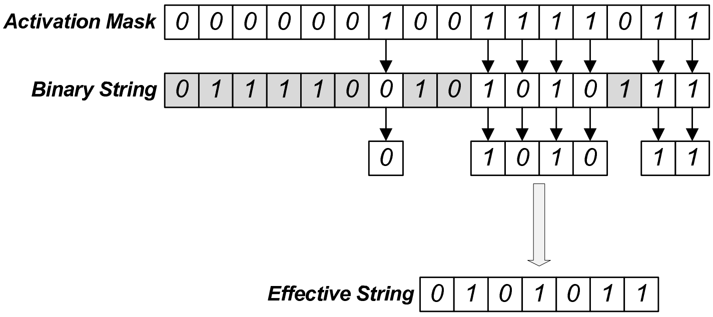


Figure 4.5: Variable Length Strings

Strings are not the only way to encode solutions, it is possible to conceive of other representations. Section 4.8 on Genetic Programming discusses some alternative methods to strings.

4.6.3 Crossover For Other Encoding Schemes

Single-point crossover is not the only way in which the operator may be applied. Other rules include *two-point* and *uniform* crossover [58]. Figure 4.6 illustrates two-point crossover. Two cutting points are picked, usually randomly resulting in 3 string fragments. Again, in fixed length encoding schemes both parents are cut at the same points. The middle fragments are then swapped between the two to create the two new offspring.

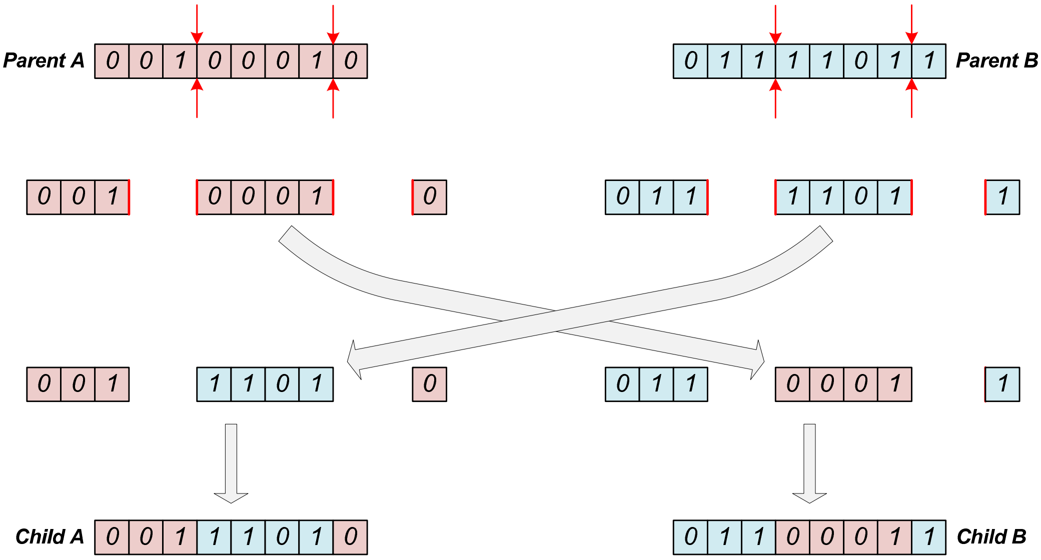


Figure 4.6: Two Point Crossover

Uniform crossover makes use of a binary template. This template is the same length as the parent strings, and contains a random binary pattern. The pattern determines which parent provides the symbol at each point in the string. The two offspring will then have a symbol from a different parent at each point in the string. Figure 4.7 shows an example of uniform crossover. One possible disadvantage with this method is that the random way in which the two parents are recombined means that useful structures are more likely to be destroyed. One and two-point crossover on the other hand tend to preserve entire fragments of each parent.

4.6.4 Selection Operators

The literature reports several implementations of the selection operator, and two of the more popular are presented here.

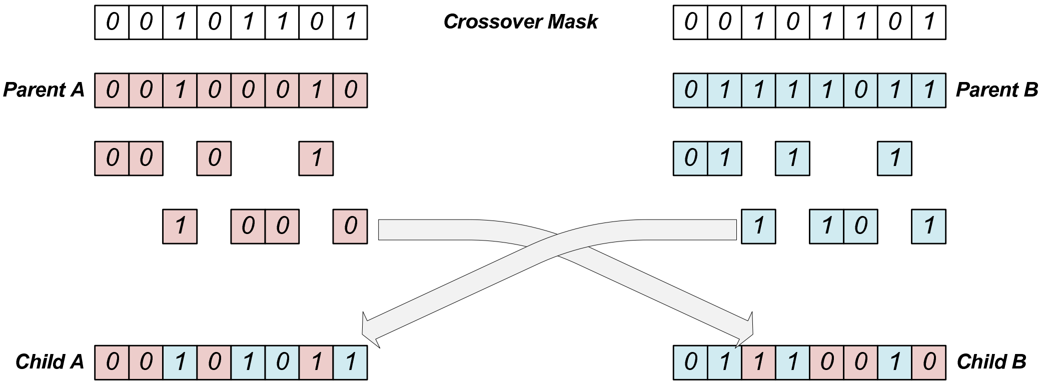


Figure 4.7: Uniform Crossover

Linear Rank

In linear rank selection [58], each individual is assigned a probability that is only indirectly based on its fitness. The individuals are sorted into order, and a probability is assigned to them found using equation 4.13, where N is their rank within the population. The function is shown graphically in figure 4.8. Two parameters η_{max} and η_{min} control the gradient of the function, and hence will produce a much bigger difference between the lowest and highest probabilities assigned. The parameters are subject to the following constraints: $\eta_{max} + \eta_{min} = 2$, where $\eta_{min} \geq 0$. Typical values are $\eta_{max} = 1.1$ and $\eta_{min} = 0.9$.

$$p_i = \frac{1}{N}(\eta_{max} - (\eta_{max} - \eta_{min})\frac{i-1}{N-1}) \tag{4.13}$$

Exponential Rank

Exponential rank selection [58] is similar to linear rank selection, in that the population is sorted into order of fitness. The probabilities then assigned to them are not derived from a linear function, but from a weighted exponential curve given by equation 4.14, shown graphically in figure 4.9.

$$p_i = \frac{c-1}{c^N - 1}c^{N-i} \tag{4.14}$$

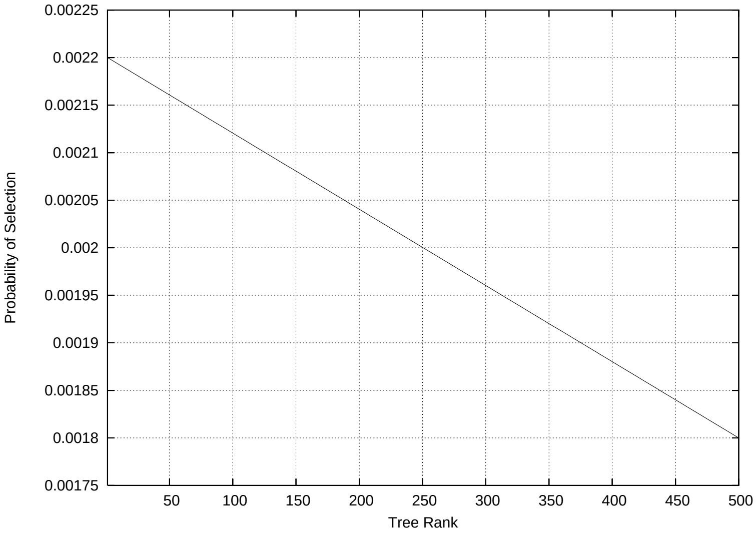


Figure 4.8: Linear Rank Selection $\eta_{max} = 1.1$ $\eta_{min} = 0.9$

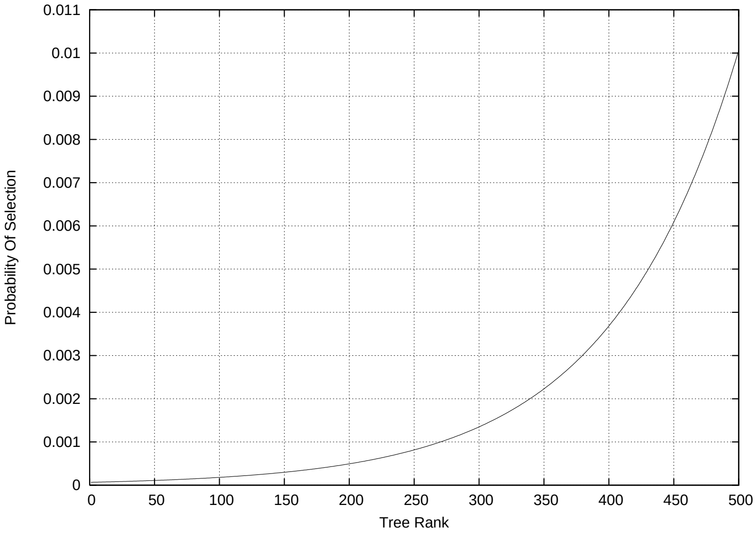


Figure 4.9: Exponential Rank Selection $c = 0.99$

4.7 Fitness Functions

Like the encoding scheme, the fitness function used by a GA is of critical importance. It provides the only means a GA has of 'seeing' the world, and it must provide a representative measure of each individual's ability to solve the stated problem. Without this, the GA is essentially without guidance as the selection, and by extension, crossover operators cannot work effectively.

It is therefore critical that the GA be able to measure the 'goodness' of each individual solution. The form this measure takes will be need to be designed in conjunction with the selection operator, however it is usual to use a single number as a fitness measure. Some way is needed of ranking the population in terms of an *absolute* fitness measure. Again, the specifics of this number are dependent on the GA application and selection operator. It may be an integer or real number and a lower value may represent either a worse or better score.

The fitness function will, in the case of circuit optimisation or synthesis, almost always require some sort of circuit simulator. In the case of analogue circuits, that will probably mean a variant of SPICE or something similar. In order to calculate a fitness measure, a translation process will need to take place which converts the individual's encoding into a form suitable for input into the simulator. This will typically be a netlist. The fitness function will need to read in the output from the simulation and compare it to the target characteristic in order to produce a single number according to a set of rules that have been designed to accommodate both the circuit type and simulation output format. The target characteristic is usually supplied in the same format as the simulation output for each candidate solution measurement.

It is not possible to describe a 'standard' fitness function as they are by necessity dependent on the application, circuit type and selection operator. However, many applications have used what will referred to in this text as a *shape fitting* fitness function. This typically involves supplying a circuit output waveform in the time domain, or frequency plot as the target characteristic. When each circuit is measured, a simulation is run which could be transient or frequency sweeps, as required. The target and simulated waveforms are then compared. Figure 4.10 demonstrates this, using the frequency response of a high-pass filter as an example.

The difference between each point in the target and trial traces are then compared. A simple way of producing a fitness score might then be to simply add up all of the absolute differences, as shown in equation 4.15. The fitness score is denoted by S_f, n is the number of points being compared, and $C(x)$ and $T(x)$ are the candidate and target point vectors respectively.

$$S_f = \sum_{x=1}^{n} |T(x) - C(x)| \tag{4.15}$$

A slightly modified version of this has been reported in the literature, where a vector of weighting factors w_x is used (equation 4.16).

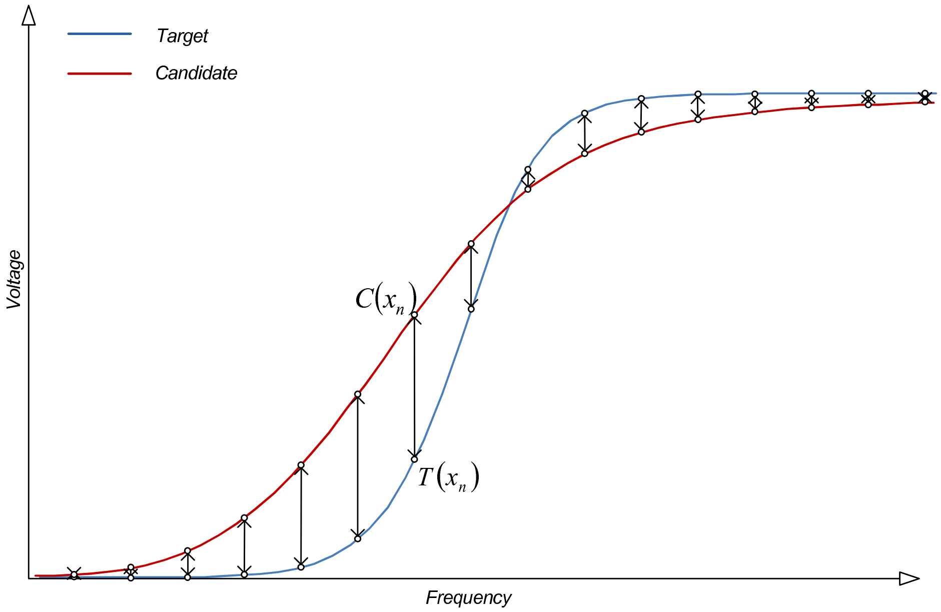


Figure 4.10: Shape-fitting fitness functions usually involve differencing the target and candidate circuit responses.

$$S_f = \sum_{x=1}^{n} w_x \cdot |T(x) - C(x)| \tag{4.16}$$

This is useful in comparing frequency sweeps, as more weight can be given to errors in the pass band, for example. Zebulum [58] reports that weight factors of 600 for the pass band, and 10 for the stop band worked well for a low pass brick-wall filter. If this approach is taken, much trial and error is often required in order to determine satisfactory values. The weights that Zebulum used may not work well for other GA implementations as there as so many other parameters that affect GA performance. There are many ways in which this approach may be implemented. For example, the difference between the target and candidate circuits may first be squared before the weights are applied.

4.8 Genetic Programming

Genetic Algorithms are a generic search technique that may be applied to many different types of problems, although they have been mainly used in the fields of science and engineering. One specific application that has been problematic for GAs is that of software engineering. John Koza developed a method of programming computers with GAs which he called *Genetic Programming* (GP), described in his 1992 book [26].

4.8.1 Parse Trees

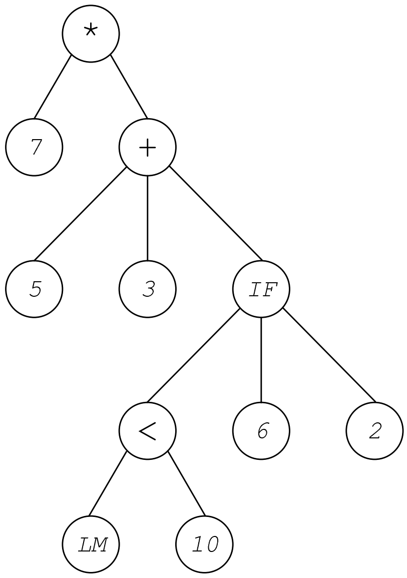
The reason that it is difficult to apply GAs to the problem of software production is that there is no obvious, natural way of encoding something as complex as a computer program into a string representation, whether using a binary alphabet or any other kind of alphabet. A computer program contains keywords, function calls and definitions, expressions and so on that all have strict syntactic rules that must be adhered to if the program is to make sense. As a result, a significant problem occurs during crossover as a way must be found of cutting a program at an arbitrary point and reassembling it without without violating these syntactic rules and introducing other errors.

Koza's solution to this problem was to use a much more sophisticated, and complex, encoding scheme. Rather than process *strings*, GP processes *parse trees* of programs. The LISP programming language was chosen for use in his GP system, for a number of reasons, although there are many languages which could be used in a GP implementation. Most program compilers or interpreters initially convert the program into an internal parse tree from. LISP provides easy access to its internal parse tree representation.

Figure 4.11 shows an example of a LISP parse tree and the corresponding LISP program. These parse trees are quite small and simple, but a significant advantage of this encoding scheme is that it is, by its very nature variable in length and size. The potential is there to evolve large and complex programs. This is also important as, like other problems, the necessary size of a solution may not be obvious a priori.

4.8.2 GP Crossover

The way in which genetic operators work is very dependent on the encoding scheme used. In GP, selection is carried out in the same manner as a more traditional GA. The only real difference with GP is the encoding scheme. Any of the typical selection operators and methods may be



```
(* 7 (+ 5 3 (IF (< LM 10) 6 2) ) )
```

Figure 4.11: A LISP program and its corresponding parse tree.

used here, or indeed any new ones that may be conceived depending on the application.

Crossover, however must obviously be very different for GP. Two parents are used in GP crossover, just as in traditional crossover operators. Cutting points are randomly picked between nodes on the tree, and the two resulting sub-trees from each parent are swapped. Figure 4.12 shows this process.

Crossover of tree structures has some important properties. The first is that because they are a naturally variable length representation, there is no requirement for the cutting point to be identical on both trees. In fact such a requirement would in some cases be overly restrictive. If crossover were applied to two parents of very different sizes, then obviously the smaller tree would dictate which points would be available for cutting. It is possible that in some cases it would not even be possible to find any meaningfully equivalent points on a pair of trees.

The second interesting property is that two identical parents will probably create different offspring. With traditional, fixed length strings, two identical parents could only ever create offspring identical to themselves as the two parents must both be cut at the same point. It is possible for two indentical parent trees to create identical offspring, but the same cutting point must be selected on both trees, which is unlikely, especially for larger trees.

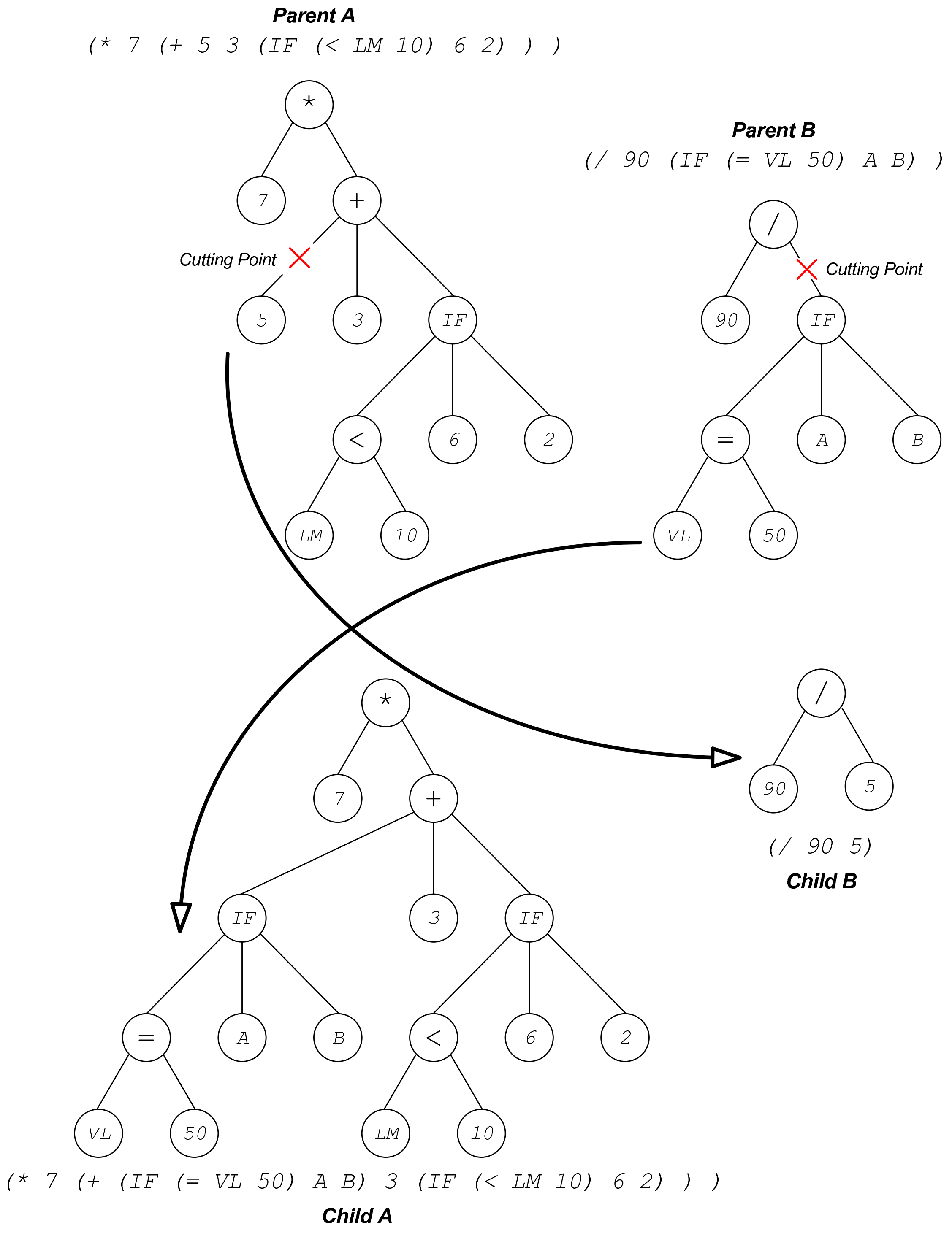


Figure 4.12: Genetic Programming crossover of two LISP programs.

Finally, crossover allows, and usually results in, offspring of a different size to their parents to be created. This allows easy exploration of different sized solutions in the search space, but it also has a disadvantage. The population can easily become dominated by very large trees and there is potential for the trees to grow unabated. Some way is needed to limit the size of the trees. Koza uses a parameter to specify a maximum tree depth in terms of the number of S-expressions created by crossover.

4.8.3 GP Mutation

Mutation is also different for genetic programming. A point on the tree is picked at random. Everything at and below this is removed completely and a new, randomly generated sub-tree is inserted in place of it.

It is important to note that Koza felt that the mutation operator was of relatively little use in GP, for two reasons. The first is that in GP, useful features of an individual are not necessarily associated with fixed positions within the structure. There are usually fewer functions and terminals within a GP tree than there are symbols in a comparable GA string, so it is relatively rare for an important feature to be lost from a population. As a result, mutation is less useful in restoring these lost features to the population.

The other reason is that in GP crossover has a similar effect to mutation when the two crossover points selected are both terminals of trees. As a result, if the mutation operator is at all useful for GP, the crossover operator already fulfills this need. Koza rarely used this operator in the GP-applied problems described in his book [26].

4.8.4 GP For Circuit Synthesis

Like GAs, GP can be applied to many different problems. Any problem, in fact, that a computer program may address. However, this thesis is concerned with circuit synthesis. Koza has applied his GP technique to this problem.

The approach taken involves generating a computer program of *circuit constructing functions*. These functions perform tasks such as inserting a new component or connecting two or more components in series or parallel. This program is then executed, the result is a circuit netlist. Koza made use of a *circuit embryo*, which is essentially a test harness for the circuit. Figure 4.13 shows a simple circuit embryo.

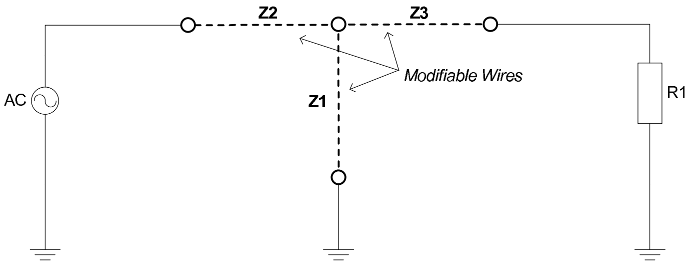


Figure 4.13: The circuit embryo serves as both a starting point and test rig for the evolving circuit.

Of particular importance are the *modifiable wires*. These wires are modified by the circuit constructing functions and a complete circuit is left as a result. The circuit embryo also identifies circuit inputs and outputs.

Chapter 5

SPICE SIMULATION & OTHER GA ISSUES

Genetic Algorithms are delicate things. Their performance and effectiveness is extremely sensitive to the many parameters that most GAs are controlled by and also to many other aspects of the way they have been implemented. Careful consideration must be given when designing a GA implementation. In some ways, this is more like an art than a science and there are no rules or even guidelines on how many parts of the GA should be configured for a given application. Choosing appropriate values for the many parameters that most GAs have will typically be a matter of trial and error.

This chapter looks at the most significant issues that can impact a GAs effectiveness, as far as applying them to the problem of analogue circuit synthesis is concerned. Issues concerned with the encoding scheme and fitness function are examined, as well as the problem of selecting appropriate parameter values. However, the bulk of this chapter is concerned with the use of SPICE, or SPICE-like circuit simulators as part of a GAs fitness function. There are a number of aspects of SPICE which make it quite unsuitable to be used in this way, however there is little alternative. The effect these issues have is discussed, as are ways to limit their impact.

5.1 Circuit Encoding Considerations

Designing a representation system for a given application can be a very complex task. There are two separate components to an encoding scheme: the structures manipulated by the GA and the mapping function between those structures and the actual solutions. It is possible for two very different encoding schemes to use identical structures. For example, two encoding schemes which both use integer strings could use very different mapping functions.

The encoding scheme used directly determines the size of the search space sampled by the GA. This is the most important property of an encoding scheme. All but the most trivial schemes will be capable of mapping to a vast number of possible solutions, but it is critical that the encoding scheme is capable of representing an acceptable solution to the problem. The user must know, or at least believe that this is the case. This must be the first consideration when designing the encoding scheme.

It is also crucial that the encoding scheme be capable of being processed by the GA, in other words it must be possible to apply the crossover and other genetic operators to the encoded solutions. The crossover operator must be designed in conjunction with the encoding scheme. This is not simply a case of picking a structure which can be cut and rejoined with other individuals, the way in which the mapping function works is just as important. Every new solution that is a result of crossover must make sense in the context of the problem, it must be possible to map that new individual to a valid solution.

Finally, some thought should also be given to the *complexity* of the mapping function. A simpler, more direct mapping function will obviously have less computational overhead when the GA is executed. However, more complex mapping functions often allow more complex solutions to be encoded.

5.2 The Problem With SPICE

The fitness function of a GA is as important as the encoding scheme used. It provides the GA with its only means of 'seeing the world', and so must provide an informative fitness score for each individual. It often needs to be tailored to the specific circuit type being evolved.

When applied to the synthesis or optimisation of analogue circuits, fitness measurement will almost certainly involve the use of a circuit simulator. There are a variety of analogue simulators available, but the SPICE simulator was the first of this type and is probably the most popular. There are many different versions of SPICE available, some are free and others are commercial. There are also other simulators which are not directly based on SPICE. However, all analogue simulators fundamentally work in the same way. At their core they are simultaneous equation solvers [24]. Solving systems of simultaneous equations is not always easy and they use certain numerical methods which provide solutions, or approximate solutions to these equations.

Even with a variety of these numerical methods available, it is not always possible to produce a solution for a given set of equations, and as a result analogue simulators cannot always simulate a given circuit. In other words, analogue simulations do not always *converge*. There are also numerous ways in which errors can creep into simulations even if they do converge. Therefore, there are two ways in which a GAs fitness function may not provide accurate feedback. When simulations are aborted the GA has no accurate information about how fit that individual is, and if the circuit does simulate the data produced may contain significant errors. Both of these situations are now examined. It should be noted that while this discussion is focused on the original, Berkley versions of SPICE, the same issues apply to at least some extent in all other SPICE versions and analogue circuit simulators.

5.2.1 SPICE Basics & Analysis Types

Even when applied to a specific area like analogue circuit synthesis and optimisation, there is a lot of scope for variation in how a GA is implemented. In particular, the fitness function will depend upon the type of circuit in question. The analysis type performed by the simulator will also depend on this.

Any analogue simulator worth its salt will offer a wide selection of circuit analysis types, the most common being transient, DC and AC sweeps and DC bias point analysis. This last one is important, as SPICE performs a DC bias point analysis as the initial stage of all of the other analysis types. DC and AC sweeps are perhaps the most common analysis types when the simulator is used as part of a GA fitness function, as usually some aspect of the circuit's characteristics are being measured. Transient simulation is less likely to prove useful for this application. However, any conceivable type of circuit analysis may potentially be used to measure a circuit's fitness.

Before important failure mechanisms in SPICE can be discussed, some background is needed on what SPICE is doing when it simulates a circuit. This will be presented in only enough detail to make sense of the following discussion and to place it in context.

As already stated, the SPICE engine is little more than a simultaneous equation solver. As an input netlist is read in, SPICE fills its internal *system matrices*. These matrices form the *system equation* which consists of a conductance matrix, a node voltage matrix and a branch current matrix as shown in figure 5.1.

The conductance matrix is square and has the same number of rows and columns as there are nodes in the circuit. Every circuit element that SPICE reads has a corresponding *element*

$$\begin{pmatrix} G_{11} & G_{12} & G_{13} & \ldots \\ G_{21} & G_{22} & \ldots & \\ G_{31} & \ldots & & \\ \ldots & & & \end{pmatrix} * \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ \ldots \end{pmatrix} = \begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ \ldots \end{pmatrix}$$

Figure 5.1: System equation of SPICE containing conductance, node voltage and branch current matrices.

template that defines conductance values that will be added to certain positions within the conductance array, which are determined by the circuit nodes the terminals of that element are connected to. After the input netlist has been read in, SPICE will have a system equation which completely describes the circuit.

The result is a set of equations and the same number of unknowns. SPICE must find a set of node voltages and branch currents which satisfy these equations. There are two ways in which SPICE can attempt to do this. If the circuit contains only linear elements, SPICE will use LU Decomposition [32] (a form of Gaussian Elimination) to solve the system equation. However, if the circuit contains non-linear elements the Newton-Raphson (N-R) Algorithm [32] is used.

Despite the fact the N-R algorithm is used when there are non-linear circuit elements present, SPICE does in fact only ever deal directly with linear equations. SPICE uses a technique known as *circuit linearisation* to produce linear approximations of non-linear models. These linear approximations are dependent on the operating region of the device in question, and so must be recalculated at every N-R iteration. Equation 5.1 shows the Newton-Raphson formula.

$$X_{n+1} = X_n - \frac{F(X_n)}{F'(X_n)} \tag{5.1}$$

Here, X_n is the current value, or initial guess of the unknown variable, X_{n+1} is the next value, and $F'(X_n)$ is the derivative of $F(X_n)$. The N-R algorithm attempts to solve systems of equations using an initial guess followed by series of iterations until it converges on an answer.

5.2.2 Aborted & Non-Convergent Simulations

All circuit simulators sometimes suffer from non-convergence. Simply put, non-convergence occurs when the simulator cannot find a set of voltages and currents which satisfy the system equation. In SPICE, specifically, non-convergence is the failure of the N-R algorithm to do this. If a set of voltage and current values cannot be converged upon, the simulator has no choice other than to abort the simulation. There are a number of reasons why this might occur, and there are also corresponding solutions. Different circuit analysis types have different failure mechanisms, but there are also some that are common to all types of analysis.

In practice, the number of N-R iterations allowed must be limited, and all incidents of non-convergence are a result of the N-R algorithm exceeding a predetermined iteration limit. However, in truth this is often just a symptom of another problem which is causing the algorithm to go through an excessive number of iterations.

Common Convergence Failure Mechanisms

The first failure mechanism common to all analysis types that will be examined is the impact of the error tolerance settings. The N-R algorithm needs some way of knowing when it has converged on a solution, and this is apparent when the values between two successive N-R iterations are equal. In practice, the criterion of having both values be exactly equal is unworkable, in part due to rounding and accuracy issues that any digital computer is subject to. However it may also require many more N-R iterations to converge to a more accurate solution, increasing the chance of exceeding the iteration limit.

SPICE addresses this problem by defining error tolerance limits. This way, the N-R algorithm will terminate and report back a set of converged values when the values found between successive iterations are *similar enough*, when $X_{n+1} \approx X_n$. There are three error tolerance limits in SPICE. The RELTOL limit sets the relative error tolerance. The VNTOL and ABSTOL limits define absolute voltage and current limits respectively. The N-R algorithm will terminate when either of these limits are satisfied. The absolute limits are needed, as for any voltage or current value which is converging to zero, the relative error tolerance will also converge to zero. The error tolerance limits can cause a convergence failure if they are set to values which require an excessive number of N-R iterations.

The other common failure mechanism is related to the conductance values of the circuit elements in the conductance array. These conductance values can also affect the speed with which the N-R algorithm converges. Recall that the algorithm is used when there are non-linear elements in the circuit, and that the process of circuit linearisation replaces these non-linear models with linear approximations. These linear approximations change at each N-R iteration, SPICE recalculates a new linear approximation for each non-linear circuit element. This means that some of the values in the conductance matrix change at each iteration. When calculating voltages, the N-R formula of equation 5.1 becomes equation 5.2.

$$V_{n+1} = V_n - \frac{F(V_n)}{G(V_n)} \tag{5.2}$$

As can be seen from this equation, the maximum and minimum circuit conductance values will affect how quickly the N-R algorithm converges on a solution. Very small values of G will result in very large changes at each iteration, meaning that V_{n+1} will be very far away from V_n. SPICE addresses this issue via the use of the GMIN variable. This specifies the value of a very small shunt resistor which is contained within every semiconductor model. The resistor is placed in parallel with every PN junction of the device. This also prevents G ever being equal to zero, which can happen as all semiconductor devices have regions of operation at which they output constant current (zero differential conductance). This would obviously lead to a divide by zero error. Very large values of G will also cause slow convergence, because each value of V_{n+1} will be very close to V_n.

In summary, there are two failure mechanisms common to all analysis types, due to the fact that both may increase the N-R iterations required and therefore lead to non-convergence and an aborted simulation. Setting the error tolerance limits to values suitable for the circuit being simulated can help. The other mechanism is the value of G in equation 5.2. The SPICE variable GMIN can be set to avoid problems with very low values of G, this should be set to the largest value that will not affect the operation of the circuit. GMIN should be set so that it's current contribution is always lower than the relative error tolerance limit. This will be of course be dependent on the circuit in question. For dealing with very high values of G, a series resistance value can be set in the semiconductor models used. This resistance should be small to avoid impacting on the operation of the circuit, but it will dominate at very high values of G and therefore can help limit the N-R iterations required.

Failure Mechanisms Of DC Bias Point Calculation

Strictly speaking, the failure mechanisms discussed here are unique to DC Bias Point analysis. But because SPICE always performs this type of analysis before attempting any other analysis type, this issues discussed here are also common to all other analysis types.

Calculating the DC operating point is the most difficult of the analyses SPICE can perform, as it starting from a blank sheet; very little information is known about the biasing conditions of the circuit before this analysis is run. There are two failure mechanisms that are relevant to this analysis type. Either the maximum N-R iteration limit will be too low for the circuit being analysed, or a poor initial guess of the unknown values in the circuit is used and results in an excessive number of N-R iterations being required.

The iteration limit for the N-R algorithm is controlled by the ITL1 SPICE variable. Increasing this from its default value can often help non-converging circuits. If DC operating point convergence fails, it is not immediately obvious what the cause is, but increasing this limit is a good place to start. It may simply be that a large number of iterations is required.

At the start of this analysis type, SPICE must make an initial guess of the voltages or currents at each node in the circuit. These values form a starting point for the N-R algorithm. The initial guess follows simple rules. Every node in the circuit is set to zero with the exception of any nodes connected directly to sources. These nodes are set to the corresponding voltage or current level. Depending on the particular circuit being analysed, this may be a particularly bad place to start.

There are two ways in which this may be addressed. SPICE gives the user the option of setting initial node voltages, and so the user can provide SPICE with known good guesses of the DC point of problem nodes. The other option is to use something called *source stepping.* This sets all the sources in the circuit to zero, and SPICE gradually increases these sources, calculating the DC bias point at each step, using the DC bias point from the previous step as a starting point for the next one. Because all the nodes are at a known value when this process starts (zero voltage) this improves the chance of finding the bias point when the sources are finally at full power.

Failure Mechanisms Of DC Sweep Analysis

There are also two failure mechanisms that can apply to DC Sweep analysis. Rapid voltage transitions and model discontinuities can both result in non-convergence. DC Sweep analysis is essentially a sequence of DC bias point calculations, with the results of the previous calculation being used as a starting point for the next one.

Rapid voltage transitions can cause non-convergence simply because the initial voltages used by the N-R algorithm can be long way from the actual voltages. The solution here is to increase the iteration limit for each step of the sweep. The SPICE variable ITL2 controls this.

Model discontinuities can be a problem if a sweep point falls on or near a discontinuity. This can effectively confuse the N-R algorithm, as at each iteration it can jump from one side of the discontinuity to the other resulting in an oscillation that just uses up iterations without progressing towards a solution. This can be remedied in two ways. Either the DC sweep step size can be increased or off set in an attempt to avoid hitting the discontinuity, or new parameters can be developed for the element model whose discontinuity is causing the issue. New parameters may ease or lessen the size of the discontinuity, effectively smoothing it slightly which might prevent the N-R algorithm from oscillating around that point.

There is another kind of sweep analysis offered by SPICE. AC Sweep analysis allows the frequency response of the circuit to be determined, however this analysis type does not actually suffer from non-convergence, other than during the initial DC bias point calculation which must be done for every analysis type. All element models are replaced by linear small-signal models during AC analysis, which allows SPICE to use the simpler LU decomposition method to solve the system equation. This method will always produce an answer.

Failure Mechanisms Of Transient Analysis

Transient analysis is the most complex analysis type that SPICE performs. It suffers from the same failure mechanisms that DC Sweep analysis suffers from, rapid voltage transitions and model discontinuities. However, transient analysis is very different to DC Sweep analysis.

During transient analysis, SPICE calculates a complete set of node voltage and branch current values at a series of time points. The solution from the previous time point is used as a starting point for the next one. During rapid voltage transitions this can lead to bad starting points for the next time point.

SPICE does not automatically abort a transient simulation the first time the N-R algorithm fails to converge. What it actually does is to discard the current time point, reduce the gap between the previous time point and the current one by a factor of eight, and tries again to find a satisfactory set of voltage and current values. This is known as *dynamic time step control* and happens automatically. This process repeats until it either converges on a set of values, or it reaches an internal step size limit and finally aborts the simulation.

One solution to this problem is to raise the iteration limit for each transient solution point (SPICE variable ITL4). This reduces the chance of SPICE failing to compute a time point and reducing the step size. A side effect of reducing the step size is an increased chance of hitting a model discontinuity, which is actually more likely during rapid voltage transitions.

However, it is inevitable that SPICE will come across a model discontinuity at some point. Most of the default semiconductor models in SPICE have their internal capacitances set to zero. Setting these to realistic, non-zero values will increase the chance of converging on a solution if a discontinuity is encountered.

5.2.3 Sources Of Error

Non-convergence is not the only issue that must be considered. The accuracy of the simulation is also extremely important if the fitness function is to work effectively. There are several ways in which errors can creep into an analogue circuit simulation.

The error tolerance limits are obviously very significant here, as are the model parameters. Both can introduce errors into all of the analysis types. However, it should in general be possible to manage this source of error. Transient simulations, however, introduce a whole swathe of new potential error sources which may be very difficult to detect.

Numerical Integration

Transient analysis must take account of time. The current flow through a capacitor is function of time, as is the voltage across a conductor. In order to calculate these quantities, SPICE uses numerical integration algorithms, and offers users a total of three different methods of numerical integration. These algorithms can be a significant source of error, worse still no single integration technique is suitable for types of circuit waveform which is why users are given a choice.

SPICE offers trapezoidal, backward-Euler and Gear numerical integration [24]. Each one has its own failure mechanisms which can introduce errors into the simulation. The default method is trapezoidal integration, because it is a good technique for many circuits. There are two ways in which trapezoidal integration can fail. It can introduce a ringing, or oscillation effect into the waveforms. These artifacts can be introduced by the simulator, they may not necessarily be due to the mechanics of the circuit. The other failure mechanism is the accumulation of error over time. All the integration methods use previous or current time points in order to calculate future time points. Errors can therefore build up in certain situations. This error is known as *local truncation error.*

Backward-Euler integration can suffer large amounts of local truncation error on non-linear waveforms, but it does not suffer from the oscillatory behaviour sometimes exhibited by trapezoidal integration.

Gear integration also does not oscillate, although it can overshoot when applied to rapidly changing or switching waveforms. It too can suffer from large amounts of local truncation error on certain waveforms. Each of these integration methods will introduce local truncation error with different types of waveform.

A significant difference between these types of failure and non-convergence failures is that SPICE simply has no idea that any error has been introduced into the simulation and therefore does not (indeed, it cannot) print a warning when this happens. It is down to the user to inspect the simulation output and detect any possible inaccuracies. The primary solution for all of these problems is simply to switch to a different integration method. Which one will depend on the circuit being simulated and the integration method which failed in the first place. If the error really was introduced by the simulator the suspected artifact will disappear. The other way in which errors from all three methods can be reduced is to reduce the time step.

Time Step Control

An important aspect of the way in which transient simulation works is the manner in which the time points are selected. SPICE solves the circuit equations at each time point. Again, there are several ways in which errors can creep in.

As already stated, SPICE adjusts the time step size as the simulation runs. The step size is reduced when a given time point will not converge, however the time step is *increased* again during periods of relative circuit inactivity in order to speed up the simulation.

SPICE has two different ways it can control how time steps are selected. The Iteration-Count time step control algorithm uses the number of Newton-Raphson iterations as its primary means of determining how the time step size should be adjusted (there are other factors which it takes into account). During periods of rapid transition, more N-R iterations will be required to converge on the next time point, and fewer iterations will be required during relatively stable phases of circuit operation.

The Local Truncation Error (LTE) time step control algorithm *estimates* the amount of error being generated by the numeric integration methods. If a large amount of error is being generated, the time step is reduced. It is increased if less error is being generated. While at times this produces good estimates of the error generated, it can be very wide of the mark in particular situations.

A poor choice of time step size by these algorithms can produce a number of problems. Aliasing, or under-sampling issues can occur for higher frequency circuits, while voltage or current pulses or rapid transitions may even be missed altogether. This will obviously lead to very erroneous simulation output. Other problems can also occur.

As with the numeric integration failure mechanisms, failures of the time step control algorithms must be spotted and corrected by the user, as SPICE has no way of knowing when they occur.

5.2.4 Other Simulators

There are several analogue simulators available, such as PSPICE, Microcap and HSPICE, amongst others. Although they are all different implementations of the same basic ideas, none of the simulators available have completely eliminated the issues discussed in this chapter.

Perhaps the most notable simulator is HSPICE [8], which is accepted as something of an industry standard. This has many improvements in many areas. For example, it uses its own proprietary time step control algorithm, known as DVDT. Many of its device equations have also been rewritten which has resulted in smoother discontinuities.

5.2.5 Implications For Genetic Algorithms

There are many ways in which SPICE, or indeed any simulator can fail to converge or produce accurate simulations. All of the failure mechanisms do have solutions, however not all solutions are applicable if the simulator is used as part of a GA. The reason is simple: the detection of many of the problems, and the application of many of the solutions require human intervention.

Any fitness function, no matter how it works or what it is evaluating, *must* be fully automated. A GA will typically sort through many thousands of candidate solutions during its search and it simply is not feasible to require a human user to inspect the solution every time there is a problem in measuring it. Indeed, this defeats the very point of automating any design process.

And this is precisely why analogue circuit simulators are not at all well suited for use as part of a GA. SPICE, and indeed any good quality circuit simulator, is capable of producing *extremely* accurate simulations and circuit analyses. But they will not necessarily do so on their own, simply as a result of pushing the start button while using default settings. Like all other aspects of analogue design, successful analogue simulation requires the experience, intuition and understanding of a human engineer. This engineer must have a good understanding of the circuit being simulated, the workings of the simulator itself and he or she must also have a good expectation of what the result of the analysis will be. As stated by Kielkowski [24]:

> ***Never simply assume the simulation output is correct.*** *If questionable results appear, use good engineering judgment to determine whether the anomaly was simulator-related or design-related.* ***Never simulate a circuit without a reasonably good idea of what the simulation output should be.*** *A good designer would never breadboard a collection of resistors, capacitors, inductors, and transistors without some reasonable expectation of the behaviour of the circuit.*

Table 5.1 summarises the failure mechanisms, and corresponding solutions, discussed in this chapter. The point of this discussion is to illustrate the *lack of robustness* common to all analogue circuit simulators. While it should never be assumed that the output from the simulator is correct, when used as part of a GAs fitness function, this assumption is in fact implicit; it is required. Therefore, there would appear to be an inherent incompatibility between analogue simulators and genetic algorithms.

What implications does this have for GAs? And how can these issues be addressed? The answer to these questions depends very much on what is being attempted. As discussed, the simulation failure mechanisms can be categorised as either causing erroneous simulation output, or causing non-convergence. Both obviously have the potential to seriously impact the effectiveness of the GA. In the case of erroneous simulation output, possible effects include GA convergence of a solution which is does not satisfy the original problem, or good candidate solutions may be thrown away.

In the case of non-convergence, the GA has very little information about the suitability of the circuit. When designing the fitness function, a decision will have to made about what to do when no simulation data is returned. There seems to be little alternative to simply assigning that circuit a very poor fitness score. This obviously has the potential to miss good circuits, or potentially useful aspects of the circuit's behaviour. The majority of the solutions a GA processes are likely to be poor, however some will be poorer than others and this strategy does not allow any distinction.

In order to address this, it may be possible to preempt *some* of these simulation issues but by no means all of them. As shown in table 5.1, there are some settings which should be used universally. Things like raising iteration limits, and using realistic device model capacitances will help.

If the problem the GA is tackling is very constrained, that is, if the topology is mostly or entirely fixed, and the allowable component value ranges are small, then it is likely that the majority of the circuits the GA encounters will be very similar, with similar behaviour. This means that a set of simulator parameters can be used that are suitable for this circuit type and behaviour. This should reduce the number of non-convergent circuits encountered but it does not guarantee that non-convergence will be eliminated.

If the GA is tackling a very unconstrained problem, such as one in which the topology is completely free to evolve, then the situation becomes much worse. It would then be very difficult, or maybe even impossible to preempt the circuit types and behaviours that the GA will encounter. It is likely encounter many *different* circuit topologies and characteristics and therefore it will not be possible to apply a set of universally suitable simulation parameters. There is probably no other alternative than to simply accept that the GA will not be able to effectively process or use many of the circuits or circuit characeristics it attempts to measure.

Analysis Type	**Failure Mechanism**	**Solution**	**Solution In GA Context/(Solution Workable?)**
Common	Poor settings of error tolerances	Set tolerances to appropriate values	Set tolerances to appropriate values for circuit being evolved. (**YES**)
	Poor minimum conductance value	Set GMIN to value that will not affect circuit	Cannot be done universally, must be done on a case-by-case basis. (**NO**)
DC Bias Point	Not enough N-R iterations available	Raise iteration limit (ITL1)	Can be set to high level for all circuits. Will be used only if needed. (**YES**)
	Initial guess is poor	1. Manually set initial voltage. 2. Use source stepping.	1. Cannot be done universally, must be done on a case-by-case basis. (**NO**) 2. Can be enabled for all circuits, but can have significant GA run time impact. (**YES**, with penalty.)
DC Sweep	Rapid voltage transitions	Raise iteration limit (ITL2)	Can be set to high level for all circuits. Will be used only if needed. (**YES**)
	Model discontinuities	1. Increase DC step size. 2. Develop new model parameters for problem circuit element.	1. Cannot be done universally, must be done on a case-by-case basis. Can affect accuracy. (**NO**) 2. Cannot be done universally, must be done on a case-by-case basis. (**NO**)
Transient Analysis	Rapid voltage transitions	Raise Iteration limit (ITL4)	Can be set to high level for all circuits. Will be used only if needed. (**YES**)
	Model discontinuities	Set semiconductor model capacitances to realistic values	Can be set for all circuits. Should be done anyway. (**YES**)

Table 5.1: Non-Covergence Mechanisms For Common SPICE Analysis Types

Chapter 6

A GENETIC ALGORITHM SYSTEM FOR ANALOGUE SYNTHESIS

This chapter presents the design and implementation of a system for synthesising analogue circuits. Based on Genetic Algorithms, it uses a circuit encoding system very similar to that used by the Genetic Programming technique [26][30] (with the addition of some novel modifications), and a novel fitness function based on pole-zero analysis. The system is designed to be extremely flexible and configurable, allowing it tackle many different types of circuit of arbitrary size and complexity. It can also accept pre-defined information about the ideal topology of the target circuit of arbitrary detail and also accept pre-defined useful sub-circuit structures.

6.1 Objectives

The holy grail of design automation, of analogue or digital circuits, is to be able to produce useful, suitable-for-purpose circuits with the minimum of human intervention and guidance. Genetic algorithms are an ideal candidate for further investigation as they allow just this possibility. Chapter 4 describes them in detail. Certainly, great claims have been made of the ability of genetic algorithms to discover novel circuit designs [30][36][53]. Despite this, many implementations have been very limited and are typically restricted to a small set of circuits with limited scope for changing topology.

In order to fully explore the potential of genetic algorithms, an implementation is required that can attempt to synthesise circuits of arbitrary function, size and complexity.

6.2 Overview Of Implementation

When discussing a genetic algorithm system an important distinction must be made between the *algorithm* itself and other aspects of the system's implementation. Indeed, at the highest level the algorithm of any GA system must, by definition, be very similar. However, the specifics of implementation of a GA system such as the encoding method and genetic operators used, as well as the *application* of a GA system can vary greatly. This is also true of the finer details of the algorithm itself - GAs can be highly complex. The algorithm used in the GA system presented here is shown in figure 6.1.

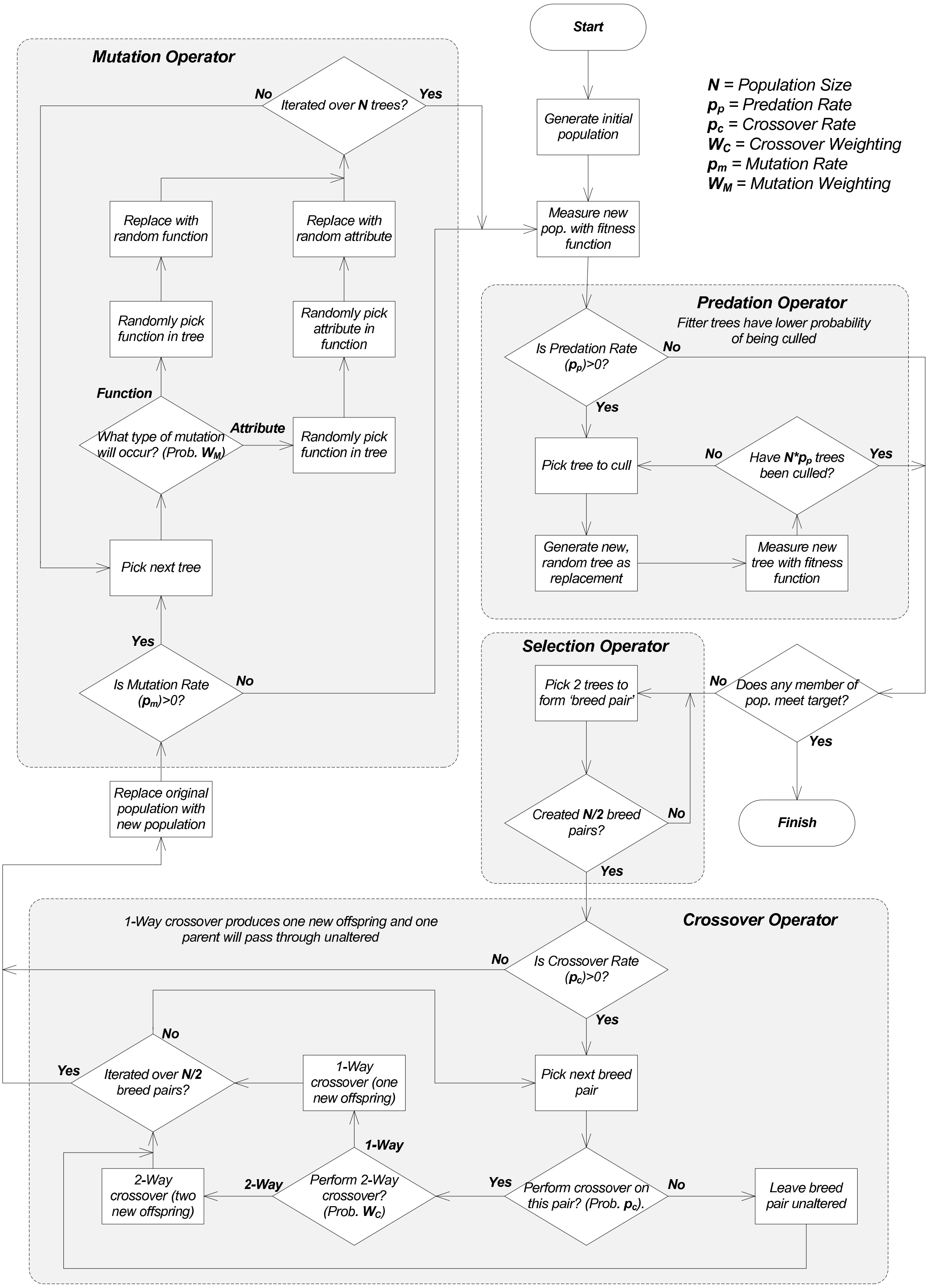

Start
Generate initial population
Measure new pop. with fitness function
N = Population Size
p_p = Predation Rate
p_c = Crossover Rate
W_C = Crossover Weighting
p_m = Mutation Rate
W_M = Mutation Weighting
Mutation Operator
Iterated over N trees?
No
Yes
Replace with random function
Replace with random attribute
Randomly pick function in tree
Randomly pick attribute in function
Function
What type of mutation will occur? (Prob. W_M)
Attribute
Randomly pick function in tree
Pick next tree
Yes
Is Mutation Rate (p_m)>0?
No
Replace original population with new population
Predation Operator
Fitter trees have lower probability of being culled
Is Predation Rate (p_p)>0?
No
Yes
Pick tree to cull
No
Have N*p_p trees been culled?
Yes
Generate new, random tree as replacement
Measure new tree with fitness function
Selection Operator
Pick 2 trees to form 'breed pair'
Created N/2 breed pairs?
No
Yes
No
Does any member of pop. meet target?
Yes
Finish
Crossover Operator
1-Way crossover produces one new offspring and one parent will pass through unaltered
No
Is Crossover Rate (p_c)>0?
Yes
No
Yes
Iterated over N/2 breed pairs?
1-Way crossover (one new offspring)
Pick next breed pair
1-Way
2-Way
2-Way crossover (two new offspring)
Perform 2-Way crossover? (Prob. W_C)
Yes
Perform crossover on this pair? (Prob. p_c).
No
Leave breed pair unaltered

Figure 6.1: Implemented Genetic Algorithm

The main inputs to the system are a collection of configuration files which define the behaviour and any constraints of the required circuit, and various GA behaviour and sizing parameters. The principle output of the GA system is the evolved circuit, or at the very least the best circuit found if convergence is not achieved. The system also outputs detailed information about the progress of the GA.

There are four main components to this GA system. The encoding method used is based on a method developed by John Koza, and is based on his Genetic Programming method [26][30]. It employs extremely flexible tree structures of circuit constructing functions, allowing easy modification of circuit size and topology. The genetic operators used in the system are very similar to those found in the majority of other GA systems, however the crossover and mutation operators have been designed for use with tree structures.

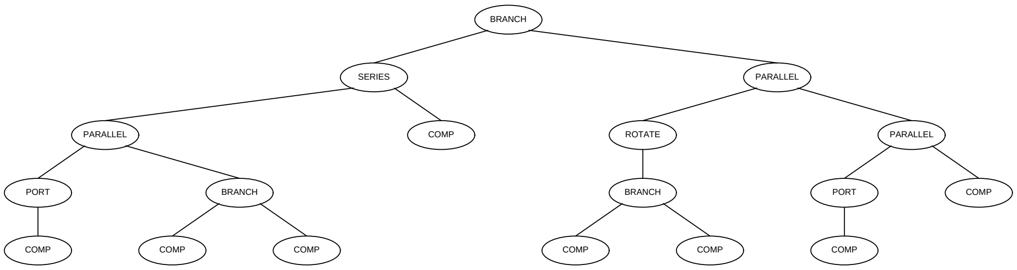


Figure 6.2: Tree of circuit constructing functions.

Measurement and characterisation of candidate circuits is carried out using the commercially available HSPICE circuit simulator [8]. The final component of this GA system is the fitness function used. There are two user selectable fitness functions available.

6.2.1 Overview Of Encoding Method

The encoding method employed is explained in detail later in this chapter in section 6.4. However, an example of how a small circuit might be encoded is provided here in order to familiarise the reader with it.

As shown in figure 6.2, each candidate circuit is encoded as a tree of circuit constructing functions. Each of these functions takes either one or two 'arguments', and outputs a single argument which is passed up the tree to the next function. These arguments are actually circuit networks consisting of one or more components. The functions take either one or two of these circuit networks and process them in some way. Each network is considered to have only two *external ports*, even though a network may have additional unconnected component terminals. These external ports determine how each function processes the circuit networks, this is explained in detail in section 6.4.1.

Each function also has a set number of *function attributes*. These are real numbers between 0 and 1, and affect how the function operates. Not all functions make use of the attributes, but all functions in the tree have the same number, which is set by the user. The type of components available to the GA (which is also set by the user) will determine how many of these attributes are required.

During the construction of a circuit, a *circuit embryo* is used. The circuit embryo defines the circuit inputs and outputs, and also things such as signal sources or power supplies which are needed by SPICE when simulating the circuit - it functions as a test harness. It also contains at least one *modifiable wire*. It is this wire that is modified by the tree of circuit constructing functions, and is eventually transformed into the circuit network output by the tree of functions.

The circuit shown in figure 6.3 is a high-pass Sallen-Key filter. Figure 6.4 shows the circuit embryo used, and figure 6.5 shows an example encoding for the circuit. Finally, figure 6.6 shows the process of *circuit finalisation* (see section 6.4.4), during which the circuit network returned from tree of functions is inserted into the circuit embryo and any dangling component terminals that may remain are connected.

In this example *Loose End A* is labelled with an attribute which is less than 0.5. This means that it will be connected to one the points defined in the circuit embryo. In this case, The two power supplies used by the Op-Amp, and ground, have been defined. The attribute is used to determine which of the points to connect to. The power supplies are not shown connecting to the Op-Amp in this example. When a subcircuit is defined in this GA system, it may be defined such that any of its circuit nodes are hardwired to one of the nodes in the embryo. In this case, the Op-Amp has been defined in such a way that it is hardwired to the power supplies of the embryo.

The other dangling point, *Terminal Port A*, is labelled with an attribute greater than 0.5. This means that the labelled ports of the constructed circuit are used as possible connection points for the dangling terminal. The ends of the modifiable wire are also always made available to dangling terminals.

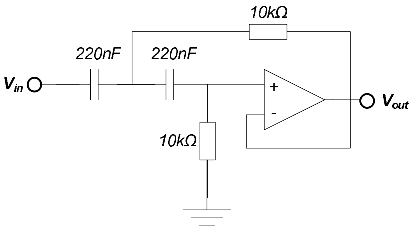


Figure 6.3: High-Pass Sallen-Key Filter.

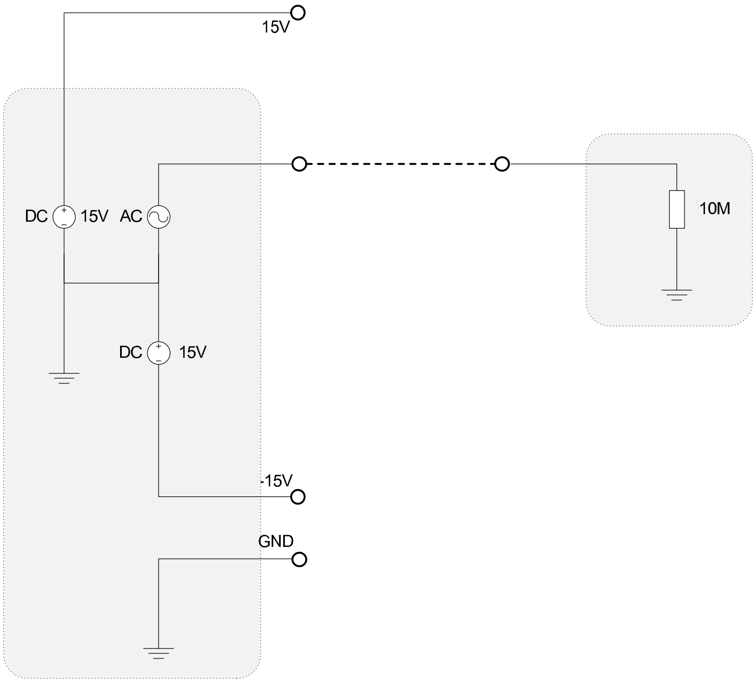
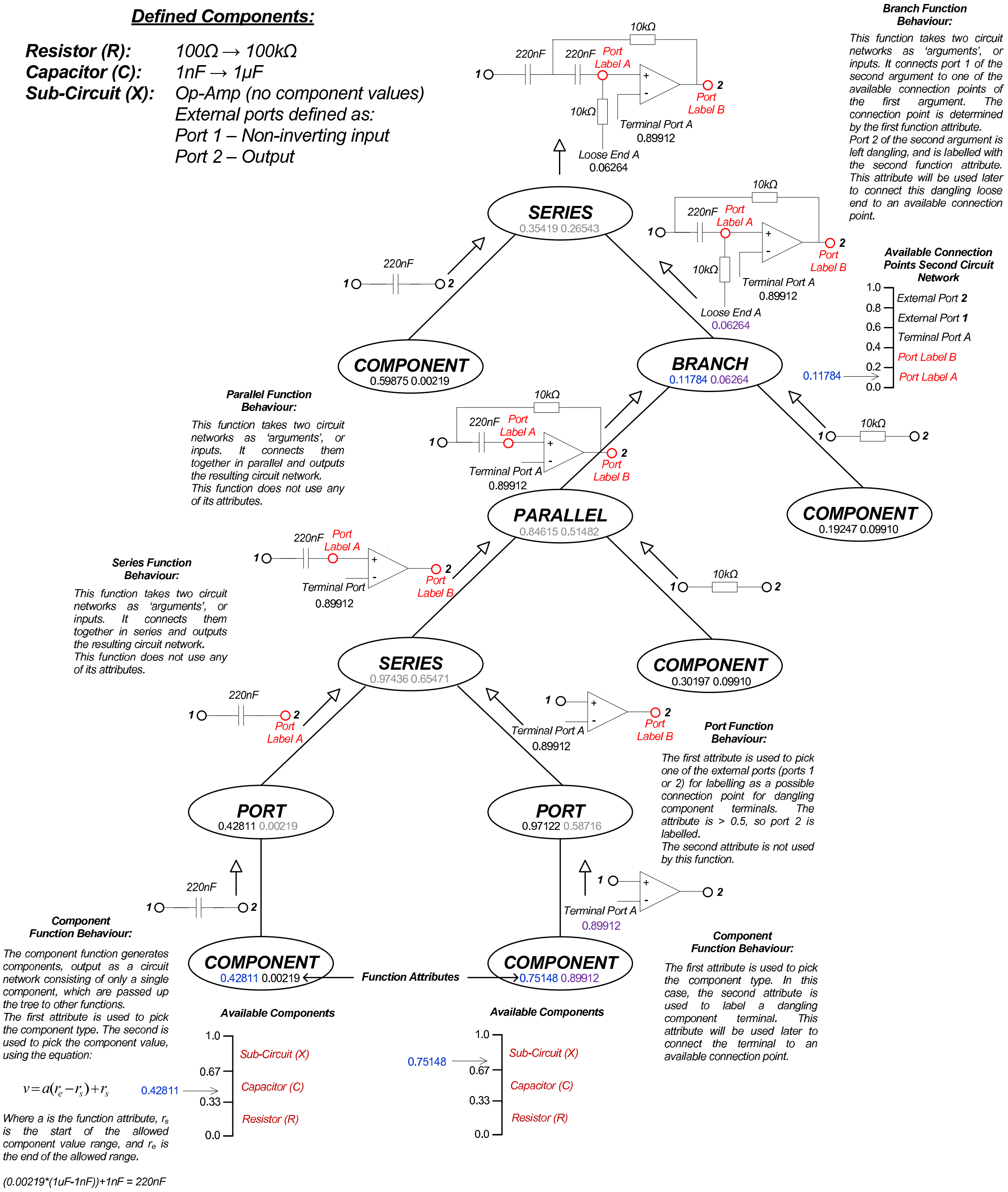


Figure 6.4: Example Circuit Embryo.



Defined Components:

Resistor (R): 100Ω → 100kΩ
Capacitor (C): 1nF → 1μF
Sub-Circuit (X): Op-Amp (no component values)
External ports defined as:
Port 1 – Non-inverting input
Port 2 – Output

Branch Function Behaviour:

This function takes two circuit networks as 'arguments', or inputs. It connects port 1 of the second argument to one of the available connection points of the first argument. The connection point is determined by the first function attribute. Port 2 of the second argument is left dangling, and is labelled with the second function attribute. This attribute will be used later to connect this dangling loose end to an available connection point.

Available Connection Points Second Circuit Network

Parallel Function Behaviour:

This function takes two circuit networks as 'arguments', or inputs. It connects them together in parallel and outputs the resulting circuit network. This function does not use any of its attributes.

Series Function Behaviour:

This function takes two circuit networks as 'arguments', or inputs. It connects them together in series and outputs the resulting circuit network. This function does not use any of its attributes.

Port Function Behaviour:

The first attribute is used to pick one of the external ports (ports 1 or 2) for labelling as a possible connection point for dangling component terminals. The attribute is > 0.5, so port 2 is labelled. The second attribute is not used by this function.

Component Function Behaviour:

The component function generates components, output as a circuit network consisting of only a single component, which are passed up the tree to other functions. The first attribute is used to pick the component type. The second is used to pick the component value, using the equation:

$$v = a(r_e - r_s) + r_s$$

Where a is the function attribute, r_s is the start of the allowed component value range, and r_e is the end of the allowed range.

(0.00219*(1uF-1nF))+1nF = 220nF

Component Function Behaviour:

The first attribute is used to pick the component type. In this case, the second attribute is used to label a dangling component terminal. This attribute will be used later to connect the terminal to an available connection point.

Figure 6.5: Example Of Tree Encoding.

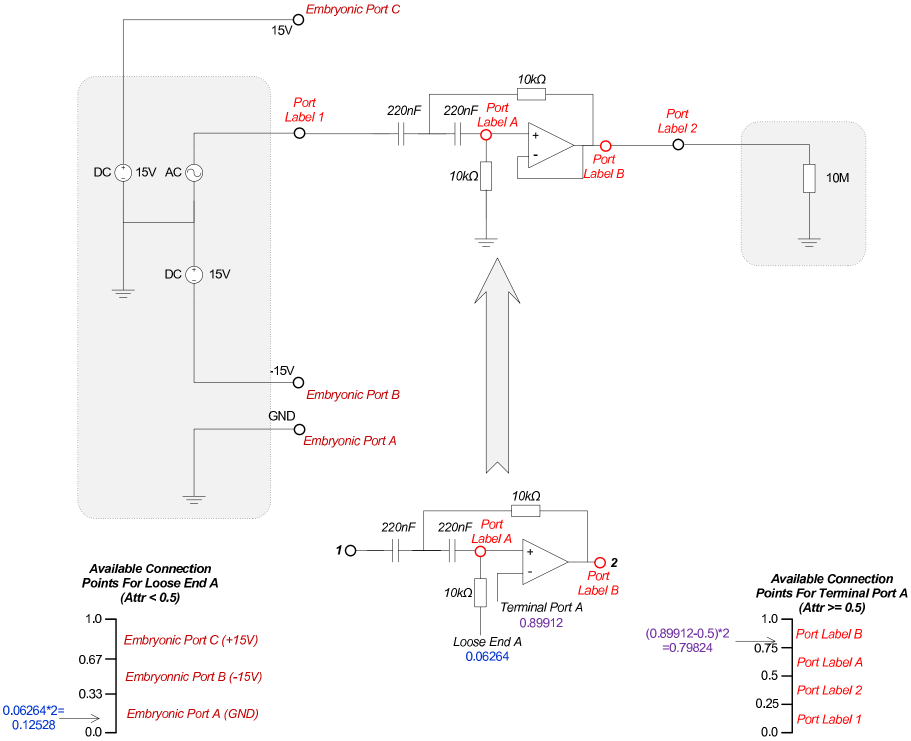


Figure 6.6: Circuit Finalisation.

6.3 GA System Inputs

The GA system needs many input parameters in order to function correctly. There are many parameters which control the behaviour and operation of the GA; these are set in a main configuration file. Parameters such as population size, maximum number of generations and stop criterion are specified here, as well as the location and name of the other input configuration files which must be read in.

One of the most important items of input data is the target data file. This contains data which specifies the desired circuit behaviour and characteristics that each candidate circuit will be measured against. The exact format of this file will depend on the fitness function being used. The fitness functions available in this GA system are discussed in section 6.7 of this chapter.

A SPICE configuration file specifies the SPICE parameters used in the simulation of each candidate circuit. The SPICE options and commands contained in this file are simply copied verbatim into each SPICE netlist written out.

6.3.1 Component Definitions

The number and type of components the GA is allowed to use are likely to have a significant impact on the effectiveness and execution time of the GA as well as the quality of results. It is one of the main parameters that directly controls the search space the GA has to work with. By specifying the available components and their allowed range of values, the user is providing information about the topology and the sizes of components of the ideal circuit as well bounding the solution space which the GA will search. The user must be sure that an adequate solution lies within this search space.

A list of available component types, along with allowed value ranges is supplied to the GA as an input. The GA system recognises a number of standard component types, in addition to this subcircuit definitions can also be made. This allows known good sub-topologies to be defined for use by the GA, for example amplifiers, buffers or current mirrors. If used, these subcircuits are instantiated in their entirety as an atomic unit and cannot be split or modified by the GA.

The complete list of recognised components is shown is table 6.1.

Component Type	**No. Terminals**	**Default Terminals (Spice Terminals)**	**Spice Name/ Model Name**	**No. Values**
Resistor	2	0, 1	R/None	1
Capacitor	2	0, 1	C/None	1
3-Terminal N-Channel MOSFET	3	0,2 (Drain & Source)	M/User Specified	2 (Length & Width)
3-Terminal P-Channel MOSFET	3	0,2 (Drain & Source)	M/User Specified	2 (Length & Width)
4-Terminal N-Channel MOSFET	4	0,2 (Drain & Source)	M/User Specified	2 (Length & Width)
4-Terminal P-Channel MOSFET	4	0,2 (Drain & Source)	M/User Specified	2 (Length & Width)
Diode	2	0, 1 (D+ & D-)	D/User Specified	None
Inductor	2	0, 1	L/None	1
Sub-Circuit	Arbitrary	User Specified	X/User Specified	Arbitrary

Table 6.1: Recognised Component Types

The GA system builds up a list of components that have been defined in the input component file. A component definition in this file must include maximum and minimum component value limits. Multiple definitions of the same component type may exist, and may also have different value ranges defined. Components such as transistors require more than one value (length and width) and limits for each value may be set separately. Subcircuit components may require an arbitrary number of values and each one may have a separate value range. Some components may have a secondary, or *model* name specified. This name refers to a SPICE model definition that must be included in the SPICE options input file. This is useful for MOSFETS and diodes, for example, where the model might specify threshold voltage, breakdown voltage and so on depending on the type of component. This allows the user complete control over the types of components available to the GA. The instantiation of components defined in this list is controlled by the *Component* circuit construction function described later in section 6.4.2. The 'default terminals' listed in table 6.1 are considered to be the 'external' connection points of the device unless changed by one of the circuit construction functions. A detailed discussion of how components are treated is deferred until section 6.4.

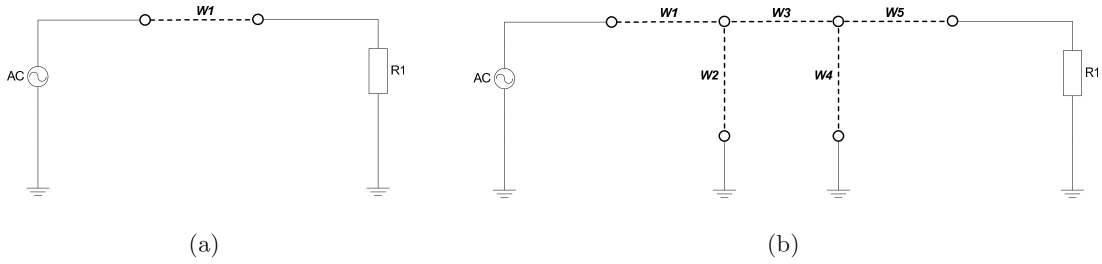
As well as simply defining the types of components which are available to the GA, certain components may also be defined with a set of restrictions. For example, specific terminals of a component may be defined as being tied to a particular node in the circuit embryo (the circuit embryo is discussed in section 6.3.2). Any number of values associated with the component may also be set to a specific value. This allows an arbitrary amount of predefined knowledge of the final circuit to be given to the GA.

6.3.2 Circuit Embryo Definition

A circuit embryo definition must be supplied. The circuit embryo at its most basic is essentially a simulation test harness for each candidate circuit. It defines available power supplies and circuit inputs and outputs. It must also define at least one *modifiable wire*. It is this wire that is modified by the trees of circuit construction functions. Certain nodes within the embryo may be defined as available connection points for any dangling component terminals or branches during circuit finalisation (see section 6.4.4). This may be useful for tying off component terminals to ground or one of the power supplies.

Any node within the embryo may also be defined as requiring termination by a large resistor if, by the end of circuit finalisation, that node is dangling - that is, it is only connected to a single component. This helps to ensure that the circuit netlist read in by SPICE is 'legal'.

The structure of the embryo may be used to provide the GA with information about a suitable circuit topology. An extremely simple embryo is shown in figure 6.7a. A more complex embryo is shown in figure 6.7b. This embryo may, for example, be used to provide topological guidance to the GA during synthesis of a simple ladder filter.



(a) (b)

Figure 6.7: Examples of Circuit Embryos

6.4 Encoding Scheme

The encoding scheme used in this GA system is based on tree-like structures as opposed to the string-based encoding schemes used in more traditional GA systems. It is inspired by John Koza's encoding scheme presented in chapter 4. The principle operation performed by GA systems is crossover, as this allows the possibility of combining the desirable characteristics of two fit individuals. In the classic GA, which utilises relatively simple string based structures, crossover is usually a simple operation. String encoding schemes are usually designed in such a way that the strings can be cut at any arbitrary point, and when the partial strings are recombined with other partial strings the result still 'makes sense' - that is, the encoding method maintains *coherence*.

The requirement here is an encoding method that allows the easy manipulation of circuit topology, not just component values. However, an encoding scheme that presents natural cutting points in a circuit topology is not obvious. String based encoding schemes seem largely unsatisfactory for this purpose, however such schemes have been used to represent topology [36]. Components can have different numbers of terminals, and can have more than one value (the dimensions of a transistor, for example). Indeed, components may have typical values of entirely different orders of magnitude depending on their type. While variable length string encoding schemes have been developed [58] (section 4.6.2), these are still inherently limited.

Koza's GP system is designed to evolve computer programs, and therefore requires an encoding scheme capable of easily representing and manipulating such a complex structure. Computer programs cannot be cut at any arbitrary point. A program can be considered to consist of atomic building blocks, such as variables, function calls, flow of control constructs, terms in expressions and so on. Manipulating the program in its parse tree form, as Koza does, enables the structure to be represented in such a way that cutting points are only presented between these atoms. This is vital for maintaining coherence during crossover.

The aim here is to automate the design of electronic circuits, not computer programs. The use of a tree of circuit constructing functions offers an extremely flexible way of manipulating circuit topology via crossover which overcomes many of the limitations of the string based schemes. It provides a way of naturally presenting cutting points that neatly isolate individual components as well as subcircuits within a circuit.

The GP method has a rather complex implementation however, which requires the execution of a LISP [48] program in order to produce a circuit. This circuit must then be evaluated which involves the use of a circuit simulator. This is a result of the GP system being very flexible. It directly evolves computer programs which can be applied to problems of arbitrary nature. When applied to the problem of circuit design automation, it only *indirectly* evolves these circuits. For the purposes of investigating only circuit evolution the execution of LISP programs is an unnecessary overhead.

The GP method is also prone to producing circuits which contain potentially useless components, such as resistors which have only one terminal connected. Transistors or other components which have more than one terminal may also have dangling terminals which may result in erratic or unpredictable circuit behaviour.

The function trees in the GA system presented here are evaluated in a bottom-up fashion, with each function returning a result that is used by the function at the next level up of the tree. Finally, when the last function executes at the top of a tree, a circuit is returned that fits into a designated section of the circuit embryo.

Any encoding scheme applied to analogue circuits must be able to fully define such circuits. An analogue circuit consists of: components of a given type, component values, circuit inputs and outputs and the interconnections between the components (circuit nodes). The nodes within a circuit form the 'backbone' of its topology, and connections of component terminal to those nodes flesh it out. The encoding scheme must not only be able to adequately express these things, but consideration must also be given to the impact of the genetic operators. As already discussed, the encoded circuit must still 'make sense' after the crossover and mutation operators have been applied. The interconnections between circuit components define topology and it is this that is perhaps the most difficult thing to encode in a natural, efficient manner.

Simpler, more direct encoding schemes will more closely resemble a netlist, since a netlist is a literal, explicit representation of a circuit. It is certainly possible to encode all the defining features listed above in such a scheme, but the issue comes when the representation is modified, particularly with regard to the interconnections and topology. In a netlist, a component is typically instantiated, followed by a list of which circuit nodes the component's port and terminals are connected to. The problem here is that in such simple encoding schemes, some way is needed to first define what circuit nodes exist, and there is unlikely to be any implicit relationship between the components in a circuit and the nodes they connect to. Not without making the encoding more complex.

A tree of circuit constructing functions, however, is a *description* of how to build a circuit. The nodes within a circuit are not explicitly defined in such a representation, but instead instructions describe its construction. The number of circuit nodes, and the components connected to them are derived. For example: *'connect these two components in series'* and *'rotate this subcircuit and then connect in parallel with this resistor'.* Because the circuits are implicitly defined, the tree of these instructions may be much more easily modified. The way the circuit topology is derived from the encoding is therefore much more elegant than simpler, more direct schemes.

6.4.1 Two-Port Circuit Networks

In order to maintain a system whereby any tree of circuit construction functions, of any combination, is capable of being evaluated, the available functions must be designed in a such a way that the output from any one of them, under any circumstances must be acceptable as input arguments to any other function, including other functions of the same type. Each one must be capable of executing its defined behaviour on anything passed to it as an argument. This requirement means that every function must treat any argument as the same thing. To use a software analogy, there is only one *datatype* used in these trees of functions, and each function both returns that datatype and accepts arguments of only that datatype.

The construction functions manipulate the physical structure of a circuit. They necessarily accept components or subcircuits as inputs and they return components or subcircuits as outputs. Therefore, this *datatype* is in actual fact a circuit network.

Within any given circuit it is possible to identify distinct modes of connection of its constituent components. It is quite apparent that series and parallel configurations can fully define some circuit topologies. However, these two configurations cannot exhaustively define *any* topology. Consider the resistor network shown in figure 6.8a. R2 is connected neither in series or parallel with any other component in the network. The same is true of capacitor C1 in figure 6.8b. Such configurations of topology will be referred to as circuit *branches*, and are defined as any configuration of two or more components that is not series or parallel.

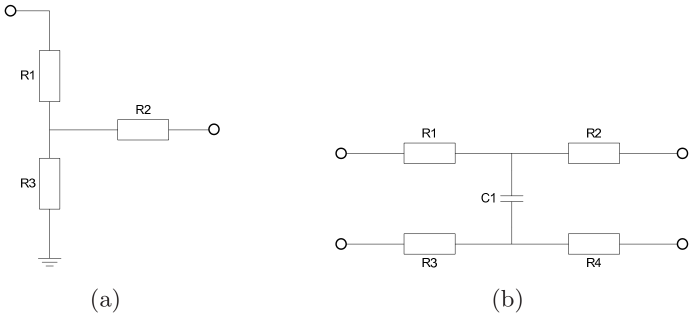
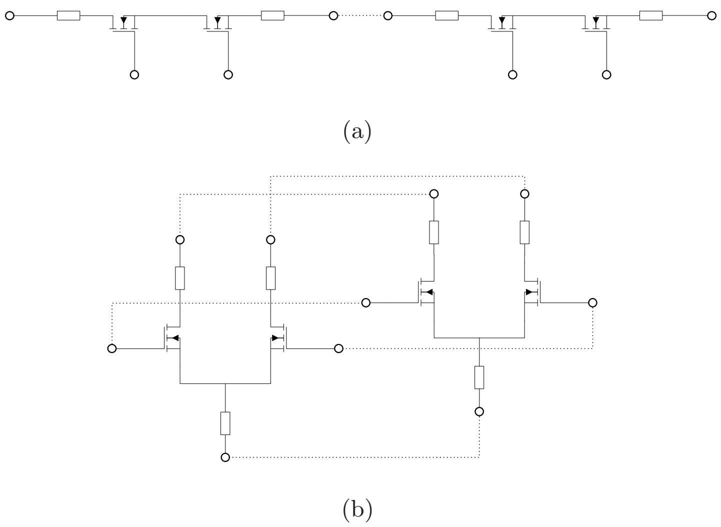


Figure 6.8: Examples of Circuit Branches

It would certainly seem useful, then, to have circuit construction functions which accept two networks and connect them in a series, or parallel configuration, and to also have a function which could connect a circuit network between arbitrary circuit nodes. While this function may sometimes result in a series or parallel connection with other components, it could also allow circuit branches to be formed.

The networks passed between these functions can be of any size or complexity, containing any number or type of components. Because a variety of pre-defined functions must operate on these networks, certain requirements must be imposed on how a network, regardless of its size, must be treated. For example, it is not necessarily obvious how to connect two networks together in series. The networks might contain several components, and some or all of those components may have more than two terminals. The two networks may be completely different. In these circumstances it may not be at all obvious, even to a human designer with knowledge of the networks function, just what a 'series arrangement' of the two networks should look like (figure 6.10a).

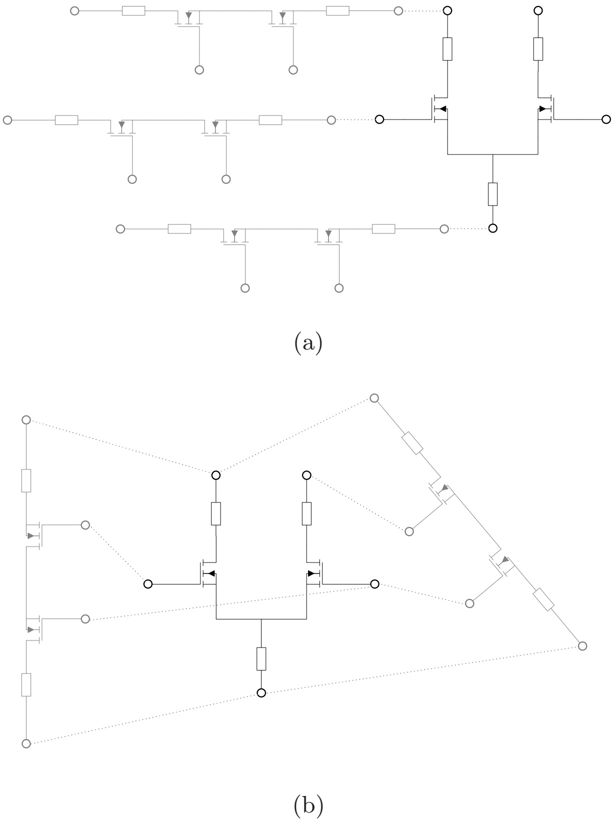
It is a similar case for connecting networks together in parallel. In this case however, if both networks are identical then each terminal on both networks can be connected (figure 6.9b). Components are connected in parallel if they have the same voltage across their terminals. However, if the two networks are different then the idea of connecting them in parallel is quite possibly meaningless (figure 6.10b).



(a)

(b)

Figure 6.9: Series & Parallel Connection of Circuit Networks

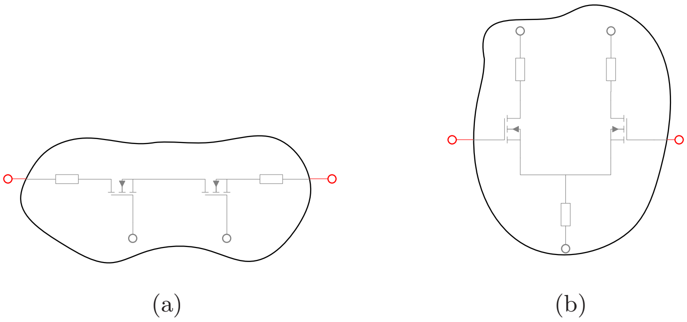


(a)

(b)

Figure 6.10: It is not always obvious how two circuit networks may be connected in series or parallel.

Two networks are only guaranteed to be recognisably connected in series or parallel in any circumstance if the networks in question have only two terminals. It is for this reason that the networks passed between the circuit construction functions are considered to be *two-port networks*. This guarantees that each construction function will treat any argument passed to it in exactly the same way, with predictable results. This in turn means that the trees of functions can be split at any point and crossed over with any other tree, and coherence will be maintained. The word 'port' is used in this text to simply refer to a single-terminal *connection point* on a component or subcircuit, so a two-port network simply has two connection points.


(a)

(b)

Figure 6.11: Multi-terminal circuit networks are always considered to have just two primary ports.

In reality, of course, networks may very well have more than two terminals. Transistors have more than two terminals, and user defined sub-circuits may have an arbitrary number. Clearly some way is needed of handling these networks. Each network has a pair of *external ports*. It is these ports that are used by the circuit construction functions when executing their defined behaviour. However, each network maintains a list of dangling component terminals. These terminals are connected to other parts of the circuit, according to certain rules, during the circuit finalisation phase. This is described in detail in section 6.4.4.

The components defined in the components definition file have default terminals specified. It is these terminals that are initially considered to be the external ports of a network. The GA system assumes certain component's terminals to be the default terminals unless the user specifies otherwise. These terminals are listed in table 6.1.

6.4.2 Circuit Construction Functions

The encoding scheme presented here employs trees of circuit constructing functions that can be evaluated to directly produce a circuit netlist without the need to execute any derived program first.

A total of six functions are used, most can accept either one or two arguments depending on its type. The arguments that each function accepts are two-port circuit networks. Each function returns a two-port network. Each function also has a list of random *attributes*, real numbers between 0 and 1, which can affect the function's behaviour. A variable in the main configuration file determines the number of attributes stored by each function. The function types employed are:

- **Component** - Does not accept any arguments, but returns a two-port network consisting of only a single component. The type and size of this component depends on the attributes of the function.
- **Series** - Accepts two two-port networks, connects them in series and returns the result as a single network.
- **Parallel** - Accepts two two-port networks, connects them in parallel and returns the result as a single network.
- **Branch** - Accepts two two-port networks. It connects the two networks together at one of their ports, and returns the result as a single two-port network with a dangling circuit branch which will be connected to another part of the circuit at a later point during the decoding of the circuit.
- **Rotate** - Accepts a single two-port network, and rotates it. It returns the result.
- **Port** - Accepts a single two-port network. It does not directly modify this network, but labels one of its ports as an available connection point for any unconnected component terminal or circuit branch.

Component Function

The component function accepts no two-port networks as arguments. The purpose of this function is to introduce components into the tree, using attributes to choose the type and values of the component. This function type *must* appear at the end of every branch in the tree and nowhere else.

The number of attributes required by this function varies depending on the components types that have been defined. The first attribute always determines the type of component generated. A list of components types is built up when the component definitions file is loaded. Figure 6.12 shows how the component type is determined.

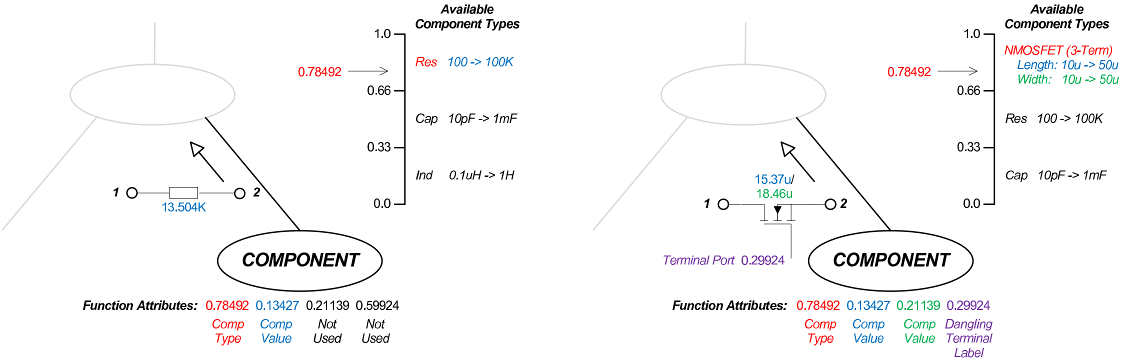


Figure 6.12: Component Construction Function

When the component type has been selected, component values are assigned. Some components may require more than one component value, and attributes are used to select these values. The second attribute is used to pick the first value (the first was used to determine the component type). Any further values are picked using the next attributes in sequence. Each component definition input into the system includes a value range for each component value. Equation 6.1 shows how the attribute is used to select the value, where v_{comp} is the final selected value, a is the attribute, r_s is the start of the allowed value range and r_e is the end of the value range. r_e is always large (less negative) than r_s.

$$v_{comp} = a(r_e - r_s) + r_s \tag{6.1}$$

Finally, for each terminal that a component has over and above the minimum of two, an attribute will be assigned to free terminals. This attribute will just be the raw, floating point number. This number will be used during circuit finalisation when dangling terminals are tied up. Section 6.4.4 explains this in detail.

The number of attributes that need to be assigned to each function will be determined by the component type with the most component values and the number of terminals it has. In reality this will be either a MOSFET or subcircuit if they are used.

Series

The series function accepts two arguments. It takes the two networks, and connects the second external port of the first network to the first external port of the second argument as shown in figure 6.13. The series function has an important property - it creates a single, new circuit node. The external ports of each argument that are connected are smashed into one, new circuit node.

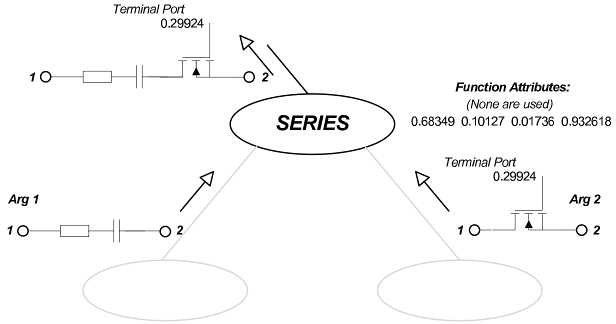


Figure 6.13: Series Construction Function

The new network has the dangling terminals and labeled ports of both the input arguments. Labeled ports are described in detail in section 6.4.4. Finally, the new, single network is returned. The series function has no use for its attributes.

Parallel

The parallel function accepts two arguments. It takes the two networks, and connects the first external port of the first argument to the first external port of the second argument. It then connects the second external port of the first argument to the second external port of the second argument. Therefore, the external ports of each argument remain the same and no new circuit nodes are created. Figure 6.14 illustrates this.

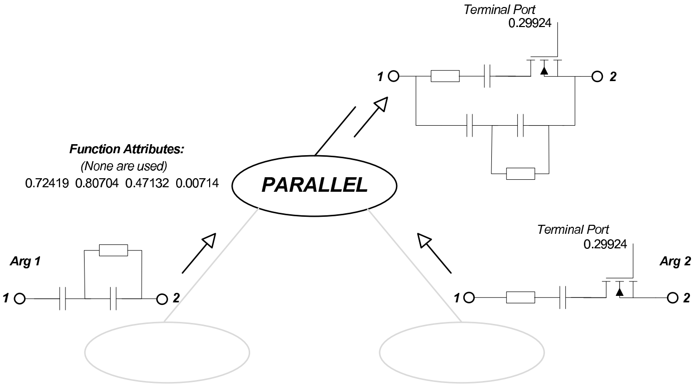


Figure 6.14: Parallel Construction Function

The new network has the dangling terminals and labeled ports of both the input arguments. Finally, the new, single network is returned. The parallel function has no use for its attributes.

Branch

The branch function accepts two arguments. It connects the first external port of the second argument to one of the available connection points in the first argument depending on the value of the first attribute of that function. The available connection points may be the external ports or one of the dangling component terminals if present. If the second argument is connected to a dangling terminal rather than an external port, a single new circuit node is created. Only this function, the series function and the circuit finalisation process create new circuit nodes.

The second external port of the first argument remains unconnected. As a result, additional dangling component terminals are created. The second attribute of this function is assigned to these new, dangling component terminals. The first external port of the new function is merged with the first external port of the second argument. Figure 6.15 illustrates the branch function.

The new network has the dangling terminals and labeled ports of both the input arguments, plus the new dangling terminals created by this function. These new dangling terminals are referred as *loose ends* and are treated differently to other dangling terminals during circuit finalisation. Finally the new, single network is returned. The branch function uses its first and second attributes but no others.

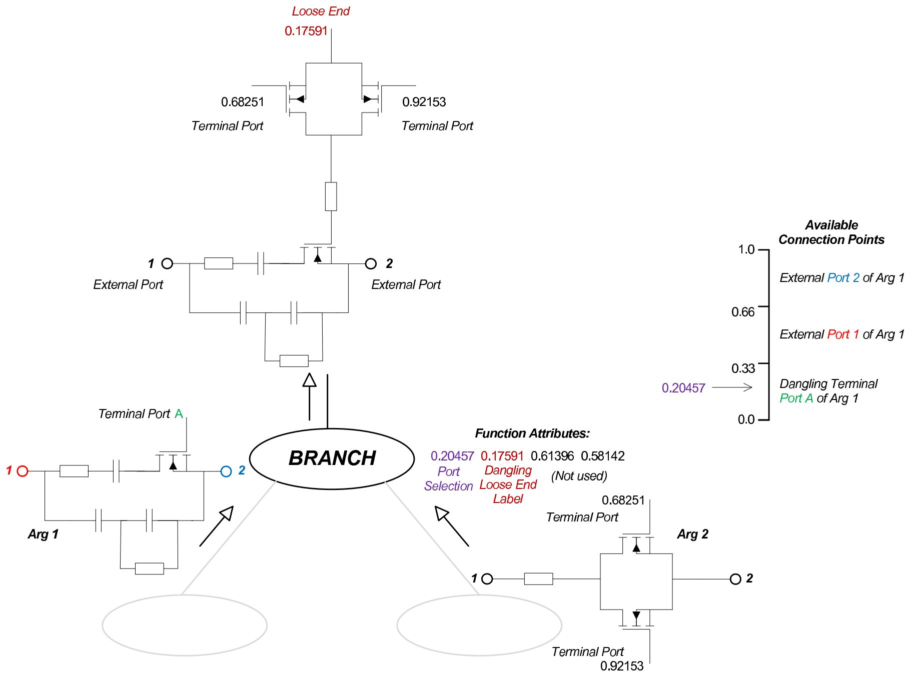


Figure 6.15: Branch Construction Function

Rotate

The rotate function accepts only one argument. It relabels external ports and dangling terminals. The first external port of the argument becomes a dangling terminal if there are any in the network, and one of the dangling terminals will become the new first external port. The same thing happens with the second external port. Figure 6.16 illustrates this.

The external port labels are shuffled along the list of dangling terminals in the order they were created. The first attribute determines the direction of rotation. The external ports may also simply be flipped depending on the value of the first function attribute. The rotate function uses its first attribute but no others.

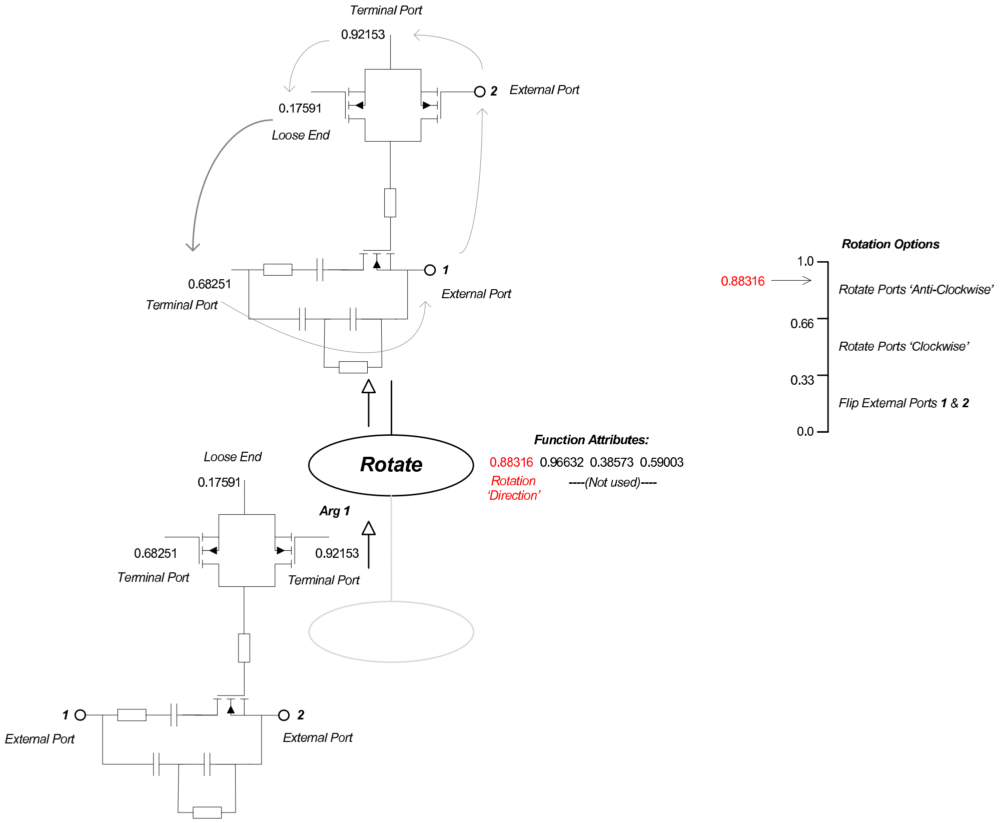


Figure 6.16: Rotate Construction Function

Port

The port function accepts only one argument. The network itself is returned physically unmodified, but one of its external ports is marked, or *labeled* with the first function attribute as being able to accept connections from dangling terminals or loose ends (circuit branches) during circuit finalisation. Figure 6.17 illustrates the port function. The port function uses its first attribute but no others.

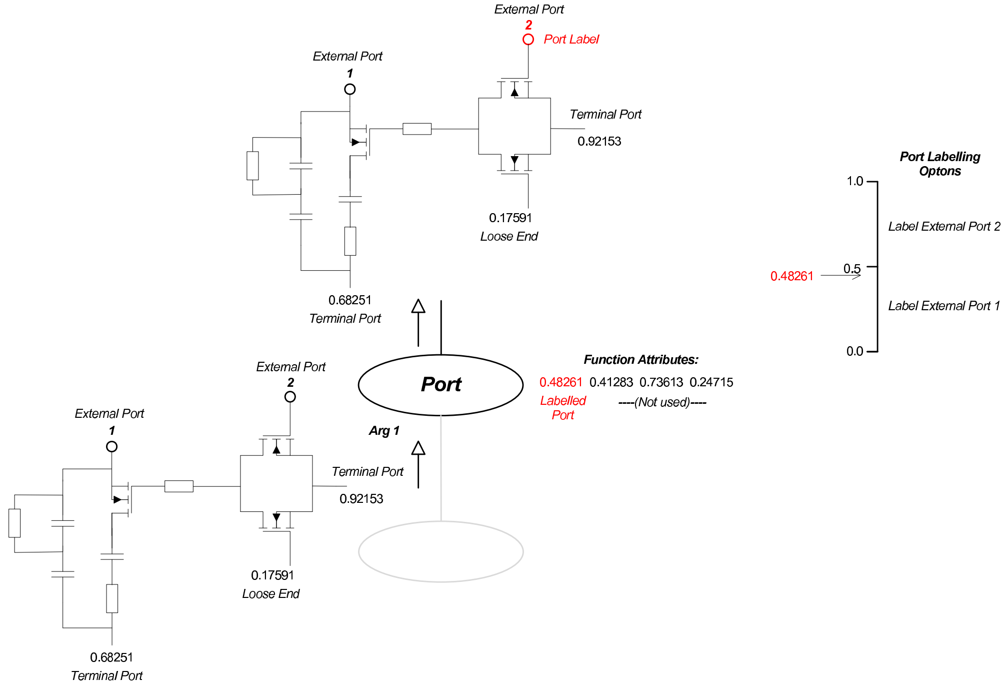


Figure 6.17: Port Construction Function

Functions of Other GP-Based Systems

The function set described here is quite small compared to both Koza's GP implementation [27] for analogue circuit synthesis, and also Sripramong [50]. Koza's system is, ultimately, a LISP program. The component creating functions in this system make use of an 'arithmetic-performing' subtree to return a value of a component. In addition, each component creating function points to a current 'highlighted' component or modifiable wire and inserts its component into that position in the forming topology, even if a component already exists there. Therefore, these component creating functions do not necessarily need to exist only at the end of tree branches. New modifiable wires may also be created during the execution of these circuit constructing functions, in contrast to the system here which uses modifiable wires merely as place holders in the circuit embryo. Sripramong's GP system behaves in a similar way, but does not in fact involve the execution of a LISP program, and therefore does not make use of arithmetic-performing subtrees to determine component values.

Both systems also make use of functions which join distant parts of the circuit. Koza's system uses *pair_connect* functions to achieve this, while Sripramong use *cross-links*. The system presented in this chapter does not use an explicit function to achieve this, but the connection of dangling component terminals during the circuit finalisation stage (section 6.4.4) can have the same effect. The *branch* function may also have a *similar* effect, but will place at least one component between distant parts of the circuit rather than merging two circuit nodes.

Sripramong's system also makes use of a *gcell* function which inserts a gain stage into the circuit being constructed. This is in addition to the user defined subcircuit library, which provides a collection of useful circuit stages which may be inserted into the circuit, although it is not clear if there are any restrictions on the size of type of subcircuits which may be used. Subcircuits of arbitrary size and complexity may be defined in the GA system presented in this chapter.

6.4.3 Tree & Circuit Embryo Organisation

One of the first things any GA does when it starts running is to randomly generate a population of individual candidate solutions. In this case, a population of trees is randomly generated. Starting with a single, randomly picked function, the trees are generated in a top-down fashion. The first function may accept either one or two arguments. If it accepts two, a new branch in the tree is opened up. Left-most branches are always traversed first and this leg of the tree will be generated before the right-hand path.

The tree grows downwards in this fashion until a component function is picked. Component functions accept no arguments, and can only be (indeed, must be) at the very end of every tree branch. There is also a limit of the initial depth of a tree, which is specified in the main configuration file. If any branch of the tree reaches this depth, a component function is always picked and any remaining branches in the tree that have yet to be grown are generated.

In addition to limiting the *initial* size of trees, there is universal cap on the maximum size of any tree which is user controllable. This is important for preventing enormous trees from developing during the run of the GA. Crossover can very easily result in a tree that is deeper than its parents. Indeed, there is a tendency for this to happen. The mechanism for limiting tree size during crossover is described in section 6.5.2.

Tree structure is also influenced by the circuit embryo. There is one tree, or perhaps more accurately: *sub-tree*, for every modifiable wire in the circuit embryo. This is best visualised as shown in figure 6.18.

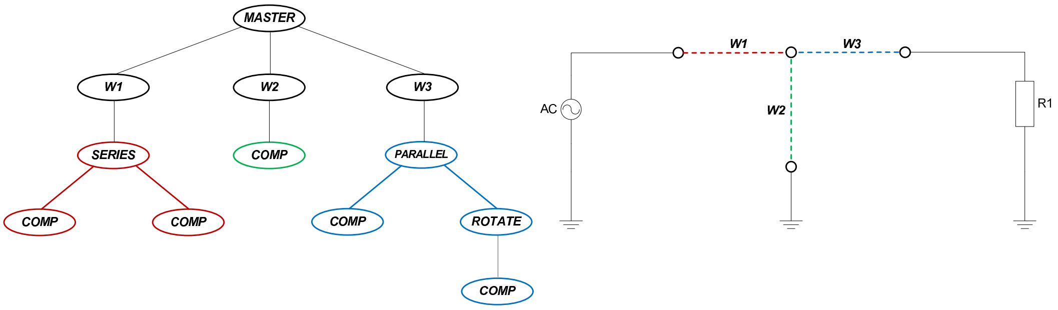


Figure 6.18: Multi-Wire Circuit Embryo

6.4.4 Circuit Finalisation

After each subtree has been decoded into a two-port network, the final phase of circuit decoding must be performed: each two-port network must be fitted into the embryo. A connection for every dangling component terminal is found, based on attributes inherited by that component when it was created by its *component* circuit construction function or passed through the *branch* circuit construction function.

Before the procedure is described, several terms must be defined or restated. *Loose ends* are created by the *branch* construction function, they are ends of circuit branches. In reality they are simply dangling component terminals. *Labeled ports* are circuit nodes which have been tagged as possible connection points for dangling component terminals. *Terminal ports* are simply dangling component terminals not brought about by the *branch* function. A two-port network will have terminal ports if it contains any components which have more than two terminals.

Dangling components or circuit branches in one subtree may be allowed to connect to available connection points in another subtree. Each subtree has a list that specifies which other subtrees its dangling terminals may connect to. This list is defined as part of the circuit embryo definition.

The embryo may contain nodes which are tagged as possible connection points for dangling terminals. These are referred to as *embryonic ports*. Power supplies in the embryo might often be tagged as such. Certain nodes in the embryo may also be tagged as requiring termination if they are left dangling after circuit finalisation is complete.

Circuit finalisation occurs in a series of steps:

1. Additional labeled ports are created at both ends of each modifiable wire in the embryo.
2. Each subtree is 'slotted' into the embryo - its two external ports are assigned the circuit nodes of its corresponding modifiable wire in the embryo.
3. For each subtree, a list is made of all available connection points for dangling terminals. This list will include connection points in other subtrees that the current subtree is allowed to connect to. A complete list of connection points will include all labeled ports and terminal ports visible to each subtree.
4. All loose ends are tied off. Each subtree is iterated over, and each loose end is connected. Loose ends will have an associated attribute, inherited from the *branch* function which created it[1]. This attribute is used to select one the available connection points. If the attribute is less than 0.5, one of the embryonic ports will be selected, using the value of the attribute to make the selection. If the attribute is greater than 0.5, a connection point is picked from the list generated in step 3. If the loose end is connected to a terminal port, a new circuit node will be created, and that terminal port will be removed from the list.
5. Finally, all terminal ports in each subtree are tied off. Each port will have an associated attribute, which was inherited from the *component* function which created the component[2]. If this attribute is less than 0.5, one of the connection nodes in the embryo will be selected, using the value of the attribute to make the selection. If the attribute is greater than 0.5, a connection point is picked from the list generated in step 3 (and possibly reduced in size by step 4). If the terminal port is connected to another terminal port, a new circuit node will be created.

[1] The *rotate* function may redistribute inherited attributes.

[2] Again, the presence of the *rotate* functions means that a dangling component terminal may not actually end up with the attribute it was given when created by the *component* function.

After these steps have been run, the result will be a complete circuit, free from any dangling terminals. Koza [27] uses an implementation which usually connects the third terminal of a transistor to a 'global' circuit node or a power supply. The circuit finalisation phase presented here allows for a much greater number of connection points within each circuit that extra component terminals may connect to.

At the conclusion of decoding each circuit, Sripramong [50] uses a component pruning technique to remove redundant components. *Current-flow analysis* is used to identify components that may be removed. This is perhaps somewhat against the spirit of GAs, although it can reduce simulation time when measuring the fitness of the circuit.

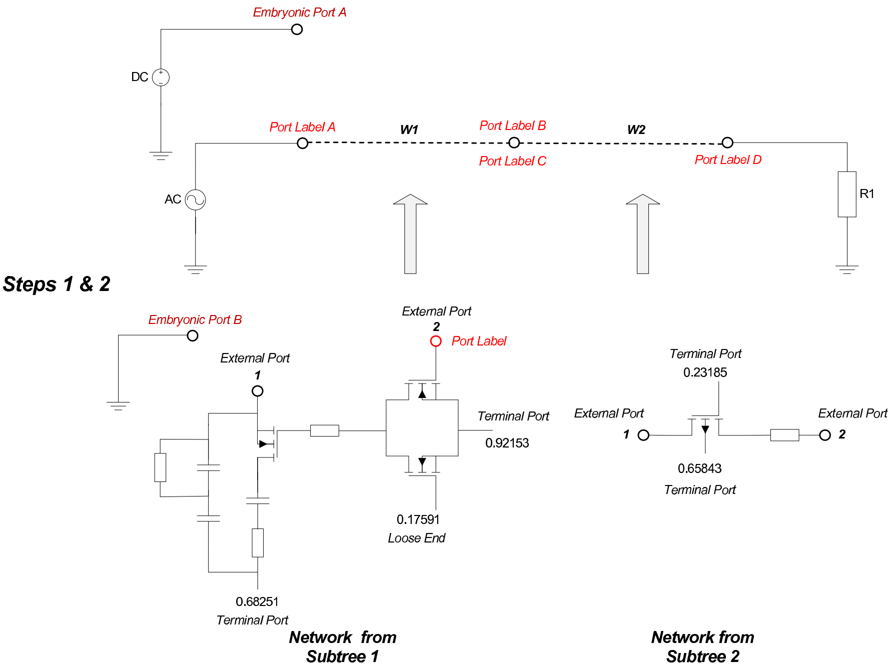


Figure 6.19: Steps 1 & 2 of Circuit Finalisation

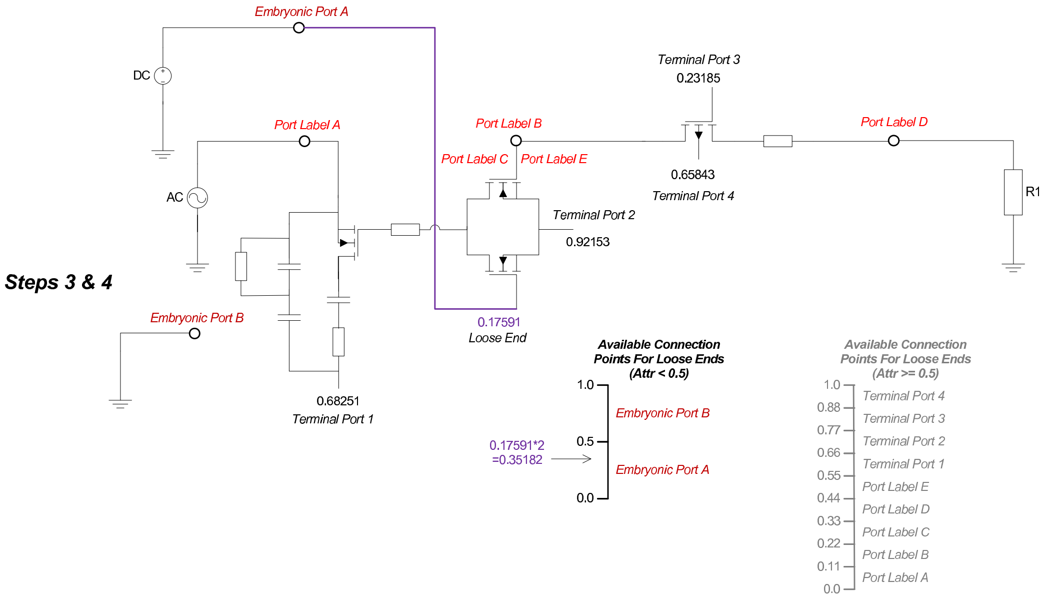


Figure 6.20: Steps 3 & 4 of Circuit Finalisation

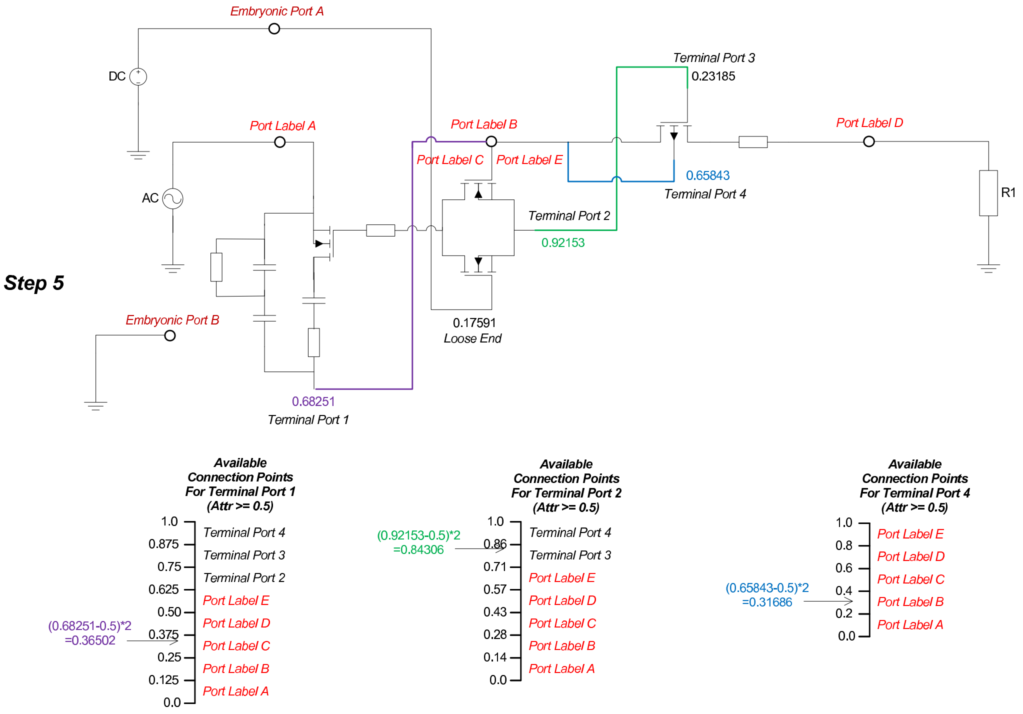


Figure 6.21: Step 5 of Circuit Finalisation

6.5 Genetic Operators

There are 4 genetic operators used in this GA system. The selection, crossover and mutation operators are common to every GA, but here they have been tailored to work with the tree encoding scheme. A fourth operator is also used - predation. This ‘culls’ a certain percentage of the population at each generation and replaces those individuals with randomly generated new trees.

6.5.1 Selection

There are three different selection operators used in this system - proportional, linear rank and exponential rank. The purpose of the selection operators is to choose which individuals will be made available for possible crossover. All three selection operators must assign each tree a particular probability of being selected based on its *fitness*. There are three types of fitness used in this GA system. The first, raw fitness, is the fitness score returned from the fitness function being used. There is more than one fitness function which can be used in this system, and they may return scores of very different magnitudes. However all fitness functions return measures of *error*, so a lower score is better.

Linear rank selection and exponential rank selection, as their names suggest, both depend on the population being placed into order of fitness. As a result, the *adjusted fitness* (equation 6.2) is needed. This simply inverts the score so that a higher score is better. The *normalised fitness* is shown in equation 6.3, where P is the size of the population. The normalised fitness re-scales the score range so that the sum of all fitnesses is unity.

$$f_{adj}(i) = \frac{1}{f_{raw}(i)} \tag{6.2}$$

$$p_i = \frac{f_{adj}(i)}{\sum\limits_{x=1}^{P} f_{adj}(x)} \tag{6.3}$$

Proportional Selection

Proportional selection is the simplest of the selection operators. The probability of an individual being selected for crossover is directly proportional to its fitness, compared to the total fitness of the population. The normalised fitness of each individual is directly used as the probability of being selected. The population is not required to be ranked in order of fitness.

Linear & Exponential Rank Selection

Both linear and exponential rank selection operators are available as a user selectable option. These operators have been implemented exactly as described in section 4.6.4.

6.5.2 Crossover

The crossover function is the most important function in a genetic algorithm. It allows mixing of the structures in a GA. Since tree structures are being used to encode the candidate circuits, the crossover operator must be tailored to them.

Rather than selecting points of binary or integer strings at which to cut, points linking two functions in a tree must be selected. One point is selected in each tree (at random). The subtrees spanning out from these points are then joined onto the opposite parent, creating two offspring in the process, as shown in figure 6.22. Unlike string encoding, the points in each tree can be different. This does not cause a problem. If identical parents are chosen, the offspring will still be different from either parent unless exactly the same cutting point is picked in both trees.

Selection of the cutting points within each tree may be restricted under certain conditions. There is a maximum size to which the trees are allowed to grow. This is necessary since there is a tendency for the trees to balloon in size if unconstrained. Therefore, only cutting points are allowed which will not create a tree bigger than a maximum specified size. The size is specified in terms of depth. Starting from the base of the tree (level 0), the depth of the tree is defined as the number of functions encountered in the longest branch of the tree.

This can mean that if two parents of maximum depth are selected for crossover, then the choice of cutting points may be very restricted. To allow greater freedom in choice of cutting points, there are two types of crossover that may be performed: 'two-way crossover' and 'one-way crossover'. In two way crossover, two new offspring are created from two parents. Both offspring form part of the next generation. In one way crossover, only one new offspring is

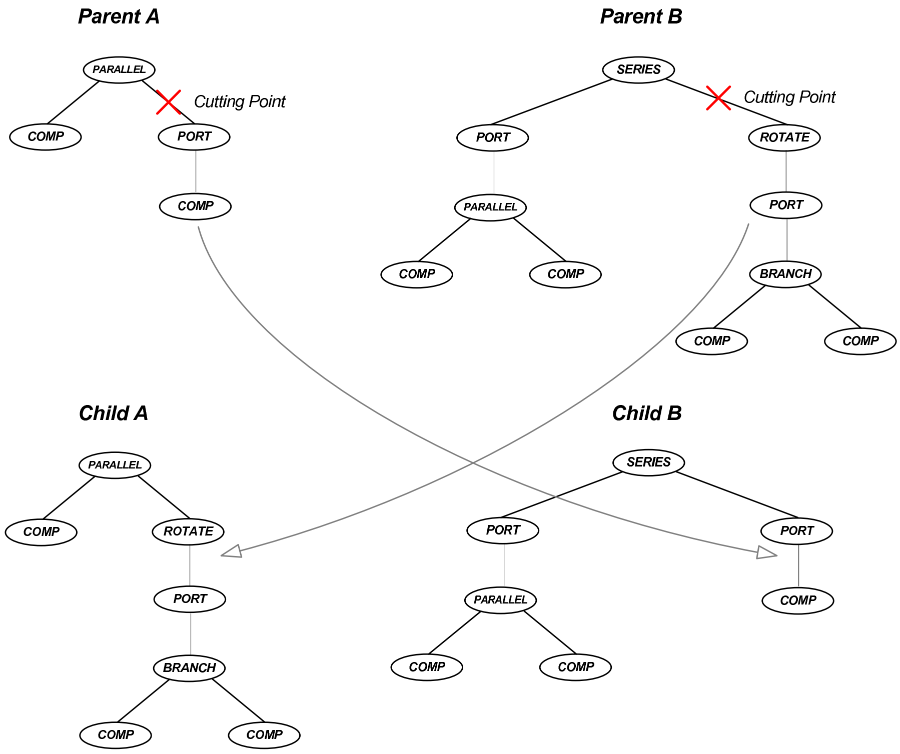


Figure 6.22: GA Tree Crossover

produced, which goes through to the next generation along with the second parent.

In two way crossover, the cutting point in parent one is selected first. The allowed cutting points in parent two are then determined based on the position of cutting point 1. This is because two constraints have to be met simultaneously. The severed subtree from parent one must create a new tree once joined onto parent two that is less than or equal to the maximum depth, and vice versa.

However, in one way crossover, after the cutting point in parent one is selected, allowed points in parent two are determined without regard for how big the offspring would be formed from the severed subtree from parent one, joined onto the cutting point of parent two. This offspring will not be created, only the offspring of the severed subtree from parent two joined onto parent one. One way crossover allows smaller offspring to be created from two maximum sized parents.

Crossover is applied at a certain rate, typically only 65% of breed pairs are selected for crossover. However, the crossovers that do take place may be weighted between one way and two way crossover. Two way crossover is usually applied at a much higher rate. The global crossover rate, and crossover type weighting are user controllable.

6.5.3 Mutation

Mutation is quite a simple operator, but is important for introducing new material into the population and thereby maintaining diversity. It randomly changes parts of randomly selected individuals.

There are two types of mutation used in this GA system: construction function mutation and attribute mutation. Function mutation randomly picks a new function type for a randomly selected construction function within a randomly selected individual, but with constraints. A two argument function may only be replaced with another two argument function. This means that if the function picked for mutation is a PARALLEL function, it can only become a SERIES, BRANCH or PARALLEL function. The new function type is allowed to be the same as the old type, in which case the mutation effectively did not happen. Likewise, a PORT function can only be replaced with a another PORT function, or a ROTATE function. This is to preserve the structure of the tree. COMPONENT functions can only ever be replaced by another COMPONENT function. Again, this is to ensure that the tree is still valid, so that it can still be decoded to produce a circuit.

Attribute mutation randomly selects one of the attributes in a randomly selected construction function, in a randomly selected individual, and replaces it with a new attribute. There are no restrictions placed on attribute mutation. The type of mutation to be carried out is determined randomly, but is based on a user selectable weighting.

The mutation operator used by Koza [26] is very different to that used here. Rather than tweaking aspects of each tree, the approach used by Koza's system is to randomly pick a tree in the population to delete. A new tree is then randomly grown in its place. In fact, it is much more like this system's predation operator (section 6.5.4), however the selection of the tree which will be deleted is not based on its fitness. This form of mutation is not considered particularly useful by Koza and it is not heavily used by that system.

6.5.4 Predation

Predation is the process of culling trees and replacing them with new ones. This happens after the whole population has been measured and a complete set of fitness scores has been determined. The selection operator is applied, assigning each tree a probability of selection.

The probability of being picked for predation is essentially an ‘inverse’ of the selection probability. The functions of figures 4.8 and 4.9 are flipped laterally to produce probabilities for predation. In a population of 500, the tree in rank 500, for example, would have its probability swapped with the tree in rank 1. Tree rank 400 would be swapped with tree rank 100. This means that the worse the fitness for a particular tree, the higher the chance it will become prey and be removed from the population. Predation only works with linear and exponential rank selection in the GA system presented here. Figure 6.23 shows the probability of predation in the case of exponential rank selection, for a population of 500.

Each new tree grown is limited by a maximum initial depth, which can be different from the initial depth used when the population is first generated but must be no greater than the absolute maximum depth. The new tree’s fitness is measured, and the selection operator is re-applied in order to rank the population with the new trees added and new probabilities are assigned. This operator can be applied in two ways. Normally, when a new tree is re-inserted into the population, it is available for culling along with all of the other trees. If *single cull* predation is switched on, then the newly inserted trees are not available for culling.

6.6 Candidate Circuit Characterisation

The GA system uses HSPICE [8] in order to produce a circuit response which can be measured by the fitness function. The SPICE commands in the *spice options file* (section 6.3) are written out into each SPICE netlist. HSPICE then performs the specified simulation. The results are parsed and fed into the fitness function.

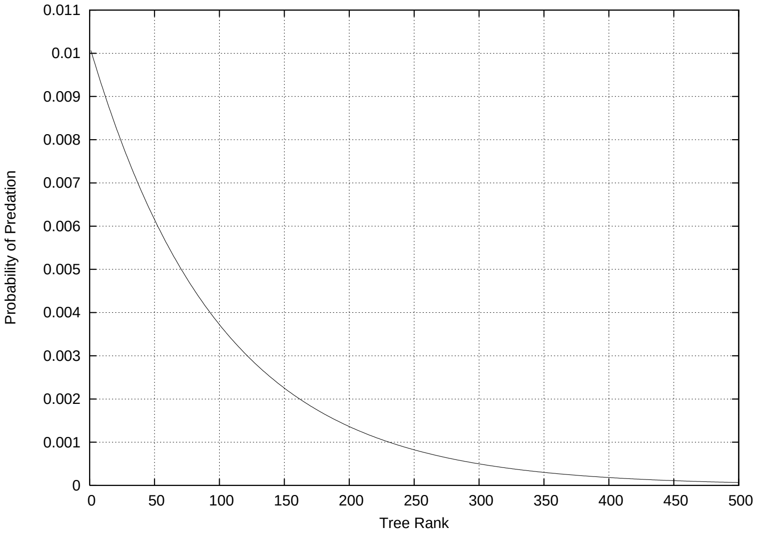


Figure 6.23: Probability of Predation Based on Tree Rank

6.7 Fitness Functions

The fitness function reads in simulation data, output from spice, for each candidate circuit. It returns a raw fitness score which is used by the selection operators. Selection operators are described in sections 4.6.4 & 6.5.1.

Genetic algorithms traditionally use *shape fitting* fitness functions. The target circuit characteristics in the time or frequency domain are usually passed to the GA as a curve or waveform. The GA then performs a simulation on each individual to generate the same type of data. The individual and target waveforms are then compared in some way and a measure of difference produced.

The GA system being discussed here has two fitness functions which may be used. The first is, very simply, a measure of the absolute error between the two circuit characteristics. The second is a completely novel fitness function. It is based on pole-zero analysis, and a set of poles and zeros are fed into the GA as target data.

If an individual circuit cannot be measured for any reason, it is awarded a standard fitness score S_{abort}. This score can be set by the user, but it should be very high so that it is treated by the GA as a very poor individual.

6.7.1 Shape-fitting

The shape fitting fitness function used in this GA system is exactly as described in section 4.7 (equation 4.15 and figure 4.10).

6.7.2 Pole-Zero

Rather than measuring a circuit's characteristics in the time or frequency domains, pole-zero analysis characterises circuits in the s-domain. Measuring the fitness of a circuit in terms of its poles and zeros is a different approach to traditional GA fitness functions. It compares the poles and zeros of the candidate circuit to the poles and zeros of the target circuit. When comparing two sets of poles and zeros, two metrics can be defined as a measure of similarity: the difference in number of poles and zeros, and the *distance* between the location of poles and zeros. More information about the pole-zero analysis of circuits can be found in [49].

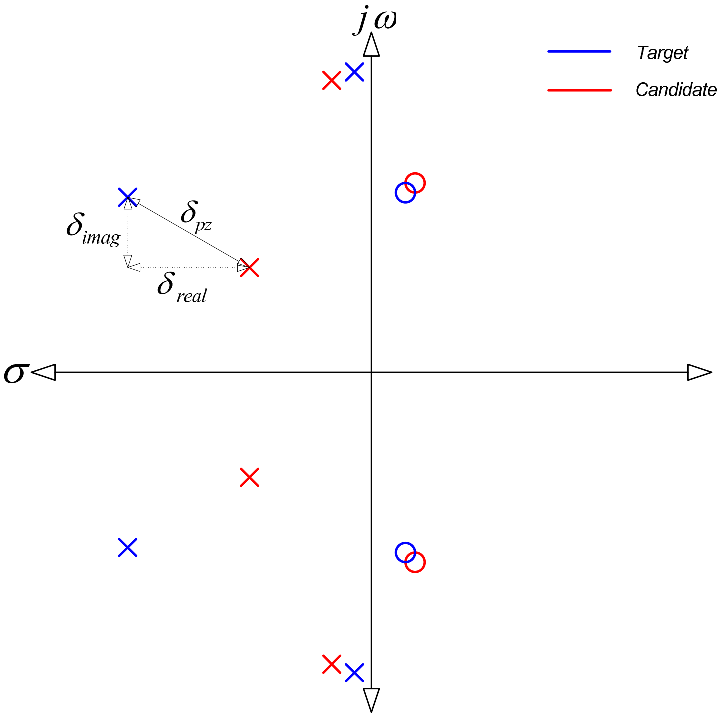


Figure 6.24: Measuring the distance between target and candidate poles (crosses) and zeros (circles).

The fitness function must return a score which captures both of these metrics, and expresses some indication of *error* between the poles and zeros of the two circuits. The final score returned by this function is made up of two parts (equation 6.4) - the *bulk* score (S_{bulk}) which is determined by the difference in number of poles and zeros (this will be referred to as *count error)*, and the *detailed* score ($S_{detailed}$) which expresses how close the candidate and target poles and zeros are. This difference in location will be referred to as the *distance error.*

$$S_{total} = S_{bulk} + S_{detailed} \tag{6.4}$$

Bulk Score

When comparing two circuits in the s-domain, by far the most significant difference between them will be in their numbers of poles and zeros. If these are different, it is of relatively little concern how close the poles and zeros that are there are to each other. Therefore, the bulk score is designed to dominate, and that any circuit which has even a difference of just one in count error but low distance error is awarded a score which is worse than a circuit which has zero count error, but high distance error. Or, stated another way, *total distance error must always be less than a non-zero count error.*

Equation 6.5 defines the bulk score, where δ_{poles} is the absolute difference in the number of poles and δ_{zeros} is the absolute difference in number of zeros.

$$S_{bulk} = S_{basic}\delta_{poles} + S_{basic}\delta_{zeros} \tag{6.5}$$

S_{basic} is the *basic score.* This number is user selectable, its main purpose is to produce a score which is a little more human 'readable' than it would otherwise be. Setting this number to one would have no impact on the effectiveness of this fitness function, however it would require the human user to deal with very small real numbers when looking at the output reports of the GA and also when selecting a tolerance (or stop criterion) value for the GA.

Detailed Score

A detailed score of each pole-zero pair of the candidate and target circuits is calculated. The $S_{detailed}$ term in equation 6.4 is the sum of all of these scores, as shown in equation 6.6, where num_{pz} is the total number of pole-zero pairs, and $S_{pz\_detailed_x}$ is the detailed score of each pole-zero pair. $S_{pz\_detailed_x}$ is the product of k_{fit_x} and max_{pz} (equation 6.7).

$$S_{detailed} = \sum_{x=0}^{num_{pz}} S_{pz\_detailed_x} \tag{6.6}$$

$$S_{pz\_detailed_x} = k_{fit_x} max_{pz} \tag{6.7}$$

The maximum allowed score for the distance error of each pole and zero (max_{pz}) is found using equation 6.8, where n_{poles} is the number of poles and n_{zeros} is the number of zeros. It is important to point out here that when the distance error scores for each pole and zero are added, the total is less than would be awarded to a circuit that had a difference in count error of just one. A count error of one would result in a bulk score of S_{basic} (equation 6.5).

$$max_{pz} = \frac{S_{basic}}{(n_{poles} + n_{zeros})} \tag{6.8}$$

The value of k_{fit_x} in equation 6.7 is determined by the distance error. This error is calculated for each target-candidate pole-zero pair very simply by using Pythagoras (equation 6.9).

$$\delta_{pz} = \sqrt{\delta_{real}^2 + \delta_{img}^2} \tag{6.9}$$

When comparing two very dissimilar circuits, the location of target and candidate poles and zeros may be very far apart. This introduces the obvious problem of *which distance errors*, between *which target-candidate pole-zero pairs* should be used in the calculation of the detailed score. In other words, each pole and zero in the target circuit must be assigned a corresponding pole or zero in the candidate circuit. This is achieved by calculating the distance error for every possible pairing of target and candidate pole or zero. The pairings with the lowest scores are always selected first.

When a distance error for each pole-zero pair has been determined, k_{fit_x} can be calculated. Equation 6.10 determines k_{fit_x}, which is the proportion of max_{pz} that will make up the distance score. The distance score coefficient curve is shown in figure 6.25. The curve is made up of two distinct regions. There are two variables in equation 6.10 which control its shape. P_{dist} is the distance at which the curve transitions from linear to non-linear. If the distance is less then $P_{dist}+1$, the curve is linear. If it is greater than $P_{dist}+1$, the curve very quickly flattens off. The two regions of the curve are designed to reward small increases in pole-zero distance much more heavily once the pole-zero locations start getting close. Small increases are rewarded much less when the locations are far apart. The curve approaches the value max_{pz} but never exceeds it. The other variable, P_{lin}, determines which values of k_{fit_x} are in the linear region. Both P_{dist} and P_{lin} are controllable by the user.

$$k_{fit} = \begin{cases} (\frac{P_{lin}}{P_{dist}+1})\delta_{pz} & \delta_{pz} < P_{dist}+1 \\ (1-P_{lin})e^{(\frac{-10}{log_{10}(\delta_{pz}-P_{dist})})} + P_{lin} & \delta_{pz} \geq P_{dist}+1 \end{cases} \tag{6.10}$$

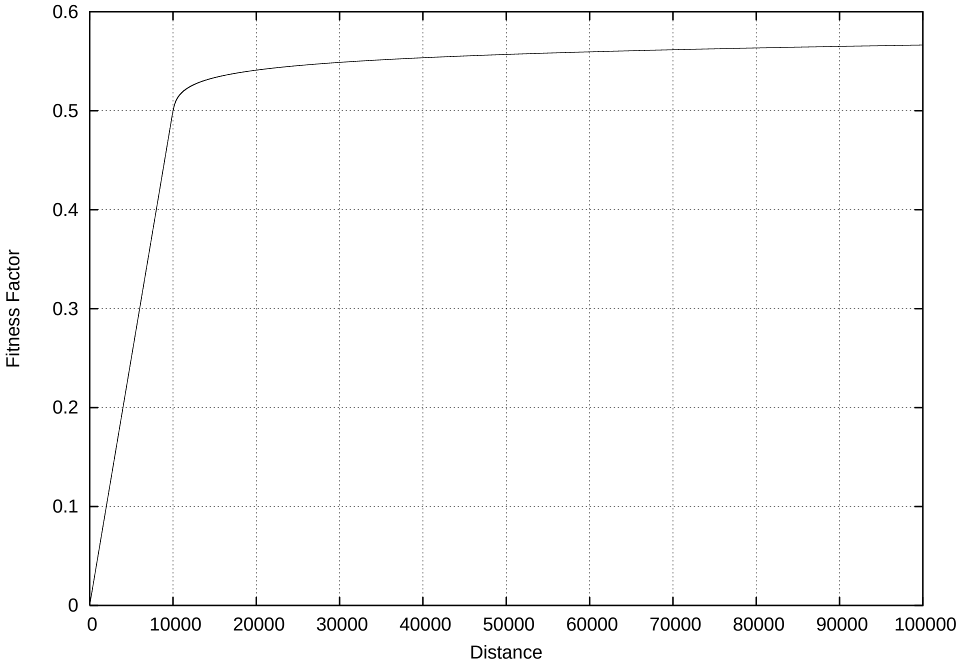


Figure 6.25: Distance Score Coefficient Curve for $P_{dist} = 10000$ & $P_{lin} = 0.5$

This fitness function is designed to award fitness scores primarily in terms of differences in the number of poles and zeros (S_{bulk}). This contributes the majority of the fitness score and is intended to be useful in the early stages of a GA run when many circuits will be unsuitable and have incorrect numbers of poles and zeros. The much smaller contribution that the difference in the *location* of poles and zeros makes ($S_{detailed}$) may still serve to differentiate between circuits which incorrect pole and zero numbers. After the GA has executed enough generations, the population is likely to be dominated by circuits with the correct pole and zero numbers and at this point only the distance between poles and zeros is of concern. The condition in equation 6.11 is always satisfied.

$$S_{detailed} \leq S_{basic} \tag{6.11}$$

A fitness function which takes into account differences in pole and zero numbers can directly apply selection pressure to the topology of a circuit. For example, the number of reactive components in certain types of RLC filters will be equal to the number of its poles.

6.7.3 Component Count

In addition to both of the fitness functions already described, the number of components in the circuit is also taken into account. In addition to the fitness score already returned by the fitness function, a score of S_{comp} is added to the total if the circuit contains just a single component, excluding the embryo. If more than half the components in the circuit are resistors, then S_{res} is added to the total fitness score (equation 6.12, where P_{res} is the proportion of components in the circuit which are resistors).

$$S_{res} = S_{comp}P_{res} \tag{6.12}$$

This extra fitness score is awarded in order to discourage single component circuits, or circuits dominated by resistors. The shape-fitting fitness functions sometimes have a tendency to produce all resistor circuits. The value of S_{comp} is usually set high, although lower than S_{abort}.

6.8 GA System Outputs

The primary output of the GA system is the synthesised analogue circuit in the form of a spice netlist, if it has converged successfully. The corresponding tree of that circuit is also reported. If it did not converge, then the closest circuit found will be output, again with its corresponding tree. In both cases, every time a new, better circuit is found then that circuit is also output.

A series of running reports are also produced as the GA runs. A history of the fitness of each new, best circuit found is recorded, as well as the average and peak fitness of each generation. The time taken by each generation is also recorded. A record is also kept of the number of circuits in each generation whose *spice simulations* did not converge or that could not be successfully simulated. In addition to this, if a circuit did not simulate, the circuit netlist and simulation output file is retained. This allows a detailed per-generation analysis of the types of simulation error seen during the course of the GA run.

Chapter 7

CASE STUDIES

This chapter presents five series of experiments using the Genetic Algorithm based analogue synthesis system described in chapter 6. Firstly, a series of *Initial Experiments* examines the evolution of some simple Chebyshev filters. This to test the ability of the GA system to synthesise the most basic circuits and to provide a reference point to compare subsequent experiments to. A series of *GA Sensitivity Experiments* then investigates the effects of varying the GA control parameters such as population, crossover rate and so on.

The strengths and weaknesses of the novel pole zero fitness function are then investigated through a series of *Fitness Function Comparison Experiments*. This section examines the evolution of some circuits using both the pole zero and shape fitting fitness functions. Next, the results of synthesising some more complex circuits are presented from a series of *Active Filter Synthesis Experiments*. Finally, the effects that the genetic operators have on circuit function are examined in a series of *Topology Mutability Experiments* which help to explain some of the results seen in the previous experiments.

7.1 Initial Experiments

The results of three initial experiments are presented in this section. The evolution of some simple analogue filters is important in showing that the GA system presented in chapter 6 is at all capable of synthesising analogue circuits, as well as providing a reference point for the following experiments. In all three experiments, the evolution of a lowpass Chebyshev filter is attempted: 4^{th}, $5^{\text{th}}$and then 6^{th}order. The cut-off frequency in each case is 1kHz. More information about passive RLC filters can be found in [57].

Chebyshev filters are characterised by their frequency response; they have a tell-tale ripple in the pass band and are flat in the stop band. Passive Chebyshev filters can be constructed using only resistors, capacitors and inductors and in this form they have a simple 'ladder' topology. This type of passive filter is already well understood and data tables are available that allow circuit designers to pick suitable component values for a given set of circuit specifications such as cut-off frequency, pass band attenuation, magnitude of voltage ripple and so on. In reality, it is unlikely that a synthesis system such as the one presented in this text would be used to design such a filter.

However, the small component set required and simple topology makes them an ideal candidate for these initial synthesis experiments. Not only are these circuits unlikely to make great demands on the synthesis system, but the fact that this circuit type is well understood makes the quality of synthesis results easy to evaluate. Discussion of Experiments 1, 2 and 3 is deferred until section 7.1.4.

7.1.1 Experiment 1: 4$^{\mathrm{th}}$Order Chebyshev Filter

The first experiment carried out is the evolution of a 4$^{\mathrm{th}}$order Chebyshev filter. The target frequency and phase responses are shown in figures 7.3a. The target circuit itself is shown in figure 7.1. The GA was supplied with the circuit embryo shown in figure 7.2, which consists simply of a signal source and a loading resistor on the output. It serves as a test harness and contains absolutely no predefined topology. A total of five repetitions of this experiment were carried out.

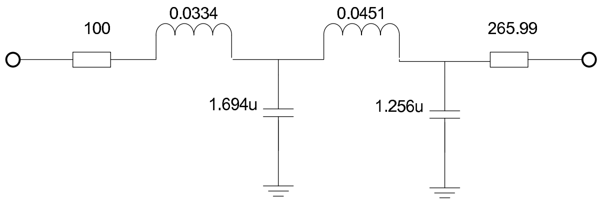


Figure 7.1: Ideal 4th Order Chebyshev

Component Set

The following components were made available to the GA for this experiment:

Resistors	$100\Omega \rightarrow 100K\Omega$
Inductors	$1pF \rightarrow 1mF$
Capacitors	$0.1\mu H \rightarrow 1H$

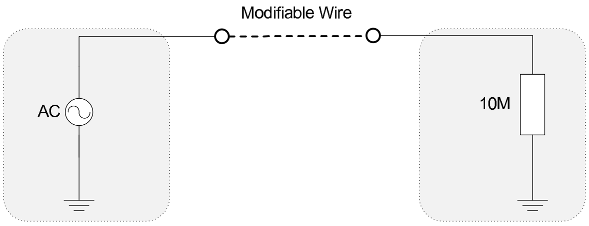
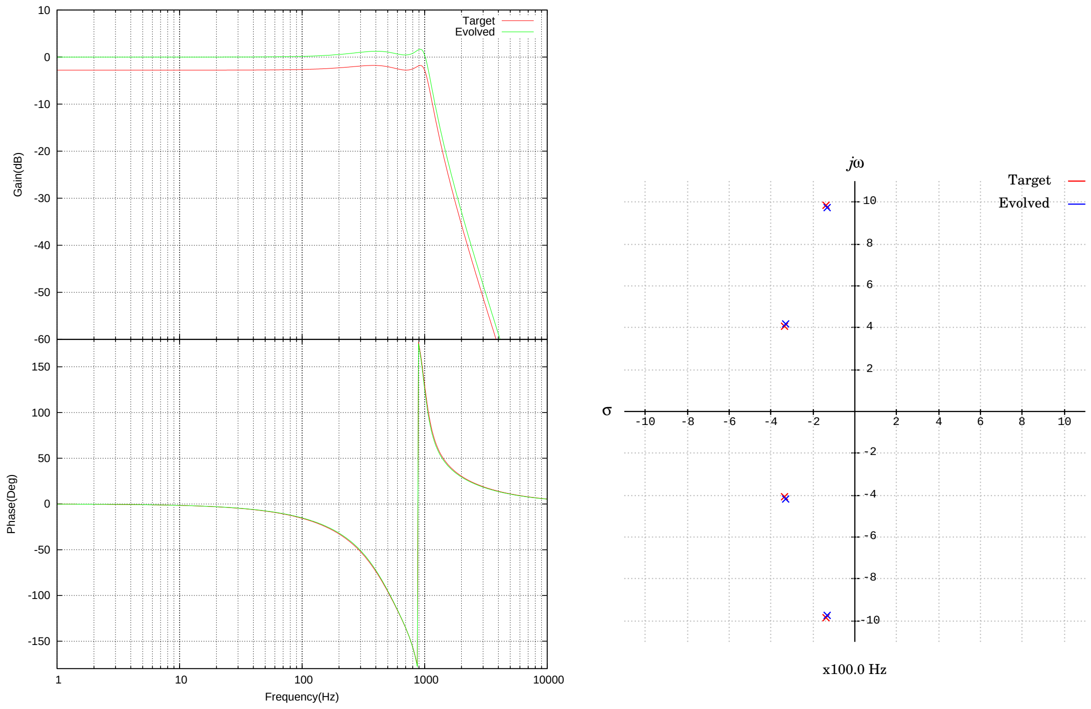


Figure 7.2: Embryonic Circuit

Parameters

The GA was run with the parameters shown below. The tolerance value, E_M was set at 40. There are a total of 4 poles in the target circuit. The maximum tree depth has been set to 5, which will give a possible maximum of 16 components.

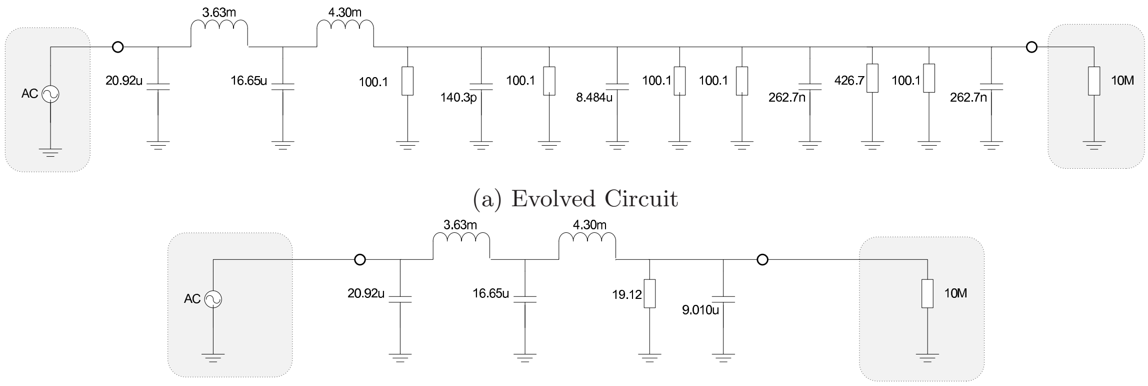
Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 40$
Probability of Mutation	$p_m = 0.5$ *(0.9 Weighting)*
Probability of Predation	$p_p = 0.05$
Probability of Crossover	$p_c = 0.65$
Initial Tree Depth	$T_i = 3$
Max Tree Depth	$T_M = 5$
Max New Tree Depth	$T_N = 5$
Crossover Type	*Mixed (0.8 Weighting)*
Selection operator	*Exponential Rank*
Fitness Function	*Pole-Zero*
Pole-Zero Parameters	$P_{lin} = 0.5$, $P_{dist} = 5000$
SPICE time-out	$t_{SPICE} = 60s$



(a) Target & Evolved Frequency & Phase Response

(b) Target & Evolved Pole Zero Plot

Figure 7.3: Target & Evolved Circuit Characteristics Of 4$^{\text{th}}$Order Chebyshev Filter



(a) Evolved Circuit

(b) Equivalent Evolved Circuit

Figure 7.4: Actual & Equivalent Evolved Circuits

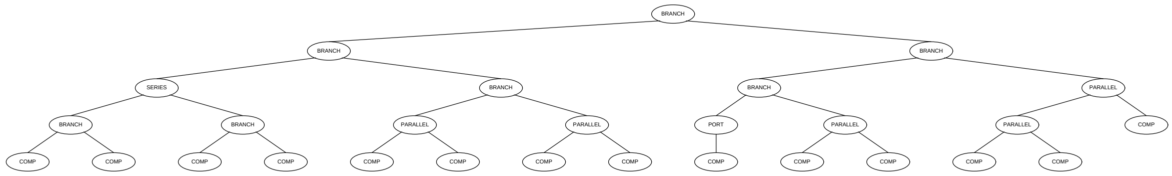
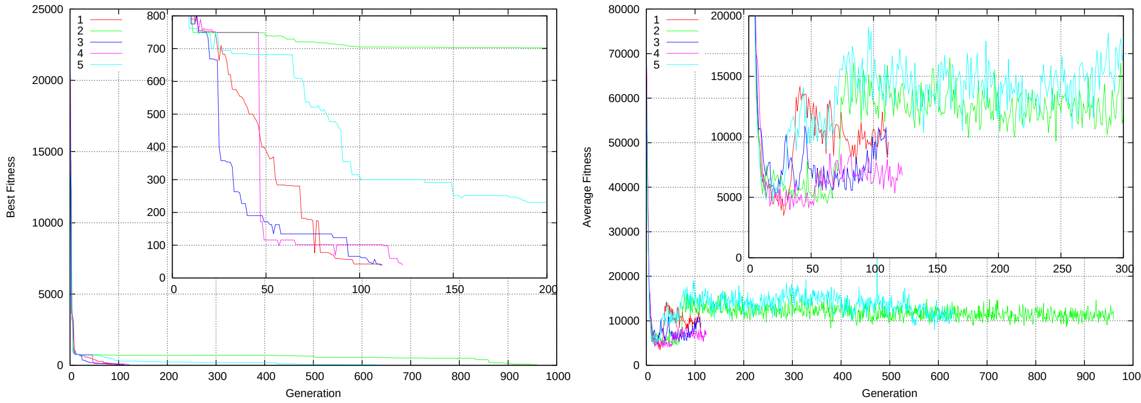


Figure 7.5: Encoding Tree Of Evolved Circuit

GA Behaviour

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
1	112	Yes	39.24
2 (Best)	960	Yes	31.31
3	112	Yes	38.65
4	123	Yes	39.15
5	632	Yes	35.56

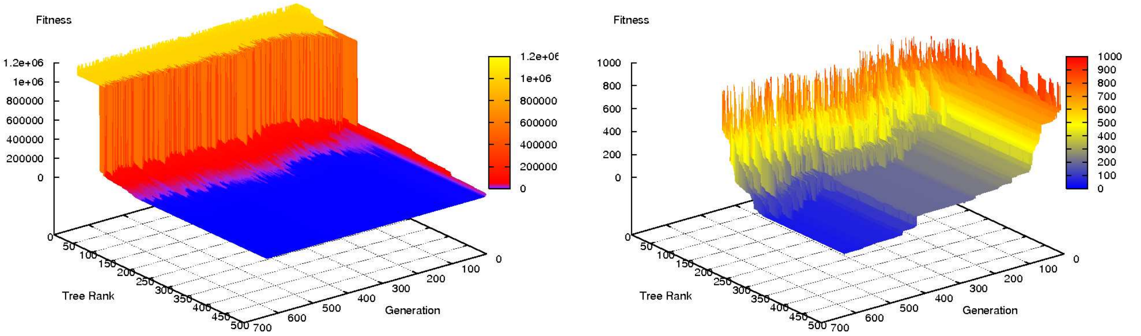
Table 7.1: Synthesis Results Summary



(a)

(b)

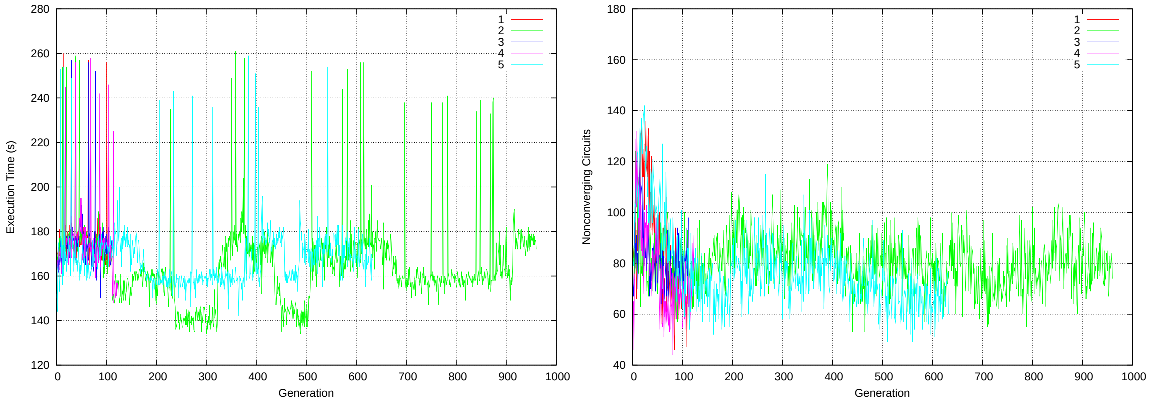
Figure 7.6: Peak & Average Fitness Graphs of Five Runs of 4$^{\text{th}}$Order Chebyshev Filter



(a) Total population.

(b) View zoomed to point of convergence (lower left corner at front).

Figure 7.7: Population Over Total Run of Experiment



(a) (b)

Figure 7.8: Time & Non-Convergence Graphs of Five Runs of 4^{th}Order Chebyshev Filter

7.1.2 Experiment 2: 5th Order Chebyshev Filter

This experiment attempts the evolution of a 5^{th}order Chebyshev filter. The target frequency and phase responses are shown in figures 7.10a. The target circuit itself is shown in figure 7.9. The GA was supplied with the circuit embryo shown in figure 7.2. The component set used is the same as in experiment 1.

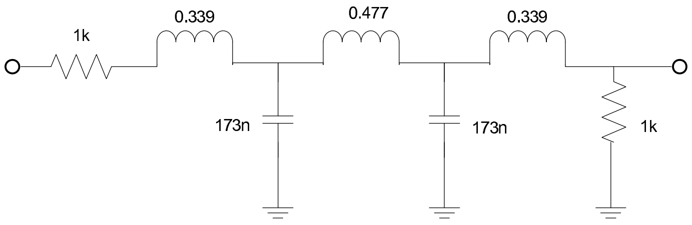
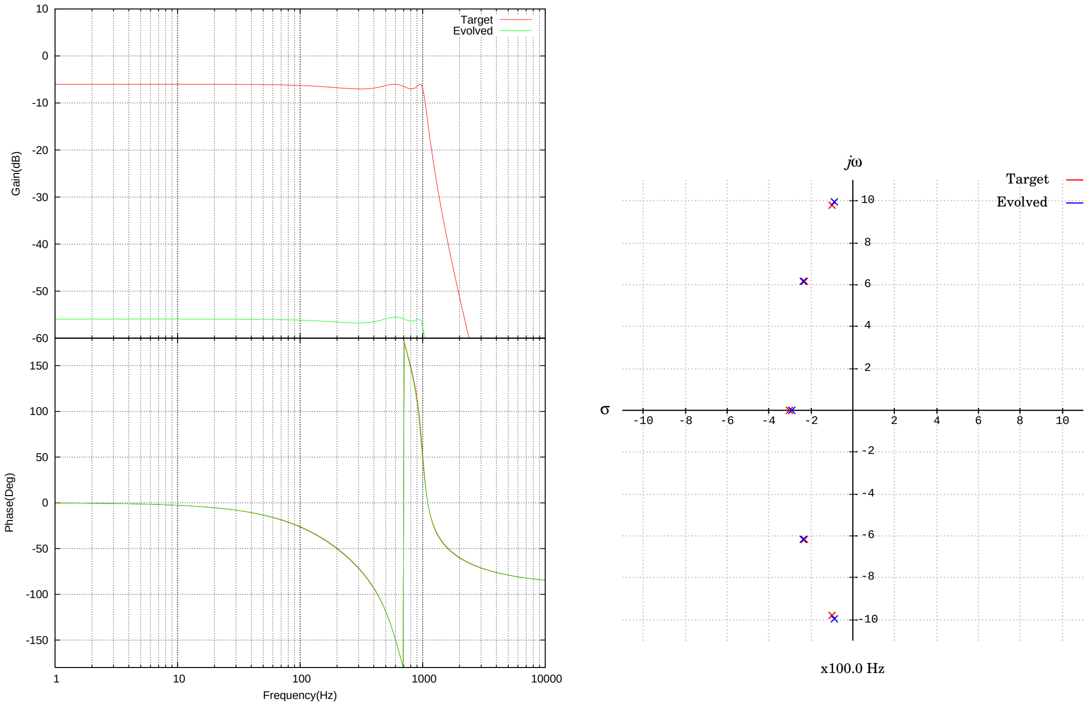


Figure 7.9: Ideal 5th Order Chebyshev

Parameters

The GA parameters used in this experiment are the same as in experiment 1, with the exception of the tolerance score. The target circuit has a total of five poles and no zeros, so the tolerance has been set to 50.

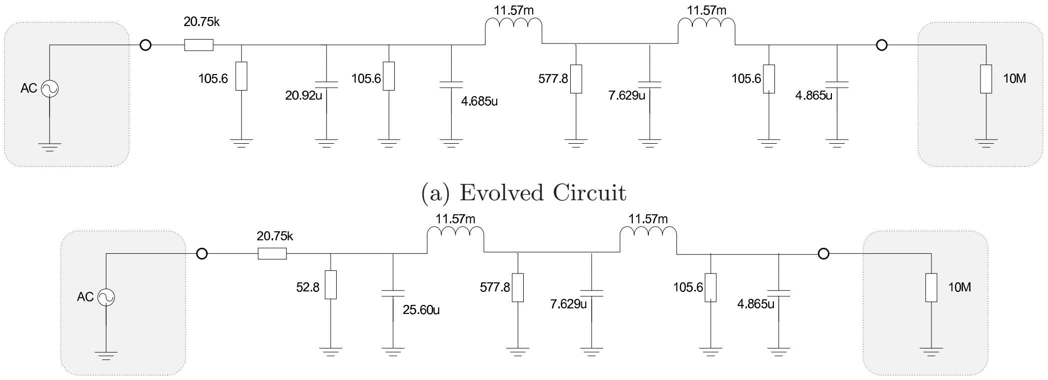
Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 50$



(a) Target & Evolved Frequency & Phase Response

(b) Target & Evolved Pole Zero Plot

Figure 7.10: Target & Evolved Circuit Characteristics Of 5thOrder Chebyshev Filter



(a) Evolved Circuit

(b) Equivalent Evolved Circuit

Figure 7.11: Actual & Equivalent Evolved Circuits

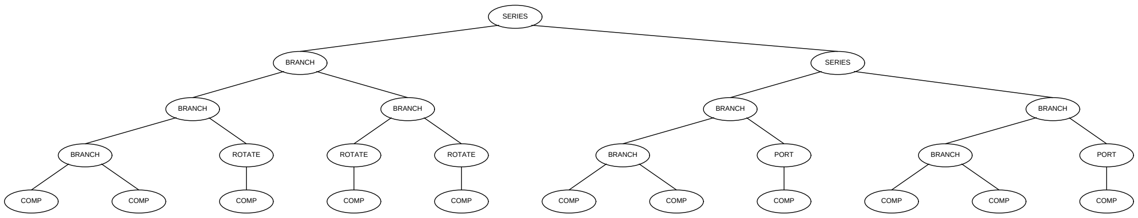
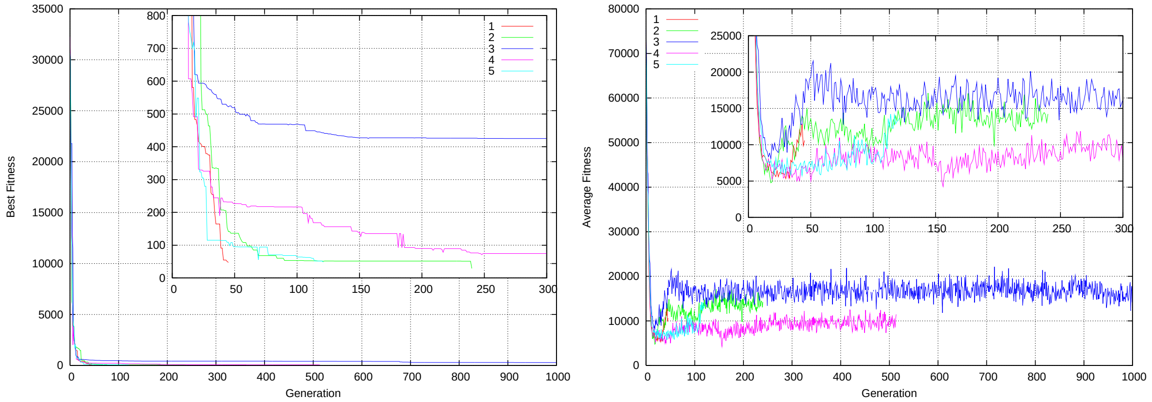


Figure 7.12: Encoding Tree Of Evolved Circuit

GA Behaviour

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
1	45	Yes	47.49
2 (Best)	240	Yes	29.16
3	1000	No	285.67 (989)
4	514	Yes	47.17
5	121	Yes	48.48

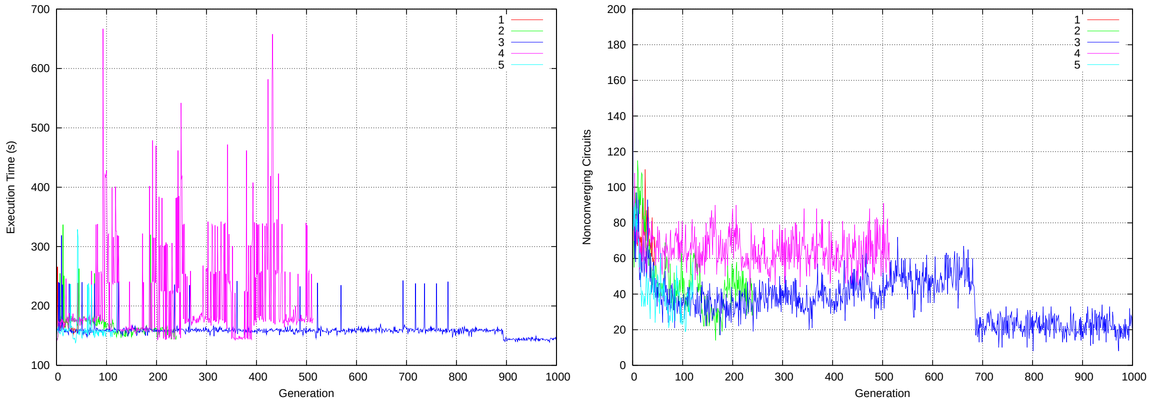
Table 7.2: Synthesis Results Summary



(a)

(b)

Figure 7.13: Peak & Average Fitness Graphs of Five Runs of 5th Order Chebyshev Filter



(a)

(b)

Figure 7.14: Time & Non-Convergence Graphs of Five Runs of 5$^{\text{th}}$Order Chebyshev Filter

7.1.3 Experiment 3: 6th Order Chebyshev Filter

This experiment attempts the evolution of a 6$^{\text{th}}$order Chebyshev filter. The ideal frequency and phase responses are shown in figure 7.16a. The ideal circuit itself is shown in figure 7.15. The GA was supplied with the circuit embryo shown in figure 7.2. The component set used is the same as in experiment 1.

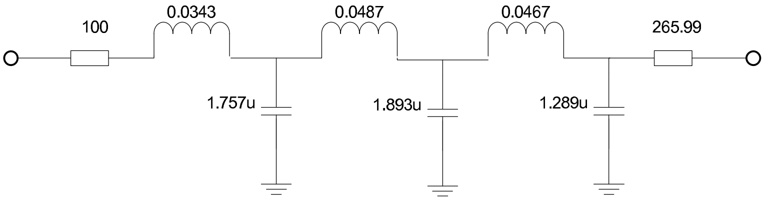
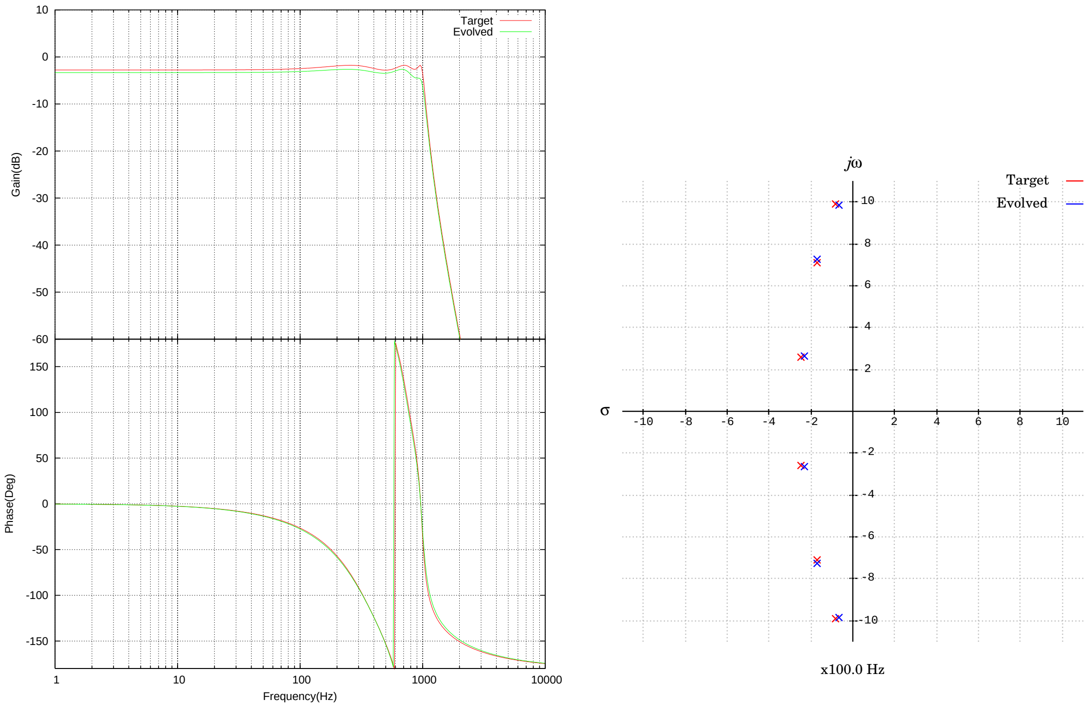


Figure 7.15: Ideal 6th Order Chebyshev

Parameters

The GA parameters used in this experiment are the same as in experiment 1, with the exception of the tolerance score. The target circuit has a total of six poles and no zeroes, so the tolerance has been set to 60.

Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 60$



(a) Target & Evolved Frequency & Phase Response

(b) Target & Evolved Pole Zero Plot

Figure 7.16: Target & Evolved Circuit Characteristics Of 6$^{\text{th}}$Order Chebyshev Filter

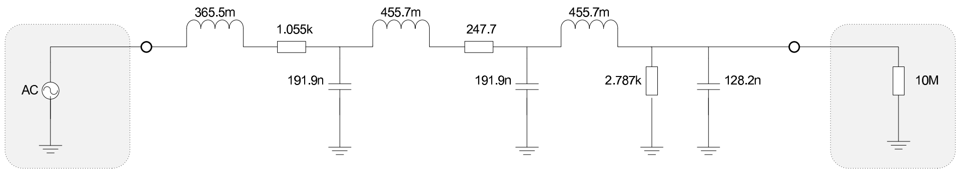


Figure 7.17: Evolved Circuit

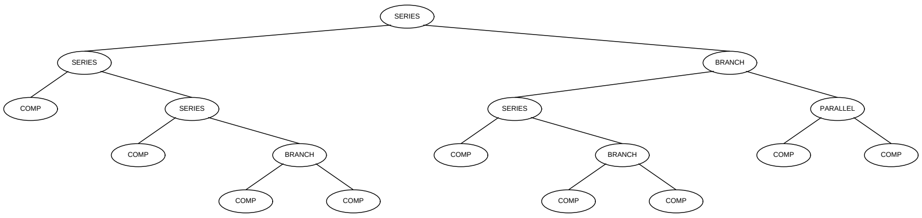
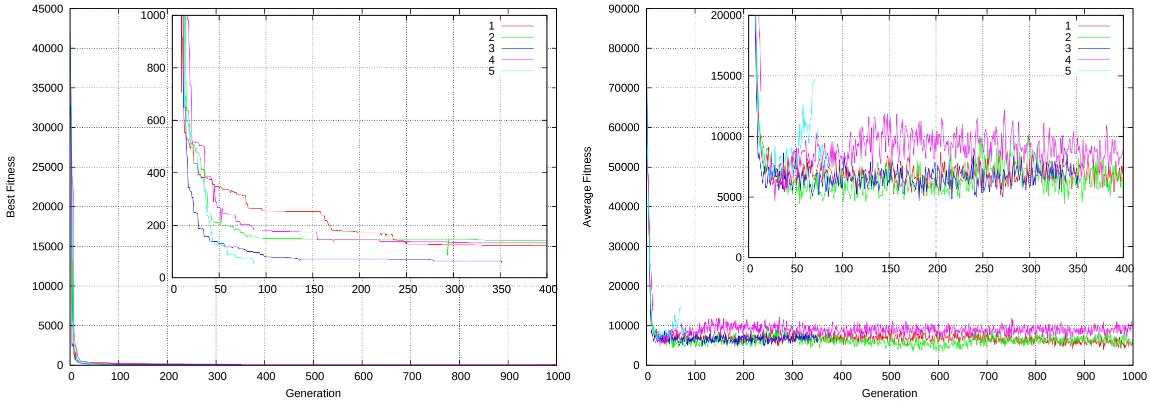


Figure 7.18: Encoding Tree Of Evolved Circuit

GA Behaviour

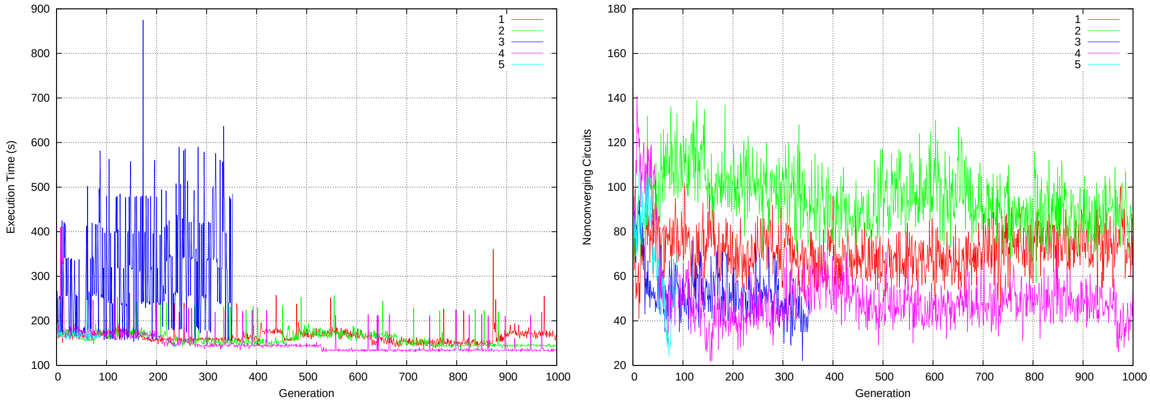
GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
1	1000	No	114.957 (996)
2	1000	No	85.23 (294)
3	352	Yes	56.78
4	1000	No	131.81 (980)
5 (Best)	87	Yes	52.01

Table 7.3: Synthesis Results Summary



(a) (b)

Figure 7.19: Peak & Average Fitness Graphs of Five Runs of 6th Order Chebyshev Filter



(a) (b)

Figure 7.20: Time & Non-Convergence Graphs of Five Runs of 6th Order Chebyshev Filter

7.1.4 Discussion of Initial Experiments

The GA converged many times during experiments 1 - 3 and in each experiment produced filters with the correct numbers of poles and zeros. All of the filters exhibited frequency responses that were of the correct shape. However, the most notable feature of these responses was the pass band attenuation. The 6$^{\text{th}}$order filter was the closest to the target in this respect. It should be noted that the frequency response plot of the 4$^{\text{th}}$order filter is not accurate. Despite being a passive circuit, the response shows that the gain is positive at certain frequencies. This is an AC sweep simulation error, which could not be corrected by increasing accuracy settings. This will *not* have affected the GA itself, as this circuit was evolved using the pole-zero fitness function. The AC sweep was only attempted after the circuit was evolved. The filter that stands out the most, however is the 5$^{\text{th}}$order filter. This exhibited *very* high attenuation in the pass band. The pole and zero locations determine the attenuation only of the network of reactive components. The cause of the difference amongst the three filters was the resistors they contained which acted as voltage dividers. The 5$^{\text{th}}$order circuit has a high value resistor (20.75kΩ) on the input.

The locations of a circuit's poles and zeros are determined by the values of its reactive components. The value of any resistors within the circuit can also affect them, this is however a relative effect. A given required input impedance will directly determine the value of the input resistor. The values of the reactive components must then be chosen so that their reactance produces the desired frequency response (and hence pole zero locations). This reactance must balance with the input resistance, but this is the extent to which the passive component (resistor) values affect the pole zero location. The poles and zeros themselves do not specify input impedance.

The pole zero fitness function measures only the location of the circuit poles and zeros, and does not explicitly measure absolute attenuation. Therefore, it is not surprising that the actual attenuation of the circuits is essentially random. The *shape* of the frequency response in all three experiments is *very* close to the target.

The reactive component values have clearly been scaled to match the essentially random input impedance chosen by the GA, in much the same way a human engineer would scale the component values for a *required* input impedance. For example, the inductor values in experiments 1 and 3 are very close to the values of their respective target circuits. However, in the case of experiment 2, the input resistor is approximately an order of magnitude greater than the target circuit. The inductor values are therefore approximately an order of magnitude *less* than that target. This is necessary in order to produce poles and zeros in the required locations.

Another notable feature about the structure of these circuits is the number of parallel resistors and capacitors they contain, especially in experiment 1. The *replication* of components in this manner is discussed in section 7.6.7.

The best fitness graphs of all three experiments are very similar. They are mostly monotonic, with only occasional instances of a small worsening of fitness. Fitness typically improves rapidly in steps as topologies with the correct number of poles and zeros are found. The average fitness graphs show data only of circuits whose SPICE simulations converged. In many cases a 'trough' can be seen just after the start of the run. The average fitness quickly falls as the best circuits in the initial random population start to increase in number, but after a relatively small number of generations it often increases slightly. This is most likely due to the circuits increasing in size with increasing generation and the number of resistors contained within them contributing to a higher (worse) fitness score.

It should be noted that a discontinuity was discovered in the pole-zero fitness function which led to enormous spikes in average fitness. The function used to calculate the contribution each pole and zero contributes to the fitness score is given by equation 6.10. There is a discontinuity where the linear and non-linear regions of the functions are joined. If the difference between a target and candidate pole, d_{pz} satisfies $P_{dist} \leq d_{pz} < P_{dist} + 1$, the discontinuity may result in a value several orders of magnitude greater than max_{pz}. There is a spike of infinite value where the two regions join. Originally, the boundary between the two regions was set at P_{dist}, rather than $P_{dist} + 1$, and the data presented in this chapter uses the original boundary. Equation 6.10 shows the corrected function. The experiments were not repeated with the corrected function due to time constraints.

The average fitness spikes have been removed from the graphs as they obscure the rest of the data. However, they are very unlikely to have had a significant impact on GA behaviour. It will effectively treat them as non-convergent circuits due to the very large fitness score as they will be assigned the absolute lowest tree ranks.

The per-generation execution time and non-convergence data from these three experiments show few common characteristics. The most obvious detail are the spikes in the time data. These are mostly likely due to circuits which SPICE has trouble simulating but does not immediately give up on. The maximum time allowed for each SPICE simulation is $t_{SPICE} = 60$s. If SPICE encounters a difficult DC bias point calculation, for example, it will go through a series of procedures which use different methods to attempt to calculate it. In some cases this will take longer than 60 seconds, and so the simulation is aborted by the GA before it can complete. It would only take two or three of these difficult simulations to add significant time to the execution of each generation.

The number of non-convergent circuits tend to be higher during the first generations. The initial population is entirely randomly generated, and so there is likely to be a high proportion of circuits that cannot be simulated for a variety of reasons. As better schemata are found and dominate the population, the number of these circuits will decrease.

Both the time and SPICE non-convergence data for all three experiments are highly erratic, with obvious periods of higher or lower average values. As the GA progresses, the nature of the circuits that dominate the population will change and may be easier or harder to simulate.

7.2 GA Sensitivity Experiments

The initial three experiments give assurance that the GA synthesis system being investigated is at least capable of generating some basic circuits. They also provide a point of reference for the following experiments.

It is useful to gain an understanding of how well tuned the control parameters of a GA need to be in order to have any chance of successfully synthesising an analogue circuit. This section presents the results of a series of experiments that examine the effects of varying each control parameter of the GA in turn.

All the experiments described in this section attempt to evolve the same circuit as in Experiment 2 - a 5$^{\text{th}}$order chebyshev filter. The GA uses the Pole-Zero fitness function, with the same target data as used in Experiment 2.

7.2.1 Experiment 4: Varying The Population Size

This experiment examines the effect of varying the size of the GA population. Experiments 1-3 have already shown that a population size of 500 is enough to allow the GA to converge and produce circuits which meet the specified tolerance. Figure 7.21 and table 7.4 show the results for this experiment.

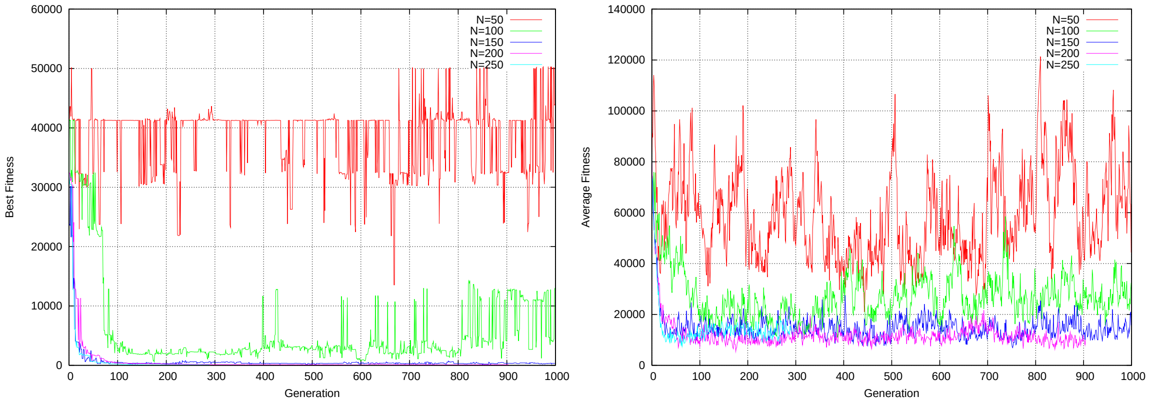
Parameters

The parameters used in this experiment are the same as those used in experiment 2, with the exception of course of the population size. Five repetitions of this experiment were carried out at each parameter value.

Maximum Generations	$G_M = 1000$
Population Size	$N = 50$, $N = 100$, $N = 150$, $N = 200$, $N = 250$
Tolerance	$E_M = 50$
Probability of Mutation	$p_m = 0.5$ *(0.9 Weighting)*
Probability of Predation	$p_p = 0.05$
Probability of Crossover	$p_c = 0.65$
Initial Tree Depth	$T_i = 3$
Max Tree Depth	$T_M = 5$
Max New Tree Depth	$T_N = 5$
Crossover Type	*Mixed (0.8 Weighting)*
Selection operator	*Exponential Rank*
Fitness Function	*Pole-Zero*
Pole-Zero Parameters	$P_{lin} = 0.5$, $P_{dist} = 5000$
SPICE time-out	$t_{SPICE} = 60s$

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
N=50			
1	1000	No	13528.00 (668)
2 (Best)	1000	No	2444.80 (95)
3	1000	No	3111.66 (800)
4	1000	No	4067.32 (665)
5	1000	No	3720.90 (488)
N=100			
1	1000	No	626.12 (175)
2	1000	No	479.18 (488)
3	1000	No	397.85 (700)
4 (Best)	1000	No	367.98 (984)
5	1000	No	587.28 (848)
N=150			
1	1000	No	125.143 (850)
2 (Best)	1000	No	52.30 (595)
3	1000	No	498.66 (991)
4	1000	No	569.00 (736)
5	1000	No	108.60 (879)
N=200			
1 (Best)	904	Yes	43.758
2	1000	No	82.99 (718)
3	1000	No	204.01 (925)
4	1000	No	99.98 (977)
5	360	Yes	49.50
N=250			
1	289	Yes	43.87
2 (Best)	789	Yes	39.70
3	418	Yes	45.88
4	1000	No	55.97 (857)
5	1000	No	56.45 (945)

Table 7.4: Summary of Population Size Experiment



(a)

(b)

Figure 7.21: Peak & Average Fitness Graphs Of Varying Values Of N

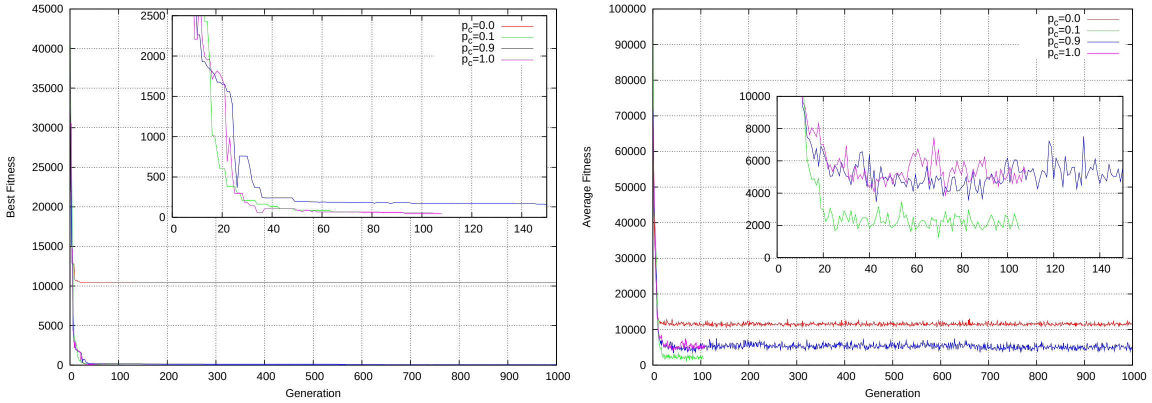
7.2.2 Experiment 5: Varying The Crossover Rate

This experiment examines the effect of varying the crossover rate. Experiments 1-3 have already shown that a probability of crossover of $p_c = 0.65$ will allow the GA to converge. Figure 7.22 and table 7.5 show the results for this experiment.

Parameters

Apart from p_c, the probability of crossover, the parameters used in this experiment are the same as those used in experiment 2. Five repetitions of this experiment were carried out at each parameter value.

Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 50$
Probability of Crossover	$p_c = 0.00$, $p_c = 0.10$, $p_c = 0.90$, $p_c = 1.00$



(a) (b)

Figure 7.22: Peak & Average Fitness Graphs Of Varying Values Of p_c

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
p_c=0.0			
1	1000	No	10407.90 (971)
2 (Best)	1000	No	1242.78 (980)
3	1000	No	21352.40 (665)
4	1000	No	21352.40 (884)
5	1000	No	10102.40 (615)
p_c=0.1			
1 (Best)	105	Yes	48.13
2	1000	No	408.79 (890)
3	224	Yes	49.78
4	1000	No	186.10 (950)
5	1000	No	1648.09 (164)
p_c=0.9			
1	1000	No	71.40 (741)
2	1000	No	54.56 (793)
3	1000	No	84.89 (961)
4	1000	No	104.74
5 (Best)	134	Yes	46.21
p_c=1.0			
1	108	Yes	40.25
2	1000	No	93.77 (990)
3 (Best)	252	Yes	31.89
4	81	Yes	49.60
5	55	Yes	36.52

Table 7.5: Summary of Crossover Rate Experiment

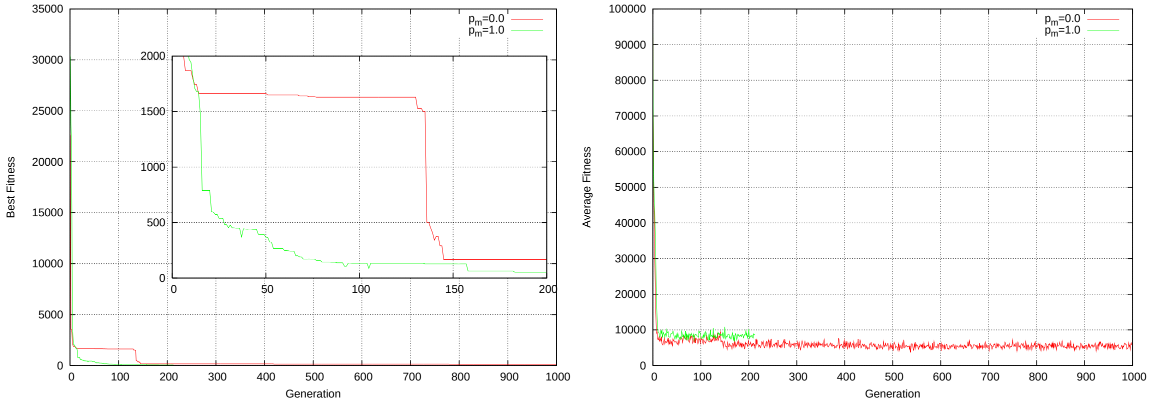
7.2.3 Experiment 6: Varying The Mutation Rate

This experiment examines the effect of varying the size of the mutation rate. Experiments 1-3 have already shown that a probability of mutation of $p_m = 0.50$ will allow the GA to converge. Figure 7.23 and table 7.6 show the results for this experiment.

Parameters

Apart from p_m, the probability of mutation, the parameters used in this experiment are the same as those used in experiment 2. Five repetitions of this experiment were carried out at each parameter value.

Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 50$
Probability of Mutation	$p_m = 0.00$, $p_m = 1.00$



(a) (b)

Figure 7.23: Peak & Average Fitness Graphs Of Varying Values Of p_m

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
p_m=0.0			
1	1000	No	113.30 (797)
2	1000	No	246.04 (344)
3	1000	No	224.56 (747)
4	1000	No	1644.45 (776)
5 (Best)	1000	No	105.29 (896)
p_m=1.0			
1 (Best)	213	Yes	39.97
2	1000	No	263.60 (953)
3	54	Yes	46.09
4	1000	No	56.53 (989)
5	124	Yes	43.25

Table 7.6: Summary of Mutation Rate Experiment

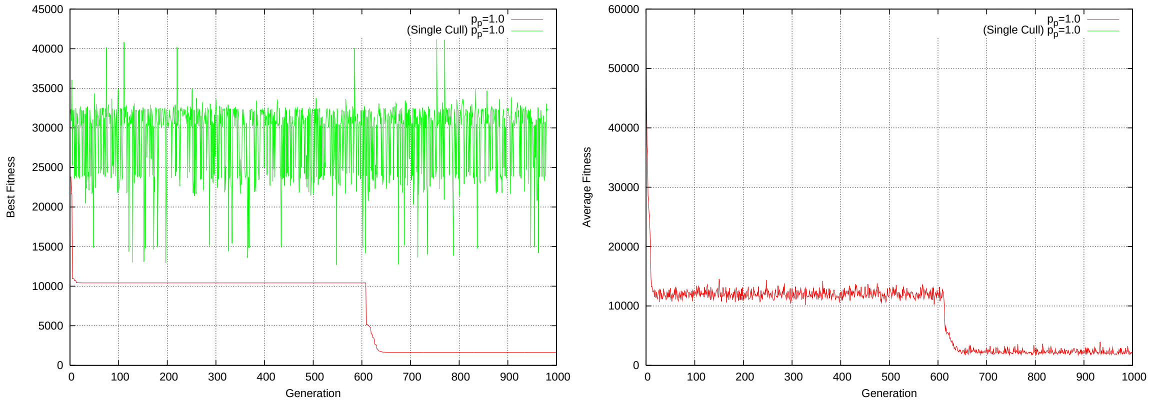
7.2.4 Experiment 7: Varying The Predation Type

This experiment examines the effect of altering the type of predation when the predation rate is set to 100%. Experiments 1-3 have already shown that a probability of predation of $p_p = 0.05$ will allow the GA to converge. Figure 7.24 and table 7.7 show the results for this experiment.

Parameters

Apart from p_p, the probability of predation, and p_c, the probability of crossover (which was set to 0.0), the parameters used in this experiment are the same as those used in experiment 2. Five repetitions of this experiment were carried out at each parameter value. No average fitness data was recorded for the single cull experiment due to a bug in the GA software.

Population Size	$N = 500$
Tolerance	$E_M = 50$
Probability of Predation	$p_p = 1.00$
Probability of Crossover	$p_c = 0.00$



(a) (b)

Figure 7.24: Peak & Average Fitness Graphs Of Varying Values Of p_p

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
p_p=1.0			
1	1000	No	1647.5 (935)
2	1000	No	253.458 (982)
3 (Best)	1000	No	107.713 (818)
4	1000	No	1250.87 (894)
5	1000	No	409.065 (870)
***p_p=1.0*(Single Cull)**			
1	984	No	12695.0 (548)
2	317	No	13070.2 (246)
3	223	No	12791.5 (41)
4 (Best)	100	No	11437.1 (7)
5	100	No	13149.9 (7)

Table 7.7: Summary of Predation Rate Experiment

7.2.5 Discussion of GA Sensitivity Experiments

Population Size Results

Reducing the population size has *very* significant impact on GA behaviour. Experiments 1-3 show that the GA is certainly capable of converging on a solution which satisfies the constraints given to it. However, reducing the population size to $N = 50$ radically alters the GAs behaviour enough to stop it functioning as an incremental search algorithm. The best fitness graphs almost resemble an entirely random search, with good circuits in general *not* being preserved from generation to generation.

The selection operator used in all of the experiments presented here is based on *exponential rank selection* (section 4.6.4). As can be seen from figure 7.25, altering the population size

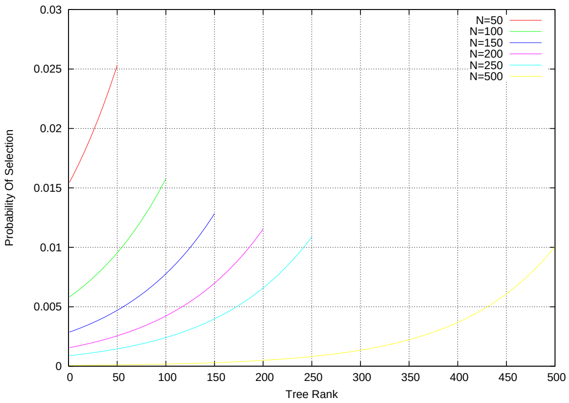


Figure 7.25: Probability Curves For Different Population Sizes (Exponential Rank Selection)

changes the selection probabilities significantly. The summation of the probabilities of the whole population must be 1, and so the trees with the highest ranks are assigned a much greater probability of being selected, however the lowest ranked trees are also assigned a greater probability. It is the *relative* difference between lowest and highest probabilities that is of interest here. Table 7.8 shows the lowest and highest selection probabilities for six values of N and the ratio between them.

	N=500	**N=250**	**N=200**	**N=150**	**N=100**	**N=50**
$p_{i_{max}}$	0.010066	0.010882	0.011547	0.012844	0.015774	0.025317
$p_{i_{min}}$	6.6808E-05	8.9010E-04	0.0015627	0.0028732	0.0058320	0.015472
$p_{i_{max}}/p_{i_{min}}$	150.67	12.22	7.39	4.47	2.70	1.64

Table 7.8: Differences Between $p_{i_{max}}$ & $p_{i_{min}}$ For Varying N

It is important here to remember that the selection operator selects pairs of individuals that may or may not undergo crossover. It executes N times, creating $N/2$ breeding pairs. The proportion of these breed pairs that ultimately undergo crossover will be the crossover probability p_c, with the rest of the pairs $(1 - p_c)$ going through to the next generation *unchanged*.

With decreasing N, the lowest ranked trees have much greater probability of being selected; the highest ranked trees have a much lower probability. There are two consequences to this. The first is that the fittest individuals are far less likely to pass through unchanged to the next generation as there will be a lower proportion of breeding pairs which contain those individuals. And secondly, fit individuals are far more likely to be paired up with a poor individual when crossover does occur.

This also illustrates the destructive effects of crossover in this application. It does not appear as though crossover generally results in gradual, incremental improvement. If so, a fit individual crossed over with an unfit individual would result in an individual with fitness somewhere between the two, and a gradual improvement in both best and average fitness would be expected. Clearly, this is not happening with $N = 50$. An overall improvement in fitness only occurs with higher population sizes, indicating that only when crossover between two fit individuals occurs (this is far more likely with higher N) is there an increase in fitness.

What is really of interest in these results is the *progress* of the GA, in terms of best fitness per generation (figure 7.21), rather than the number of times the GA actually converged. The experiment was run for 1000 generations at each value of N. This means that a different number of circuits were trialled for each N and comparing the frequency of convergence is not a direct comparison. Table 7.9 shows a summary of the fitness values of the best circuits found in the first 50,000 trialled circuits for each run - this many circuits were trialled during 1000 generations with $N = 50$.

Population Size	***N=50***	***N=100***	***N=150***	***N=200***	***N=250***
Generations needed to trial 50,000 circuits	**1000**	**500**	**334**	**250**	**200**
GA Run					
1	13528.00 (688)	626.12 (175)	246.18 (226)	167.07 (136)	54.22 (175)
2	2444.80 (95)	479.18 (488)	248.01 (277)	98.10 (145)	225.15 (183)
3	3111.66 (800)	755.62 (461)	4327.91 (14)	413.22 (229)	107.35 (198)
4	4067.32 (665)	676.34 (454)	595.63 (331)	103.50 (180)	208.83 (192)
5	3720.90 (448)	780.34 (115)	305.74 (240)	96.59 (221)	415.93 (184)

Table 7.9: Best fitness values found in population size experiments after 50,000 circuits had been trialled. Generation in which circuit was found is shown in brackets.

There is still clearly a marked difference behaviour at $N = 50$. The best fitness values are much worse at this population size. However, *none* of the experiments converged within the 50,000 trialled circuits.

Crossover Rate Results

Completely disabling crossover also significantly impacts the GA. In four of the five runs carried out, after 1000 generations, no circuit was found that even had the correct number of poles and zeros. The best fitness quickly reached a minimum and remained flat.

The results of the other three settings of p_c show typical GA behaviour, very similar to what was observed in Experiments 1-3. There is not enough data to draw any firm conclusions about the effect that p_c has on the GA, other than that it does play an important role and crossover does need to happen at some rate. Not enough time was available to allow more data to be generated. A summary of the run time of each experiment is given in Appendix A.

A degradation in GA behaviour may have been expected when the crossover rate was set to 100%. However, mixed crossover was used in this experiment with a weighting of 0.8, meaning that 20% of all the crossover operations were one-way. This means that *some* circuits did pass from generation to generation unchanged.

Mutation Rate Results

The effect of disabling mutation is primarily one of *slowing* the progress of the GA. None of the five runs converged, although the best fitness values shown in table 7.6 suggest that it may have done if the GA had been allowed to continue. There are long, flat sections in figure 7.23a, followed by sudden large improvements. The addition of new 'genetic material' into the population would appear to result in a more efficient search than crossover can achieve on its own.

Increasing the mutation rate to 100% results in very typical GA behaviour that has been exhibited in experiments 1-3 and 6. Not all mutations will have an effect, which means that some individuals will remain unchanged even after undergoing mutation.

Predation Type

Setting the predation rate to 100%, and setting the predation *type* to single cull, results in a completely random search, as would be expected.

7.3 Fitness Function Comparison Experiments

A repeat of experiment 2 is presented here, but with the use of different fitness function. Previous experiments have been based on a novel pole-zero based fitness function. In order to identify the relative merits of this, a more traditional 'shape-fitting' fitness function has been employed, described in sections 6.7 & 4.7. The desired frequency response was given to the GA as a target.

7.3.1 Experiment 8: Shape Fitting Fitness Function

This experiment requires a different tolerance limit, and here it has been set to 4.0. There were a total of 401 data points in the target frequency response. A frequency response for each candidate circuit will be generated by SPICE, in the form of a decadic frequency sweep from 1Hz to 10kHz, 100 points per decade. A tolerance limit of 4.0 allows for 0.01 volts of error at each point. All other GA parameters are the same as used in experiment 2.

Parameters

Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 4.0$
Probability of Mutation	$p_m = 0.5$ *(0.9 Weighting)*
Probability of Predation	$p_p = 0.05$
Probability of Crossover	$p_c = 0.65$
Initial Tree Depth	$T_i = 3$
Max Tree Depth	$T_M = 5$
Max New Tree Depth	$T_N = 5$
Crossover Type	*Mixed (0.8 Weighting)*
Selection operator	*Exponential Rank*
Fitness Function	*Shape-Fitting*
Pole-Zero Parameters	$P_{lin} = 0.5,\ P_{dist} = 5000$
SPICE time-out	$t_{SPICE} = 60s$

Synthesis Results

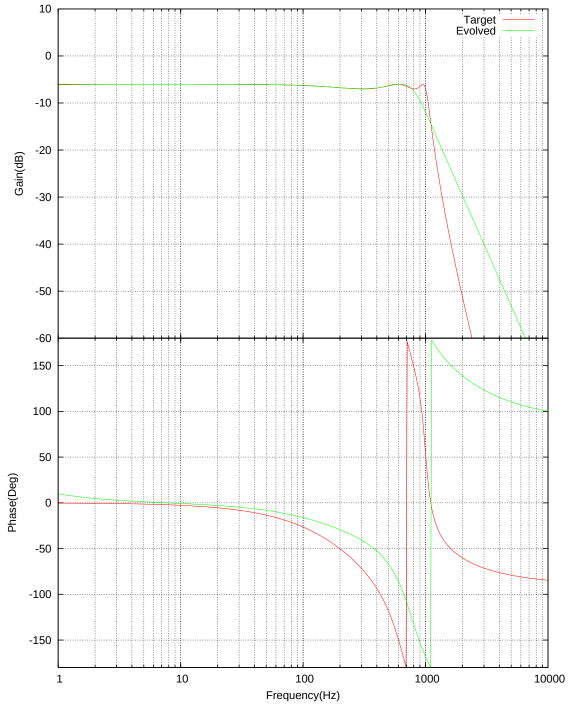


Figure 7.26: Target & Evolved Circuit Characteristics Of Shape-Fitted 5^{th}Order Chebyshev Filter

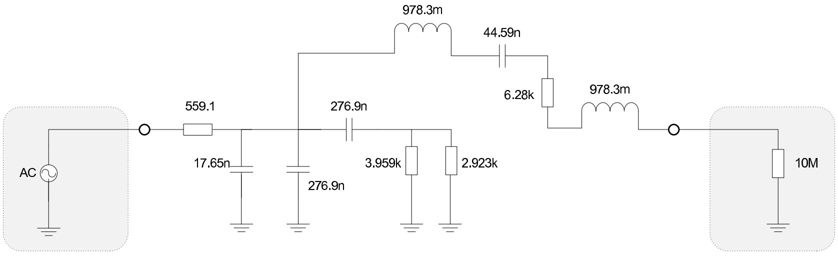


Figure 7.27: Evolved Circuit

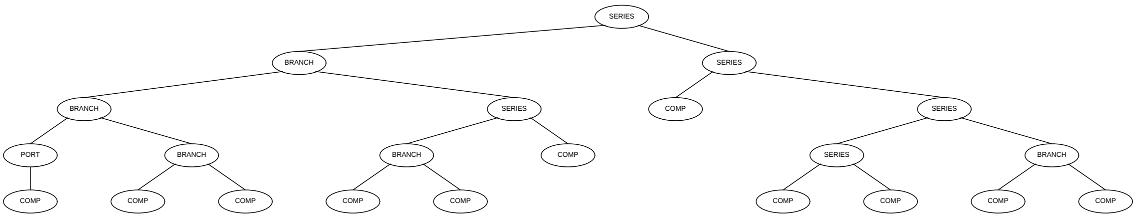
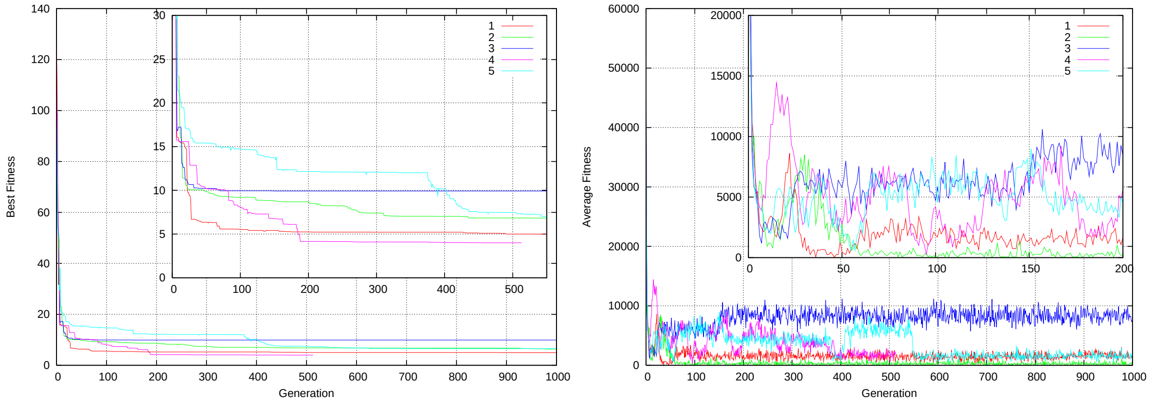


Figure 7.28: Encoding Tree Of Evolved Circuit

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
1	1000	No	4.94 (985)
2	1000	No	6.33 (985)
3	1000	No	9.92 (798)
4 (Best)	513	Yes	3.996
5	1000	No	6.47 (629)

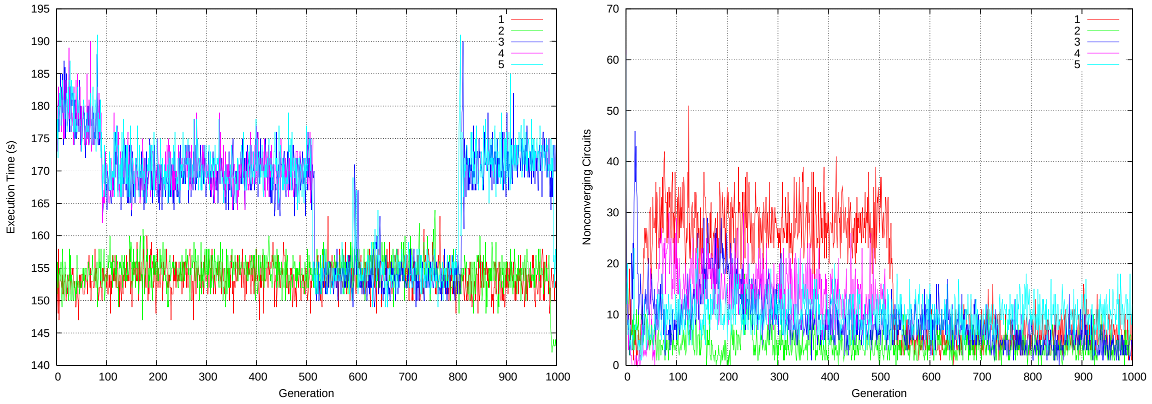
Table 7.10: Synthesis Results Summary

GA Behaviour



(a) (b)

Figure 7.29: Peak & Average Fitness Graphs of Five Runs



(a) (b)

Figure 7.30: Time & Non-Convergence Graphs of Five Runs of Shape-Fitted 5$^{\text{th}}$Order Chebyshev Filter

7.3.2 Discussion of Shape Fitting Experiment

The results of this experiment are quite different to those of experiment 2. The frequency is clearly not that of a 5$^{\text{th}}$order chebyshev filter, it does not have the same number of characteristic 'bumps' in the pass band. The roll-off is also much slower. However, the attenuation (which was the main issue with the circuit response in experiments 1-3) is much better here. The phase response is also significantly shifted, although the same argument will apply here as applied to the attenuation issue in experiments 1-3. The GA was given only the frequency response as a target and so had no information about the required phase response.

The most striking difference between the results of experiment 2 and this one, is the structure of the circuit. The initial experiments all produced circuits that were recognisably typical of the 'ladder' topologies of passive chebyshev filters, whereas the circuit produced this time has a much less recognisable structure. No pole-zero plot has been shown for this circuit as SPICE was unable to reliably produce this.

The best fitness graph (figure 7.29a) shows more gradual improvement in fitness than that exhibited by the pole-zero fitness function. Although the large step changes that result each time the number of poles and zeros in the current best circuit changes are not displayed here, there are still a number of smaller steps clearly visible.

7.4 Active Filter Synthesis Experiments

The synthesis results of two experiments are presented in this section. A 2$^{\text{nd}}$order Sallen-Key filter is the target of both synthesis experiments. Sallen-Key filters [57] are active and make use of operational amplifiers. The target circuit is shown in figure 7.31, with an operational amplifier circuit shown in figure 7.32. This is a CMOS two-stage op-amp [9].

The frequency response shown in figure 7.34 was given to the GA as a target for both experiments, the cut-off frequency is 1kHz. The phase response is also shown in that diagram but the GA was given only the frequency response. The pole-zero fitness function could not be used as SPICE proved unable to determine the poles and zeros accurately of many reasonably complex circuits, such as the target Sallen-Key filter used in these experiments.

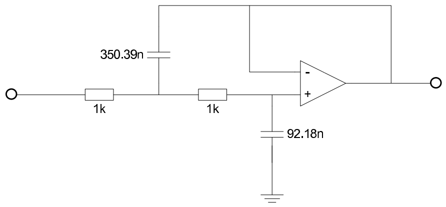


Figure 7.31: Sallen-Key Target Circuit

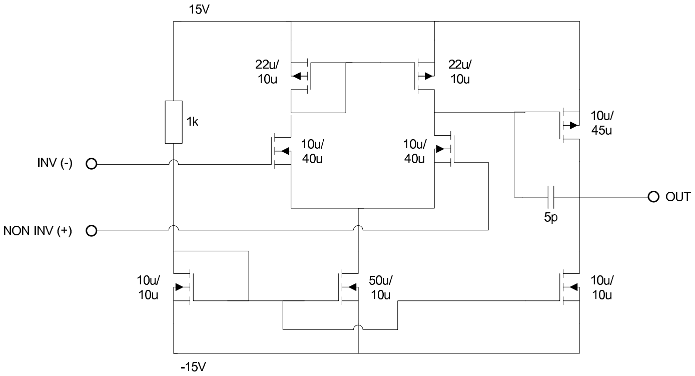


Figure 7.32: Two Stage CMOS Operational Amplifier

7.4.1 Experiment 9: 2^{nd}Order Sallen-Key Filter - Transistors

The Sallen-Key filter described above was synthesised using the component set shown below. The circuit embryo shown in figure 7.33 was used. Absolutely no predefined topology was given to the GA. The embryo is essentially identical to the embryo used in all previous experiments, with the addition of two extra voltage supplies.

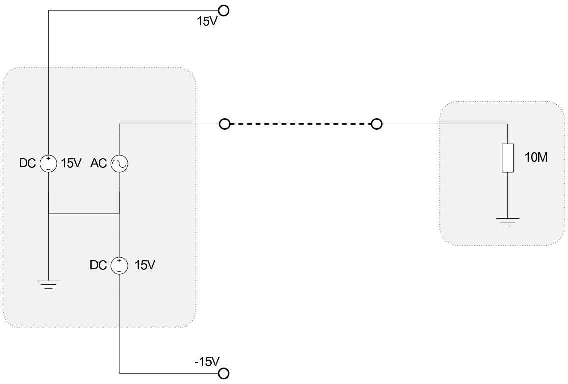


Figure 7.33: Embryo Circuit

Component Set

Resistors	$10\Omega \rightarrow 100K\Omega$
Capacitors	$0.1pF \rightarrow 1000\mu F$
N-Channel MOSFETs (3-Terminal)	*Length:* $10\mu m \rightarrow 50\mu m$
	Width: $10\mu m \rightarrow 50\mu m$
P-Channel MOSFETs (3-Terminal)	*Length:* $10\mu m \rightarrow 50\mu m$
	Width: $10\mu m \rightarrow 50\mu m$
N-Channel MOSFETs (4-Terminal)	*Length:* $50\mu m \rightarrow 100\mu m$
	Width: $50\mu m \rightarrow 100\mu m$
P-Channel MOSFETs (4-Terminal)	*Length:* $50\mu m \rightarrow 100\mu m$
	Width: $50\mu m \rightarrow 100\mu m$

Parameters

In most of the other experiments carried out, the maximum tree depth, T_M was set to 5. There is an expectation that a greater number of components will be required for the circuit being synthesised in this experiment, so here it has been increased to 7. This will give a total of 64 maximum components ($2^{(T_M-1)}$) which should easily be sufficient.

Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 10$
Probability of Mutation	$p_m = 0.5$ *(0.9 Weighting)*
Probability of Predation	$p_p = 0.05$
Probability of Crossover	$p_c = 0.65$
Initial Tree Depth	$T_i = 5$
Max Tree Depth	$T_M = 7$
Max New Tree Depth	$T_N = 7$
Crossover Type	*Mixed (0.8 Weighting)*

Selection operator	*Exponential Rank*
Fitness Function	*Shape-Fitting*
SPICE time-out	$t_{SPICE} = 60s$

Synthesis Results

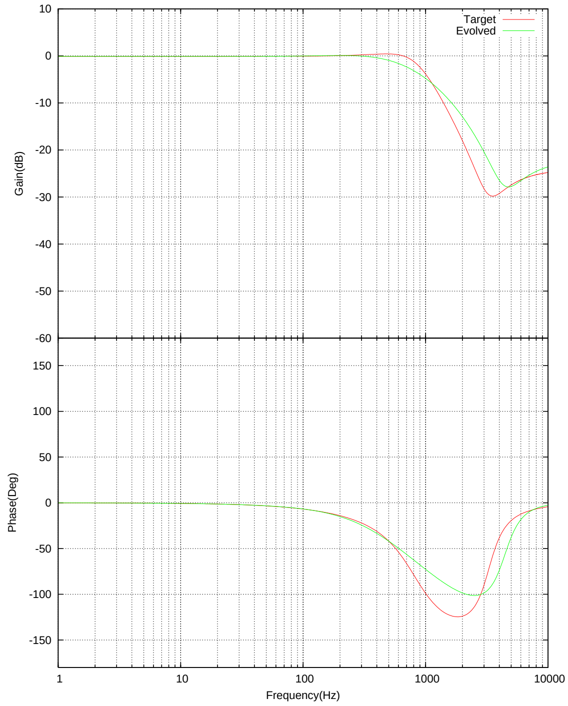


Figure 7.34: Target & Evolved Circuit Characteristics of 2$^{\mathrm{nd}}$Order Sallen-Key

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
1	1000	No	16.52 (992)
2	1000	No	17.09 (984)
3	1000	No	19.16 (473)
4	1000	No	23.51 (991)
5 (Best)	1000	No	11.50 (999)

Table 7.11: Synthesis Results Summary

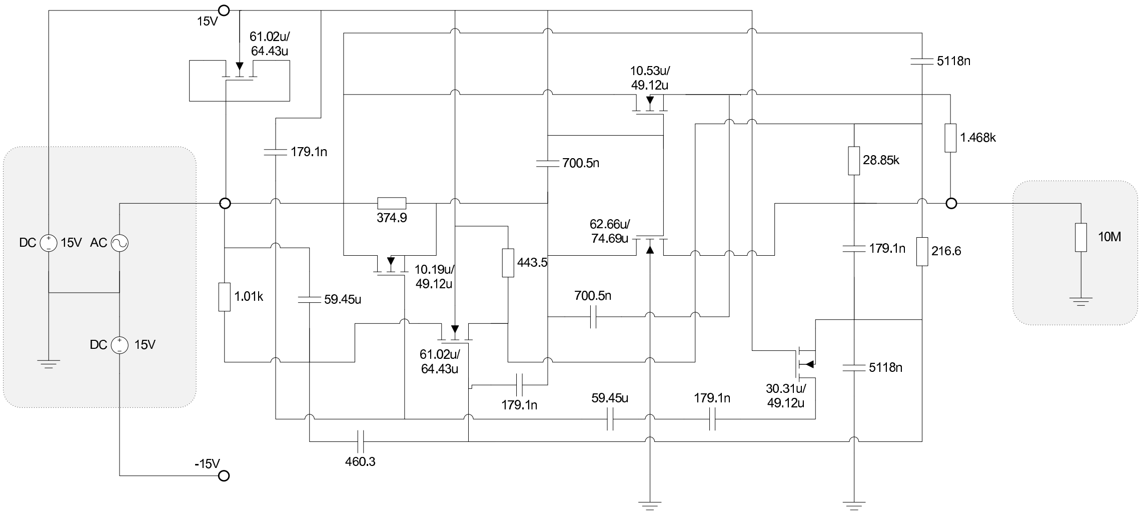


Figure 7.35: Evolved Circuit

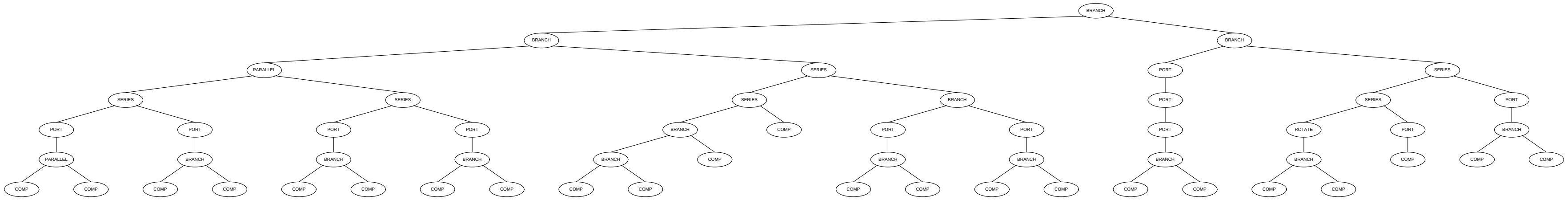
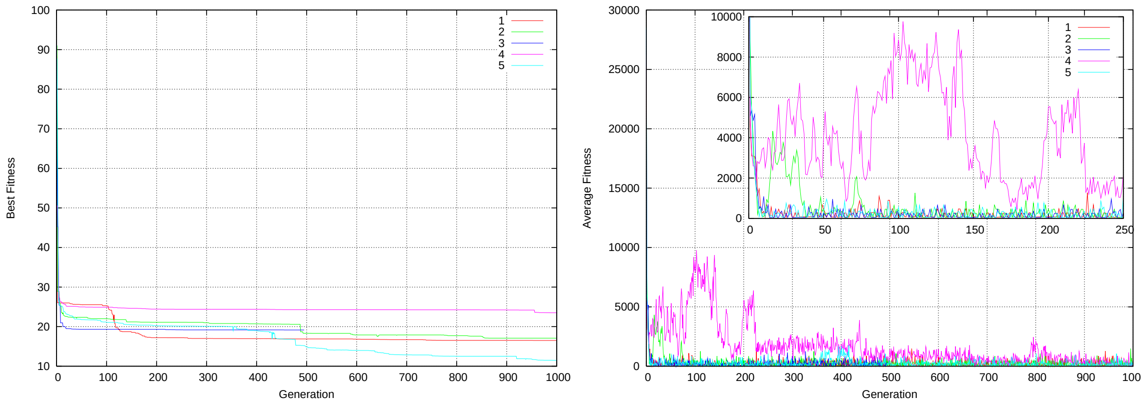


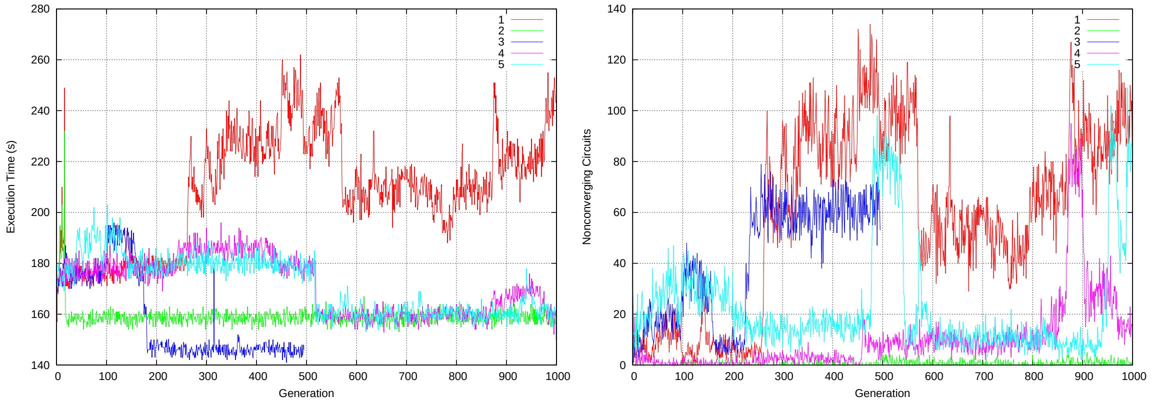
Figure 7.36: Encoding Tree Of Evolved Circuit

GA Behaviour



(a) (b)

Figure 7.37: Peak & Average Fitness Graphs of Five Runs



(a) (b)

Figure 7.38: Time & Non-Convergence Graphs of Five Runs of 2$^{\text{nd}}$Order Sallen-Key

7.4.2 Experiment 10: 2$^{\text{nd}}$Order Sallen-Key Filter - OpAmps

Experiment 9 was repeated, but with a different component set. The transistors were removed and replaced with a complete op-amp subcircuit (figure 7.32). Everything else, including most of the parameters with the exception of maximum tree depth, remained the same as for experiment 9.

Providing a complete op-amp design as a subcircuit represents a significant amount of predefined design knowledge. This also requires the use of a different circuit embryo. The one used in experiments 1-8 (figure 7.2) was used here. The 15V power supplies were contained within the op-amp subcircuit definition.

Component Set

Resistors	$10\Omega \rightarrow 100K\Omega$
Capacitors	$0.1pF \rightarrow 1000\mu F$
Operational Amplifier	*No parameters (circuit shown in figure 7.32)*

Parameters

Since an op-amp instance will count as just one component in this experiment, the maximum tree depth was set back to 5. This will give a possible maximum of 16 components.

Maximum Generations	$G_M = 1000$
Population Size	$N = 500$
Tolerance	$E_M = 10$
Probability of Mutation	$p_m = 0.5$ *(0.9 Weighting)*
Probability of Predation	$p_p = 0.05$
Probability of Crossover	$p_c = 0.65$
Initial Tree Depth	$T_i = 3$
Max Tree Depth	$T_M = 5$
Max New Tree Depth	$T_N = 5$
Crossover Type	*Mixed (0.8 Weighting)*
Selection operator	*Exponential Rank*
Fitness Function	*Shape-Fitting*
SPICE time-out	$t_{SPICE} = 60s$

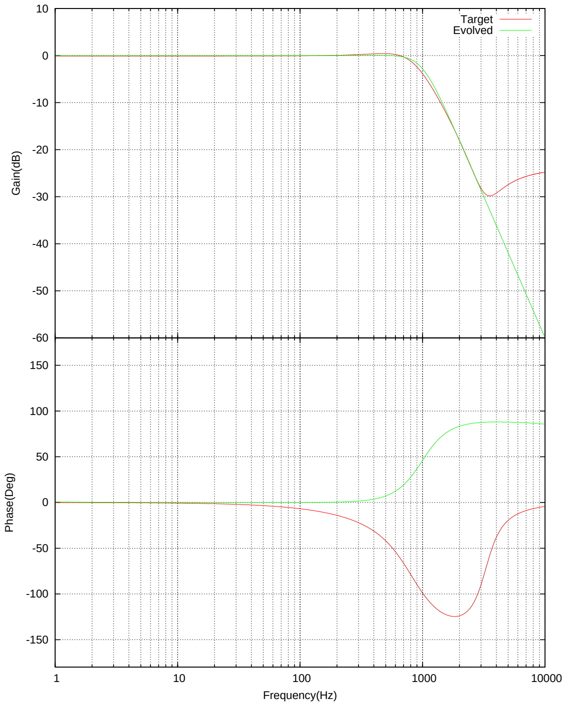


Figure 7.39: Target & Evolved Circuit Characteristics of 2$^{\text{nd}}$Order Sallen-Key

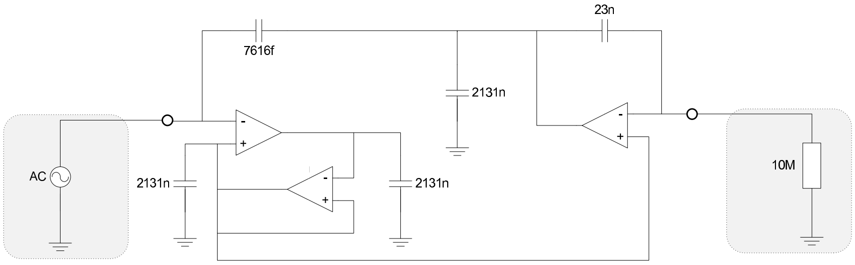


Figure 7.40: Evolved Circuit

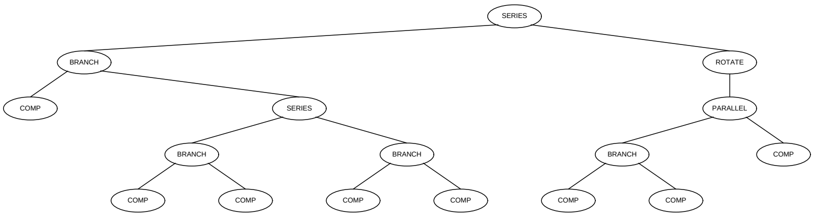
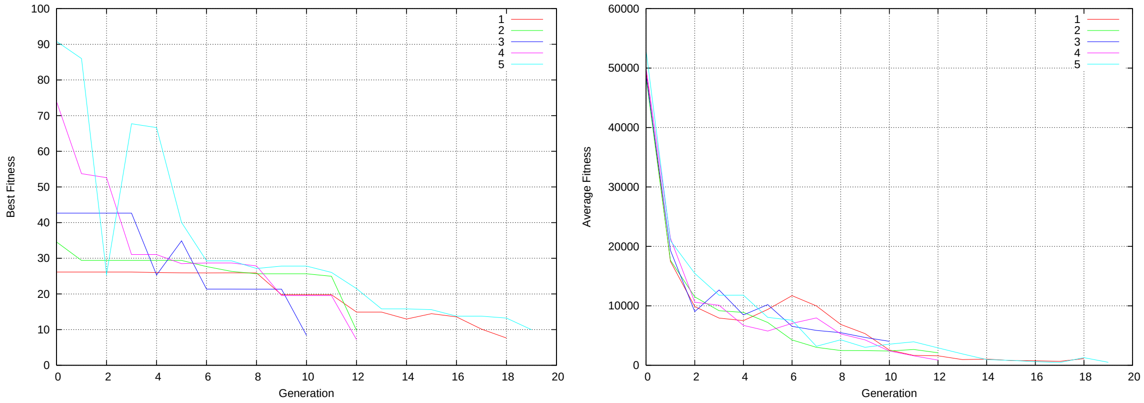


Figure 7.41: Encoding Tree Of Evolved Circuit

GA Run	Generations Executed	Converged?	Best Fitness (found in gen)
1	18	Yes	7.65
2	12	Yes	9.78
3	10	Yes	8.39
4 (best)	12	Yes	7.19
5	19	Yes	9.98

Table 7.12: Synthesis Results Summary

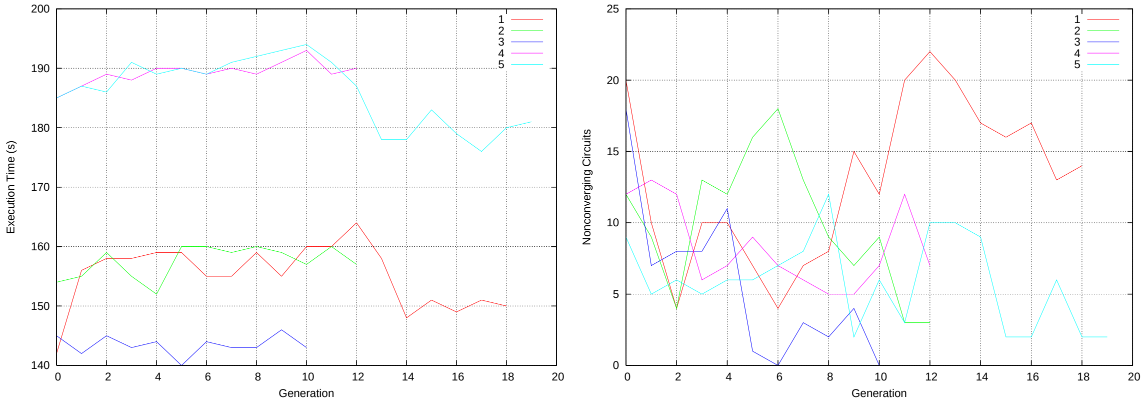
GA Behaviour



(a)

(b)

Figure 7.42: Peak & Average Fitness Graphs of Five Runs



(a)

(b)

Figure 7.43: Time & Non-Convergence Graphs of Five Runs of 2$^{\text{nd}}$Order Sallen-Key

7.4.3 Discussion of Active Filter Synthesis Experiments

None of the runs of experiment 9 converged, however the 4^{th}run came *very* close and might well have converged if left running. The circuit shown is the best circuit produced in a thousand generations of all five runs. The frequency and phase responses are very close to the target. Choosing a suitable tolerance score for a circuit is not an easy task before a few synthesis attempts are made. It would seem that the tolerance score used here of $T_M = 10$ was more stringent than it might have needed to be.

All of the runs in experiment 10 converged after a very short number of generations. Considering that the GA was provided with the majority of the target topology in the form of the op-amp in the available component set, this is perhaps not surprising.

Neither of the circuits produced by the GA resembled the target circuit (figure 7.31). Experiment 9 produced a circuit (figure 7.35) that is not recognisably similar to the op-amp circuit of figure 7.32. The circuit shown in that figure is a simple two-stage CMOS op-amp and is made up of a collection of common analogue subcircuits such as current mirrors and differential pairs. The evolved circuit does not contain any subcircuits that can be easily identified as performing a particular function, such as amplification or the setting of bias currents. The transistor in the top left hand corner of the circuit schematic appears to be completely useless, and is example of a *redundant* component. These are discussed in more detail in section 7.6.7.

Experiment 10 also produced a circuit (figure 7.40) which is extremely dissimilar to the target circuit. Structurally, the target circuit is very simple at the macro (op-amp) level whereas the evolved circuit is noticeably more complex. A total of three op-amps have been used, and a complex series of positive and negative feedback loops exist between them. This most certainly does not resemble a human-designed circuit.

7.5 Schemata Re-examined

Now that a series of experiments have been examined, it is useful to consider how the GA is working and why it is working in that manner. It was John Holland [20] who developed the first and probably still most widely accepted mathematical model of GAs in the form of Schemata Theory (sections 4.3 and 4.5). It is, admittedly, an incomplete model although it can still provide useful insight into GA behaviour. It is therefore useful to attempt to apply this theory to the results so far obtained.

The concept of schemata presented in section 4.3 apply to the encoding method used by the first GAs - fixed length binary strings. The theory is also easily applied to fixed length strings of arbitrary cardinality. Things are not so clear cut when applied to a variable length encoding system such as variable length strings or, worse still, more complex systems such as the tree encoding system used here or the one employed by Koza.

So what form might tree-based schemata take? Koza defines the schemata for his encoding system as being a set of subtrees that represent a set of LISP s-expressions [26]. A similar definition of schemata might be applied to the encoding system presented in chapter 6, with perhaps a few changes. However this will not be done here because, as will be shown, this would be not actually be of any benefit.

Central to the concept of schemata is the ‘don’t care’ symbol. This is precisely how the GA achieves its *implicit parallelism*, by effectively sampling other points in the solution space that share important features or characteristics with the actual point being tested. Characteristics which are *not* shared with the current point are not important. If they were, then those points could not be considered to have been sampled in parallel and would therefore belong to a different schemata.

So, for a given GA system, what determines the *don’t care* points? What determines whether particular characteristics of a point in the solution space are important? Schemata are usually represented (written) in terms of the encoding system, and indeed they are perhaps better understood in this way. However the encoding system does not play a significant role in *determining* what the don’t care points, and therefore what the schemata, are. Clearly, this depends upon the nature of the problem being addressed by the GA. It depends upon the nature of the entity being evolved; it depends upon the *application*. The encoding system is ultimately not as important as this is simply a means to an end. The encoding system serves to allow the evolving structures to be manipulated by the GA, to allow the GA to traverse the solution space. It is conceivable that some encoding systems could *mask* particular schemata, but even the most carefully design system cannot increase the number of schemata beyond that that the application allows.

The application is not the only thing that determines the don't care points of the candidate solution. Whether a particular feature or characteristic of that solution is important or not clearly also depends upon what is *required* in a satisfactory solution. Therefore, the *fitness function* also plays a role in the determination of the schemata.

No attempt will be made to apply the concept of schemata to the encoding system used here. It will be much more informative to apply the concept of schemata to the application of analogue circuit evolution.

7.5.1 Definitions For Analogue Circuit Schemata

It is necessary to consider what is meant by a *don't care* point in an analogue circuit. A given point in a binary string may be important or not, in general it is probably not ambiguous (although that may depend upon the application). Analogue circuits are complex, and in all but the most trivial of cases there will probably not be any physical aspect of the circuit which can said to be *totally* unimportant. There may, however, be certain physical features in that circuit that are of relatively little importance. Other aspects may be moderately important and others still will be very important. Some aspects will be absolutely crucial to the correct operation of the circuit.

Analogue circuits, therefore, do not in general possess features which can be considered to be *do care* or *don't care* points; it is not that black and white. In reality, the 'points', or features of an analogue circuit exist on a sliding scale of importance. It is perhaps more useful to be concerned with the number of important features in an analogue circuit than unimportant ones. Whether a feature is important or not will depend on whether it plays a significant role in contributing to the required characteristics of that circuit.

The fundamental Theorem of Genetic Algorithms (section 4.5) employs the useful concepts of *length* and *order* of schemata, although these are defined in terms of fixed length strings.

These concepts must be re-cast in terms of analogue circuits.

The *order* of a schema was originally defined by the number of alphabet symbols (anything except the don't care points) that the schema contains. This is useful in determining how likely that schema is pass through the mutation operator unchanged. *The order of a schema to which a given circuit belongs may be considered to be the number of components whose type, value or terminal connection points play a significant role in determining the required characteristics of that circuit.*

The *length* of a schema was originally defined by the distance between the first and last alphabet symbol in the schema. This is useful in determining how likely that schema is to pass through the crossover operator unchanged. *The length of a schema to which a given circuit belongs may be considered to be the proportion of the structure of the circuit which must remain intact in order to maintain the required characteristics of that circuit.*

7.6 Topology Mutability Experiments

This section presents four experiments designed to test the sensitivity of circuit function to changes in circuit sizing and topology. The aim is derive information about the length and order of the schemata of analogue circuits.

The smallest change that can be made to a circuit is to modify the value of one of the components. This experiment will be based on the target circuit from experiment 2 for two reasons. It is a small, simple circuit which will simplify its modification and analysis. The second reason is that this circuit was used in the GA parameter experiments and therefore serves as a reference point. Figure 7.44 shows the circuit.

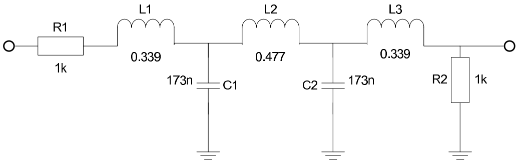


Figure 7.44: Filter Circuit

The four experiments presented here look at the effects of the modification of component *values*, modification of component *type*, modification of circuit *topology* by altering a single connection point of a single component, and finally the possible effects of the crossover operation on the circuit are examined. The first three effects examined could be as a result of the mutation operator, but all four could be caused by the crossover operator.

The definitions of length and order of schemata given in section 7.5.1 are defined in terms of the mutation and crossover operators. Although examining the effects of these operators will shed light on the length and order of these schemata, it must be stressed that this will strictly only be true *for the GA system presented in this thesis*, if *the definitions of length and order given in section 7.5.1 are followed*. The point is that there are many ways in which to implement the various genetic operators. As a result, the characteristics of a circuit that determine the corresponding schema's order will depend on how the mutation operator is implemented since schema order is defined in terms of mutation. Likewise for crossover and schema length. Some of the things that determine schema order in this GA implementation may play a role in determining schema length in another GA implementation. When it comes to analogue circuits, there may well be significant overlap between the definitions of length and order anyway.

The specifics of exactly what qualifies as length or order of schemata are ultimately unimportant, but the definitions given in section 7.5.1 provide a useful framework for discussion. These experiments will demonstrate the mutability of analogue circuit sizing and topology, and so it is again the *application* that is important here rather than the GA implementation. Therefore, these results will be applicable to all GA implementations designed to evolve analogue circuits.

7.6.1 Experiment 11: Component Value Modification

The circuit in figure 7.44 was simulated with varying component values, as shown in table 7.13. In each case, all other values were as shown in figure 7.44. Three values were picked for each component that was varied; a value very close to the original, a value much bigger than the original and a value much smaller.

R1	**L1**	**C2**
$1.1k\Omega$	$400mH$	$200nF$
1Ω	$2.0H$	$1pF$
$0.1M\Omega$	$10uH$	$10\mu F$

Table 7.13: Modified Component Values

Results

The frequency and phase responses of all nine modified circuits, plus the original circuit, are shown in figure 7.45. The corresponding pole zero plots are shown in figure 7.46. The pole zero plots become very difficult to read if the poles and zeros of ten circuits are displayed on a single plot. Figure 7.46 is therefore five plots. Plots 7.46a, 7.46b and 7.46c show the results of the R, L and C component modifications respectively. Some of the poles in the plots of R1=100kΩ, C2=1pF and L1=10μH are not shown as they do not fit on the scale used. As a result, the complete set of poles and zeros for these circuits are shown in plots 7.46d and 7.46e.

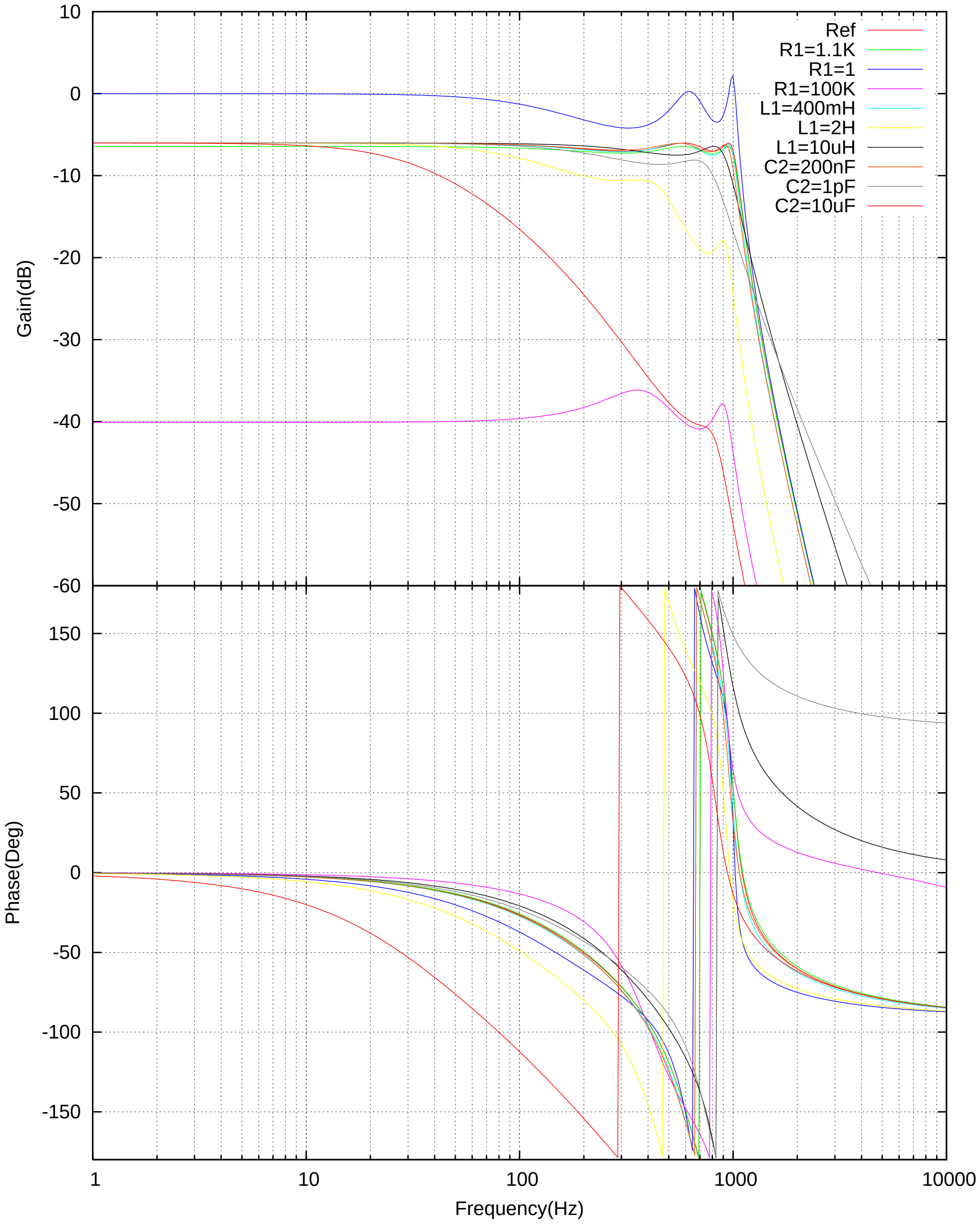
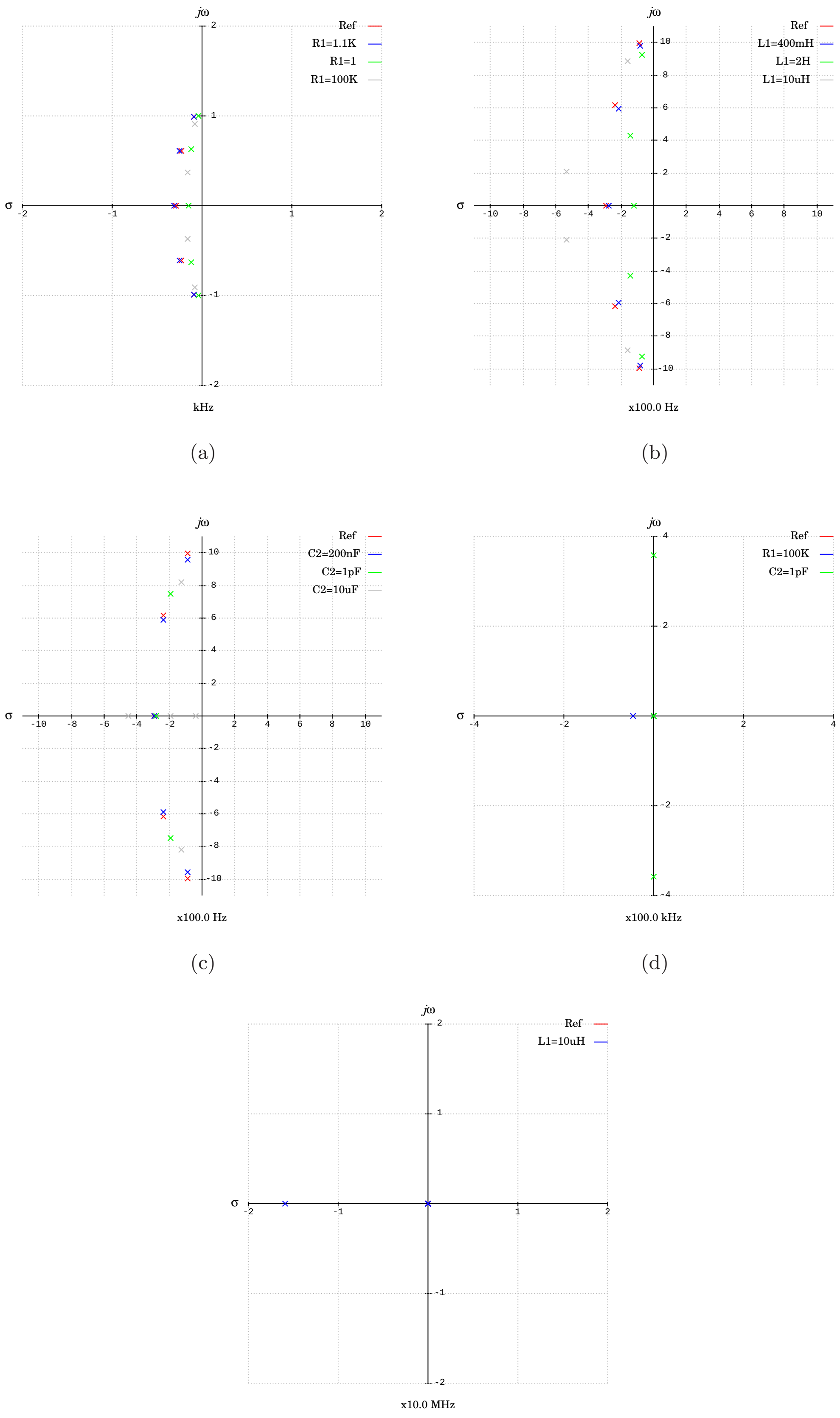


Figure 7.45: Frequency & Phase Responses



(a)

(b)

(c)

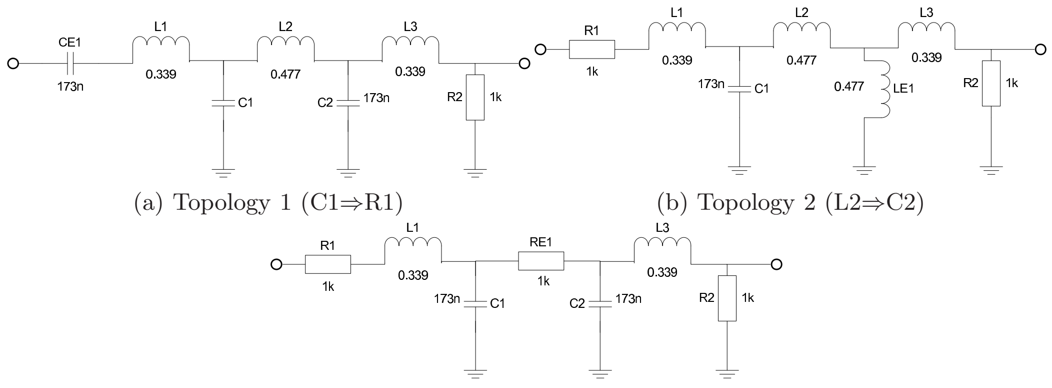
(d)

(e)

Figure 7.46: Pole Zero Plot Of Modified Circuits

7.6.2 Experiment 12: Component Type Modification

In this experiment three components of the circuit shown in figure 7.44 were replaced with a different type of component. The three circuits are shown in figure 7.47.



(a) Topology 1 (C1⇒R1)

(b) Topology 2 (L2⇒C2)

(c) Topology 3 (R1⇒L2)

Figure 7.47: Circuit Topologies Used in Experiment 12

Results

The frequency and phase response of all three circuits are shown in figure 7.49. The pole zero plot of topology 3 is shown in figure 7.48, the plots of the other two topologies are not shown as SPICE could not accurately determine them.

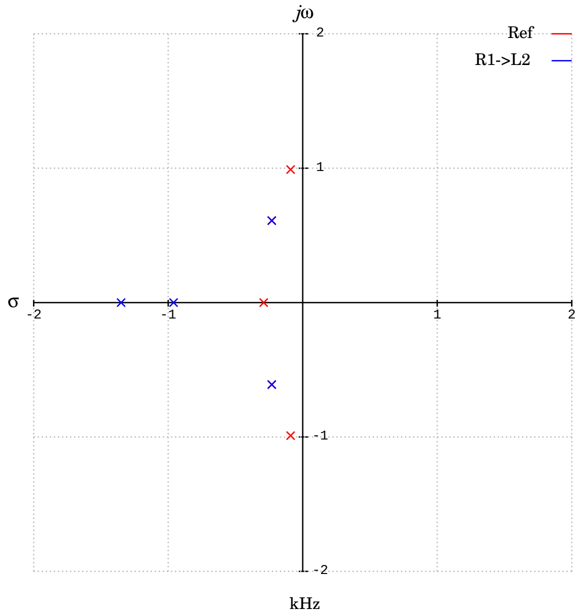


Figure 7.48: Pole Zero Plot Of Modified Circuits

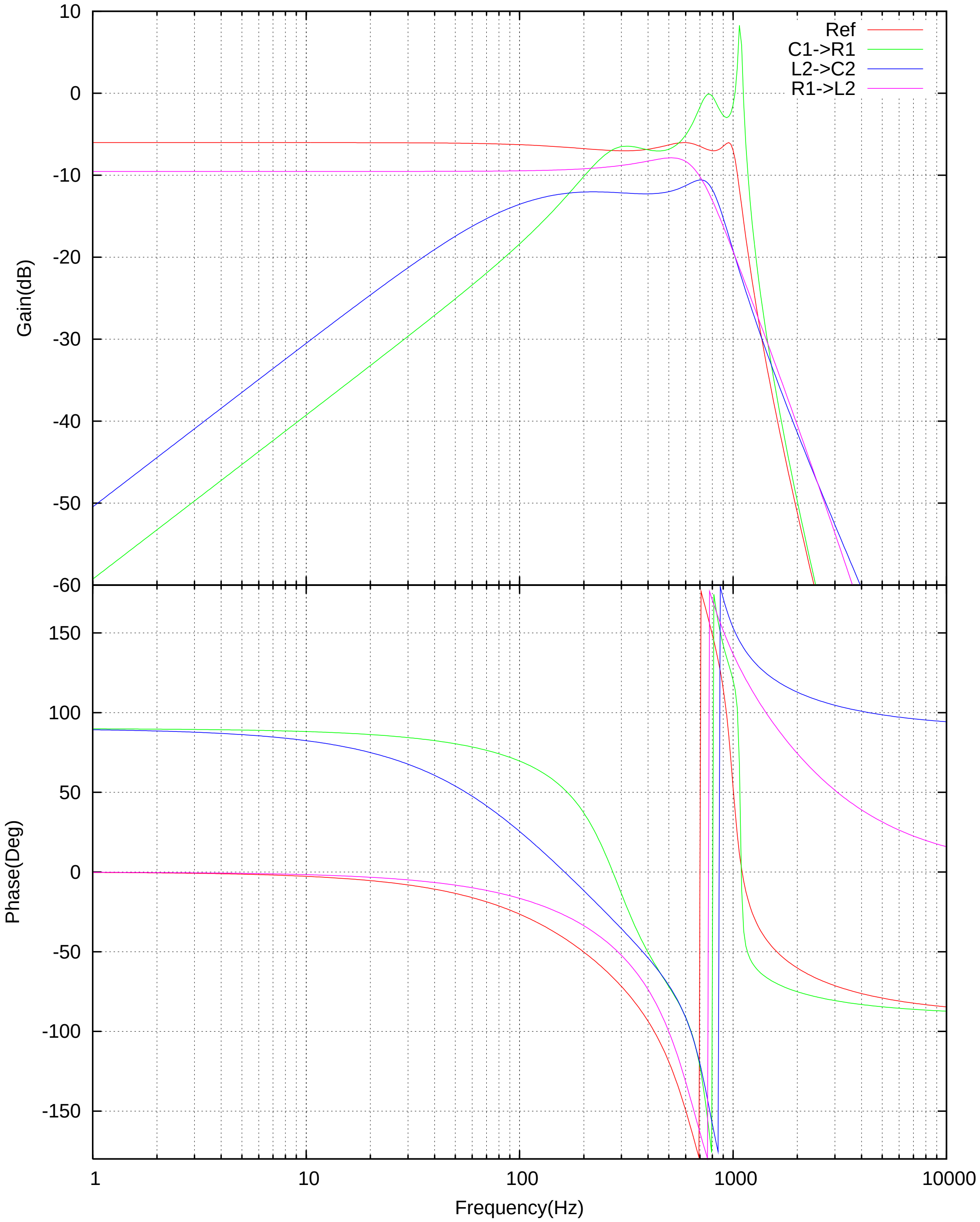
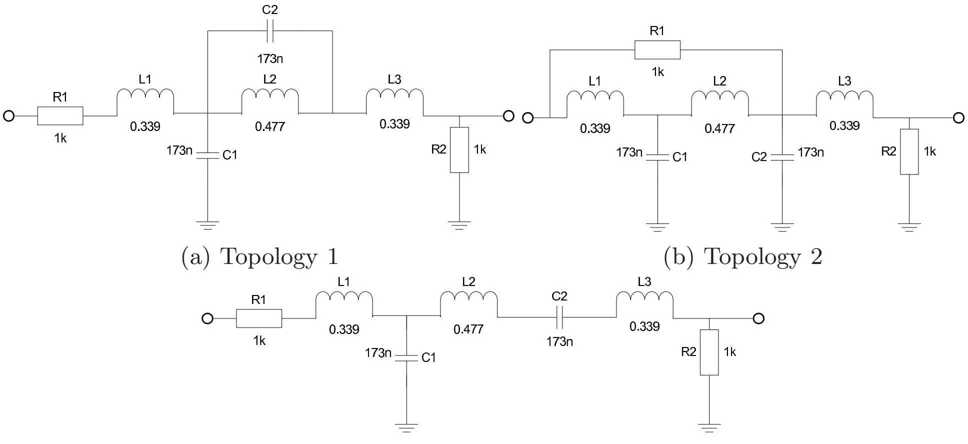


Figure 7.49: Frequency & Phase Responses

7.6.3 Experiment 13: Structural Modification

In this experiment three structural modifications were made to the original circuit of figure 7.44. In each circuit, one of the components had one of its terminals reconnected to a different node.



(a) Topology 1

(b) Topology 2

(c) Topology 3

Figure 7.50: Circuit Topologies Used in Experiment 13

Results

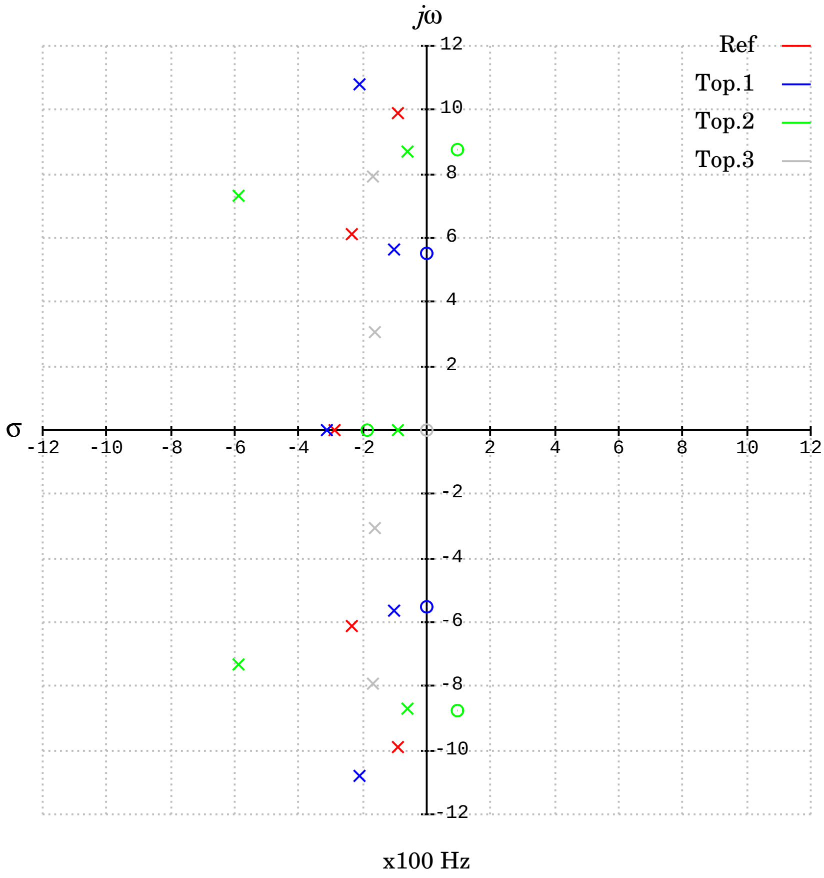


Figure 7.51: Pole Zero Plot Of Modified Circuits

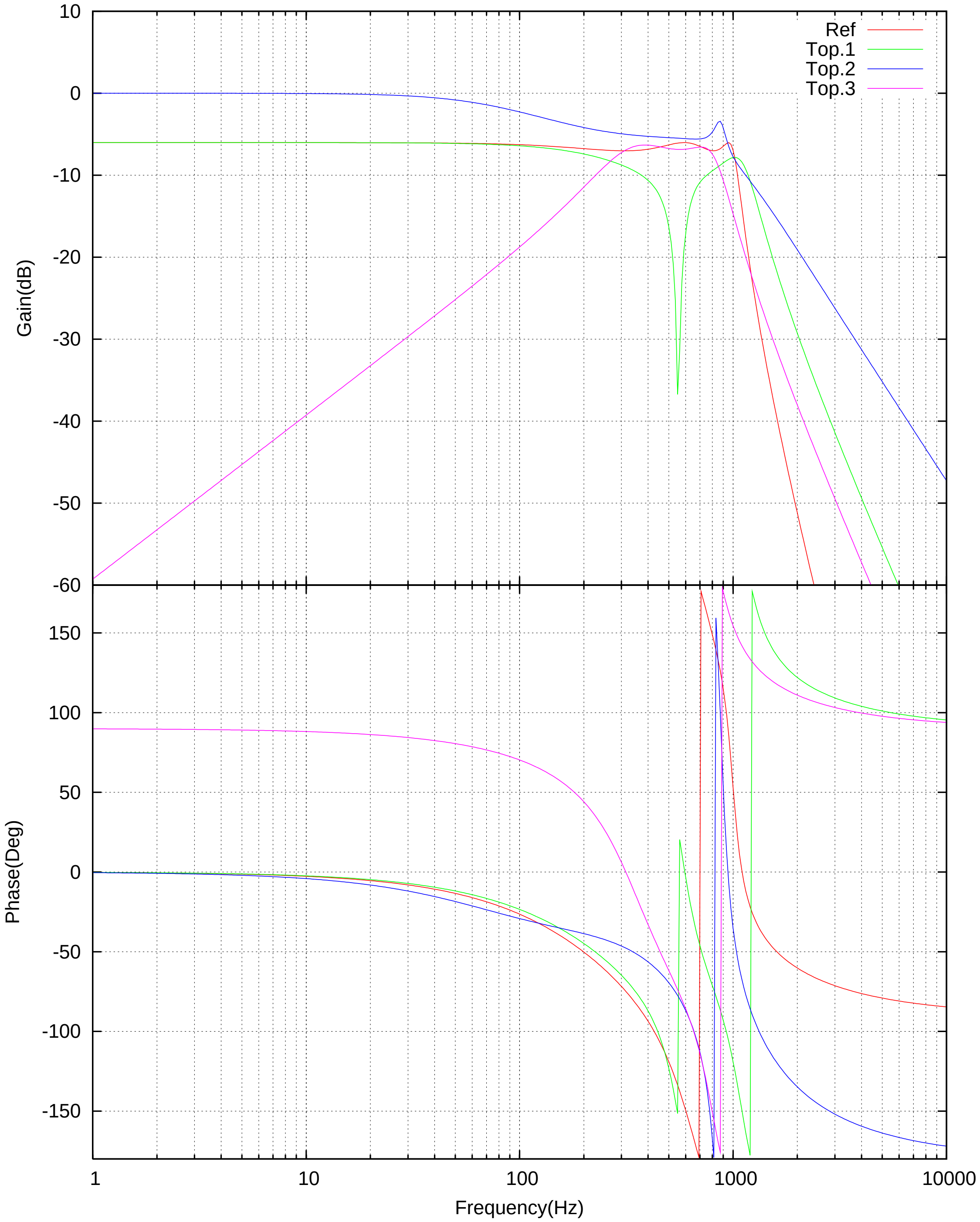
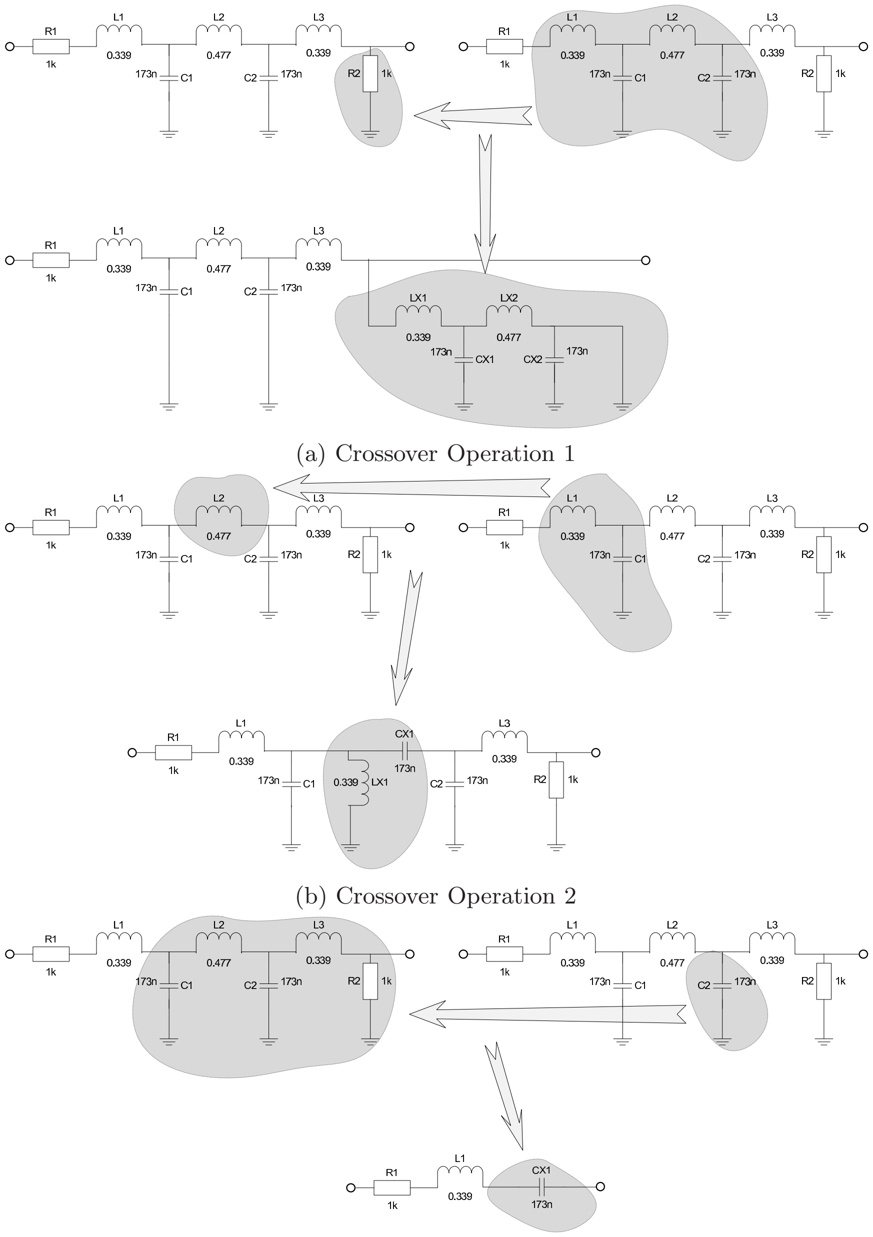


Figure 7.52: Frequency & Phase Responses

7.6.4 Experiment 14: Modification By Crossover

The possible topology modifications caused by crossover are examined here. Three possible topologies which are the result of crossing over the circuit of figure 7.44 with itself are shown in figure 7.53.



(a) Crossover Operation 1

(b) Crossover Operation 2

(c) Crossover Operation 3

Figure 7.53: Circuit Topologies Used in Experiment 14

Results

The frequency and phase responses are shown in 7.54, and the pole zero plots are shown in 7.55. The plot for topology 2 is not shown as SPICE could not accurately determine its poles and zeros.

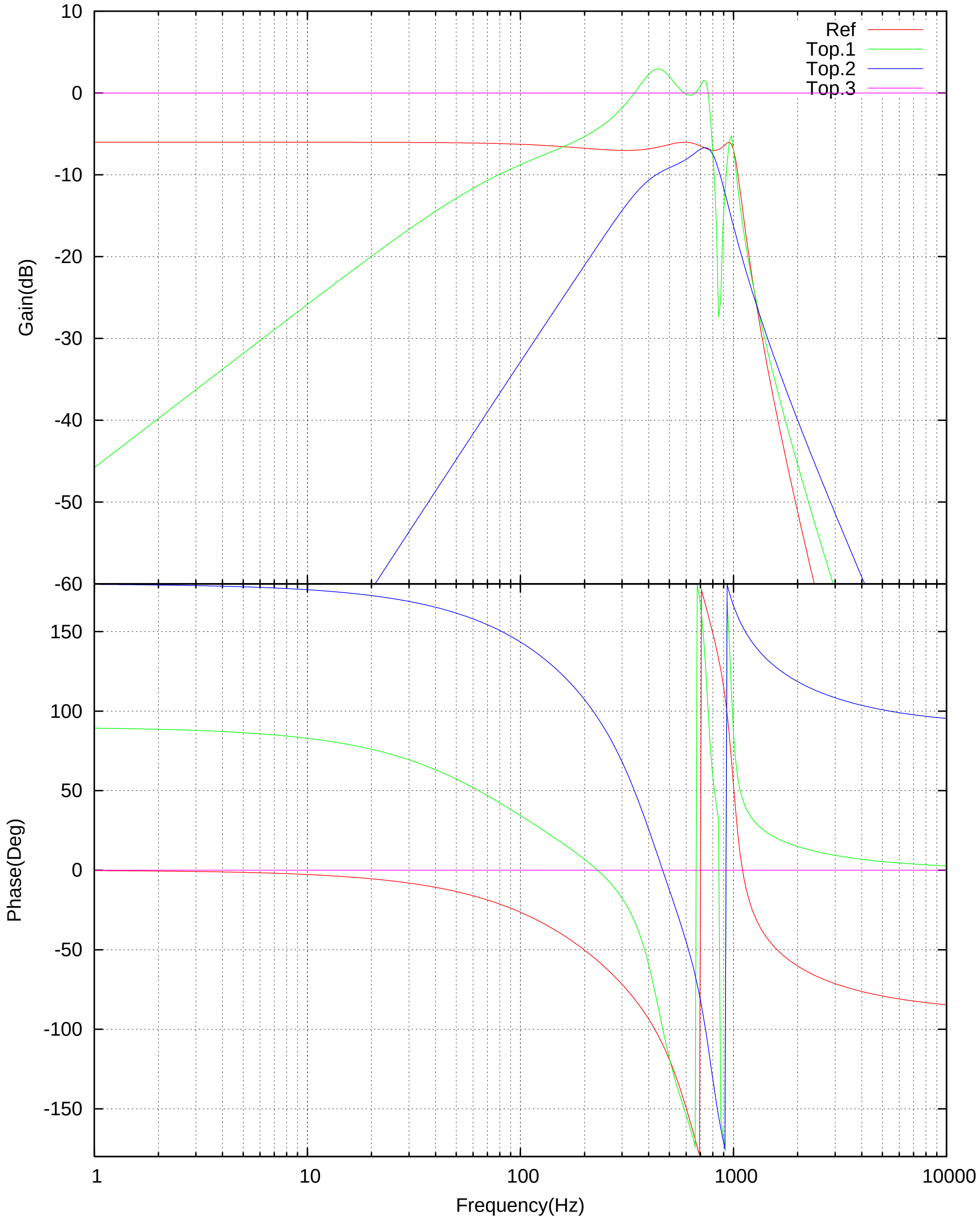
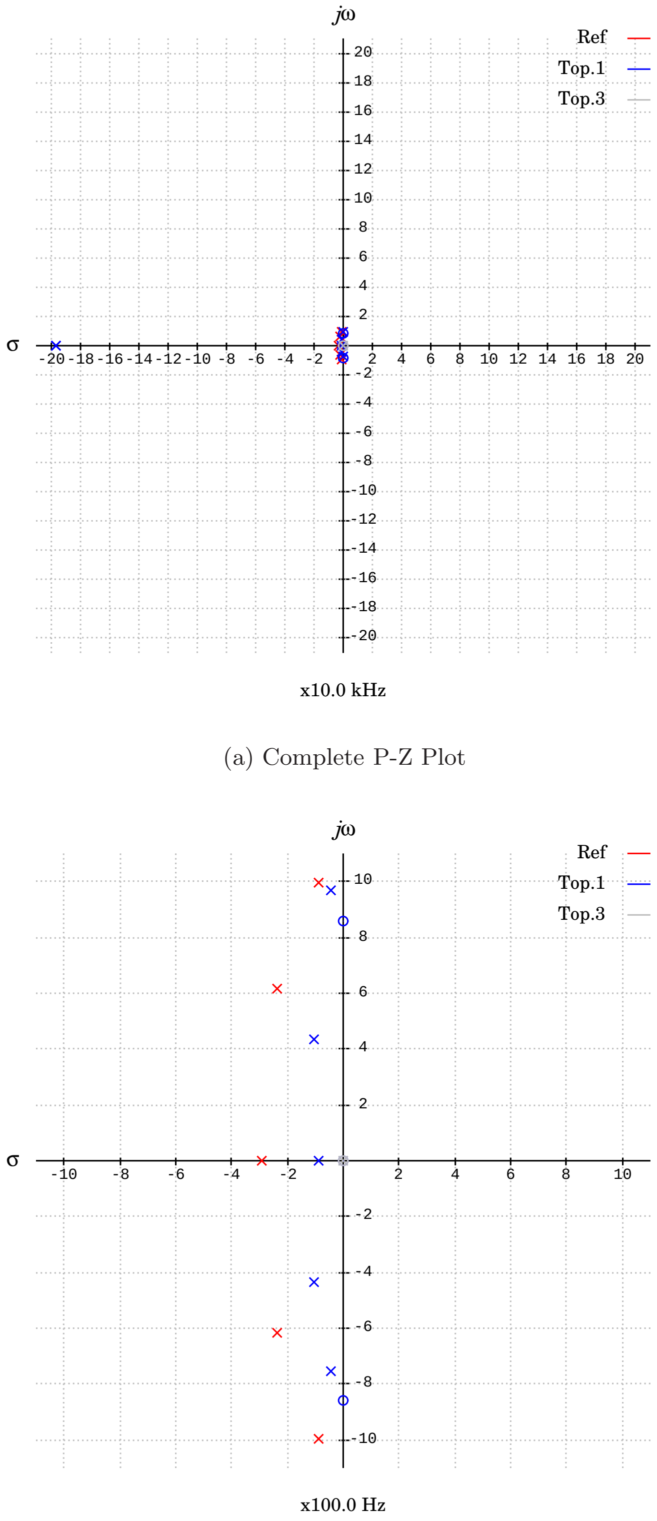


Figure 7.54: Frequency & Phase Responses



(a) Complete P-Z Plot

(b) Zoomed Plot (Some Poles Not Shown)

Figure 7.55: Pole Zero Plot Of Modified Circuits

7.6.5 Discussion Of Topology Mutability Experiments

The results obtained in this section will be compared to the definitions of the *length* and *order* of circuit schemata defined in section 7.5.1. The circuits used in these experiments are very basic, however that makes them ideal to serve as a demonstration. GA systems will vary in how the mutation, and perhaps to a lesser extent the crossover operators are implemented, and so the definition of length and order given in section 7.5.1 may not readily apply to other GA implementations used to synthesise analogue circuits.

It should be remembered that the point here is to consider issues inherent to the application of analogue circuit synthesis, and the mutability of analogue circuits is key to this. Therefore, other definitions of length and order maybe suitable for other GA systems as these definitions are framed in terms of genetic operators. However, the general observations and conclusions about circuit mutability will be applicable to all.

Order

The order of a schema was defined primarily in terms of its ability to pass through the mutation operator unchanged. Experiments 11, 12 and 13 show how the mutation operator may modify the circuit. Changes that result in only slightly modified component values, unsurprisingly, do not lead to significant changes in circuit function. However, this operator also has the capacity to make significant changes in component value, as well as change component type and in some cases even make changes in circuit structure. As can clearly be seen from the results most changes that the mutation operator make have the capacity to severely disrupt circuit function. In many cases, the circuit no longer acts as a low pass filter, instead exhibiting behaviour more akin to a band pass filter. The pass band attenuation can also be severely affected.

Length

The length of a schema was defined primarily in terms of its ability to pass through the crossover operator unchanged. Experiments 11-14 show how the crossover operator may modify the circuit. This operator clearly has even greater capacity than mutation to severely degrade or even completely change circuit function. It could potentially make any of the changes that the mutation operator makes, as well as completely transforming the entire circuit.

7.6.6 Deduced Nature Of Schemata Of Analogue Circuits

It is clear from experiments 11-14 that in general, the schemata of analogue circuits are *long* and of a *high order*. The examples discussed in this section are of small circuits, however other analogue circuits will almost certainly exhibit the same behaviour. Analogue circuits rarely contain redundant components, and in general it is true that all components in a circuit play a significant role in determining its characteristics. In experiment 2, for example, the input resistor did not determine the location of the poles and zeros of the circuit, however it had a significant impact on pass band attenuation. The topology and sizing of analogue circuits are highly sensitive to modification.

It is interesting to consider whether there is any way in which a circuit could be made less sensitive to modification. This is in fact conceivable if one considers *equivalent* topologies. All of the synthesised circuits in this chapter exhibit some level of redundancy or component replication. This is very significant because, as will be shown below, this acts to shorten the length and also possibly lower the order of the schemata.

7.6.7 Component Redundancy & Replication

Redundancy in a circuit takes the form of components connected in such a way that they contribute *nothing* to the function or characteristics of the circuit. Replication is the splitting of a single component into many, for example a capacitor of value $12\mu F$ could be replaced by two parallel capacitors of $6\mu F$, three of $4\mu F$ and so on. The *equivalent reduced* circuit is one in which all *replicated* components are *merged*, and all *redundant* components are *removed*.

Figure 7.56a shows a circuit which is exactly equivalent to the target chebyshev filter of experiment 2. The capacitor C1 has been replaced by three parallel capacitors C1a, C1b and C1c. While this circuit may be exactly functionally equivalent, it is quite different as far as the GA is concerned.

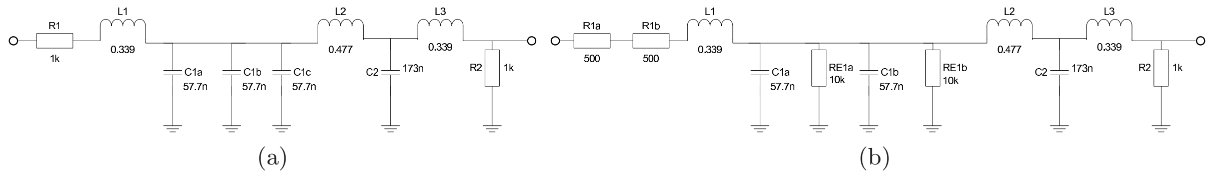


Figure 7.56: Examples Of Replicated Components

Mutation

The possible effects of mutation on this circuit equivalent will be considered. The most obvious difference is that the number of *points* available for the mutation operator to act upon has been increased. The mutation operator may do one of three things to the parallel capacitors; it may alter their value, it may change them into a different component or, in the case of the GA implementation presented in this text, it may alter the points in the circuit that the terminals are connected to possibly altering the circuit topology. As seen in experiments 11-13 most of these changes would normally lead to fairly significant changes in circuit behaviour, with only relatively small changes in component value having little impact.

In the case of these replicated capacitors, however, these high impact changes are not as certain. Changes in component value may not be as significant. The three capacitors are effectively just a single capacitor, so these changes will in effect only act on a third of that capacitor. If one of the capacitors is transformed into another component, it does not remove the capacitor from that part of the equivalent reduced circuit but instead acts to lower its value. Of course, if it is replaced with another reactive component like an inductor the impact on circuit behaviour may still be significant, depending on its value. A passive component like a resistor will probably have a much lower impact, particularly if it is of a high value.

This shielding effect is even more pronounced if the circuit in figure 7.56b is considered. This circuit is *not* exactly equivalent to the target circuit of experiment 2, but it is very similar. The chebyshev filters evolved in experiments 1 - 3 typically display this kind of structure, with extra resistors 'thrown in'. If capacitors C1a or C1b are changed by mutation to a resistor, or if either resistor RE1a or RE1b are changed to a capacitor, then effectively all that has changed are component *values*. That portion of the circuit would effectively contain one resistor and one capacitor in parallel, and it would still do so if either of the stated mutations took place.

The same argument can be made for resistor R1b. If changed to an inductor, then again all that has happened is modification of resistor R1a and inductor L1 component values. Although single change of this type would effectively modify *two* component values, it is still less likely to lead to a drastic change in circuit behaviour than a real modification of component type. A 'real' component type modification would be one that results in the addition or deletion of a component in the *equivalent reduced* circuit.

Depending upon how components are replicated, the nature of the circuit and what the important features are (in other words, what the fitness function is measuring), these replicated components could conceivably act to *increase* the order of the circuit. In general, however, they will act to lower the order as mutation will have a slightly lower probability of significantly altering the behaviour or characteristics of the circuit. For replicated components, there are fewer significant modifications that mutation can make. The same is true of redundant components.

Crossover

The effect of component redundancy and replication on the crossover operation is even more pronounced. During crossover, there are two ways in which the circuit may be altered. First of all, a portion of the circuit is cut out, and then secondly, a new circuit portion in inserted into the same place. Both the removal and insertion of circuit portions may significantly alter circuit function.

It is quite clear, however, that if a section is removed that contains only replicated components, as long as *one* instance of each of the replicated component types remain, then again all that is effectively happening is the modification of component values. The insertion of a new circuit portion is another matter. Either a single component may be inserted, or a very large sub-circuit may be pasted in. However, even if the chance of circut disruption on *one* side of the operation is lowered, the chance of the entire operation causing circuit function disruption is also lowered.

In reality many crossover operations will happen between the same or very similar circuits. As can be seen in experiment 4, a particular schema needs to gain a foothold in the population.

Many instances of a given schema need to be present in order for the GA to converge, and these will be relatively fit. This means that fit individuals are more likely to be crossed over with each other than unfit ones. Because instances of the fittest individuals are likely to be very similar, if not identical, then if they contain replicated components, it means there is an increased chance of a similar circuit portion being pasted *back in* to the circuit during crossover as was originally removed. The same argument is true of redundant components.

Again, it is clear that redundant and replicated components act to protect the circuit, and therefore schema, against the destructive effects of crossover. They act to *shorten* the length of the schemata.

Chapter 8

CONCLUSIONS

A series of experiments has been run that have examined various aspects of analogue circuits synthesised by GAs. Some simple passive filters were synthesised using a novel fitness function based on pole zero analysis, and then the sensitivity of the GA to its control parameters was investigated. A passive filter was synthesised using a much more common shape-fitting fitness function in, and then active Sallen-Key filters were synthesised. Finally, the effect of altering the components and topology of a filter circuit was investigated.

Although almost every GA system is unique in some way, as there are so many aspects of their implementation which may be varied, there are things which they all have in common. Some of the concluding remarks in this chapter are applicable only to the GA system presented in chapter 6, while others may be applied to any GA used to tackle the problem of automated analogue circuit synthesis. Some of them may even be applicable to other application domains, but the focus here is analogue circuit synthesis.

8.1 Applicability of GAs to Analogue Synthesis

The central focus of the thesis is about how *practically useful* GAs are in the application domain of analogue circuit synthesis. The results presented in Chapter 7 show that GA-based analogue synthesis systems can indeed produce viable circuit topologies. This does not, on its own, show that a realistically useful analogue synthesis system can be based on GAs.

GAs are ‘generate-and-test’ algorithms which search a *solution space* containing all possible circuits. An alternative way of searching this solution space would be to take a *brute-force* approach and sample the solution space entirely randomly. The effectiveness of such of a method would clearly depend on the *size* of the solution space, which in turn would depend on the complexity of the circuit. The greater the number of parameters required to define the circuit, the larger the solution space would be.

The topology mutability experiments presented in Chapter 7 (Experiments 11-14) show that circuit behaviour is highly sensitive to relative component values, the types of component used and the topology of the circuit. A large number of parameters would therefore be required to define a complex circuit, meaning it is less likely that such a circuit would be generated randomly.

In practice, therefore, it is extremely unlikely that such a brute-force approach would produce a useful result in anything like an acceptable time frame. A random search would be an inefficient and slow process. It would be possible to speed up this process by providing more resources[1]. If enough resources were provided, this approach *would* eventually find a satisfactory circuit, although the amount of resource needed would almost certainly be prohibitive.

It would also be possible to speed up this process by *guiding* it in some way. This is what a GA is, a guided search in which randomness plays a significant role.

8.1.1 Guiding the Search

How should a search through the solution space be guided? Both GAs, and the brute-force approach described above, start the search in the same way. A collection of points in the search space are randomly chosen. From those initial points, a path to a satisfactory solution must somehow be found.

If starting off from an entirely random point, and there is no information available at all about where the end point may be, it is intuitive to test points close to the initial point, in order to derive a direction in which to continue the search. In order to *guide* this search process, it must be possible to *incrementally* construct a path to a final solution.

GAs derive new solution points from current solution points. The function of the selection genetic operator means that the better a given solution is, the greater the number of new points that will be derived from it. *Ideally*, this will have the effect of focusing the search on the current best solution points, and ideally, new points will tend to be grouped around them.

A key question that should be considered here, is what constitutes a 'close' point in the solution space? The aim of the search is to find a circuit that meets a set of stated requirements. The nature of these requirements may be quite varied, but the primary requirement will of course

[1] Time and computational power.

be the *behaviour* of the circuit. In order for a GA to derive a direction in which to move through the solution space, the nature of that space must be such that solution points *close* to a current point are similar in terms of *behaviour*, this is an implicit assumption.

GAs use the crossover and mutation operators to derive new solution points. These operators may change the value, type and/or connectivity of components within a circuit, resulting in a new circuit. These new circuits correspond to new points in the solution space.

As shown by Experiments 11-14, circuit behaviour is highly sensitive to these changes. As a result, *a small change in the circuit may very easily result in a significant change in circuit behaviour.* In other words, if the solution space is considered to be such that 'close' points are similar in behaviour, small circuit changes will in many cases result in a solution point that is very far away from the current point.

What constitutes a 'small' circuit change? Two circuits would be considered to be *physically* similar if they varied only in the value or type of one (or very few) components. They may also be considered to be physically similar if one (or very few) component terminals were connected to different circuit nodes.

The action of the genetic operators of crossover and mutation result in *physical* modification of circuits. For a small physical circuit change to *guarantee* resulting in a new solution point being close to an original, or current, solution point, the nature of the solution space would need to be different to that considered so far. Solution points which were 'close' to each other would need to be *physically* (not behaviourally) similar. If this were the case, then the genetic operators could indeed produce new solution points that were 'close' the original, or current, solution points. However, because circuit behaviour is highly sensitive to physical circuit changes (as shown by Experiments 11-14), these new, 'close' points, would likely *not* possess similar behaviour. This could seriously hinder the ability of the GA to derive a new search direction from the current points.

As has been discussed, there are two ways in which the solution space may be considered. Ultimately, however, this is unimportant. Both ways highlight a fundamental problem that GAs are faced with when applied to the problem of analogue circuit synthesis:

- **Genetic Algorithms use circuit *behaviour* to determine which points to sample next, but only have the ability to directly modify *physical* circuit characteristics.**

As shown by Experiments 11-14, in many cases there is no link between the *size of the physical circuit change* and the *size of the resulting change in circuit behaviour*, unless the physical change is *very* small. Therefore, for all but the very smallest physical changes, new solution points will in many cases not be behaviourally similar to current solution points. A consequence of this is that:

- **The ability of Genetic Algorithms to *incrementally* construct a path through the solution space to a final, acceptable solution is impaired.**

The weaker the link between the size of physical changes and size of behavioural changes, the more reliant the GA is on chance and the more it resembles a brute-force search. *This means that the weaker this link is, the greater the resources (time and computational power) it will need to arrive at a solution.*

8.2 Drawn Conclusions

8.2.1 Resource Requirements

The resource requirements for GAs are considerable. The main resource is time and/or computational power. The biggest proportion of the execution time of a GA-based analogue synthesis system is spent simulating or analysing candidate circuits. GAs usually require many generations to find an acceptable circuit.

8.2.2 Fitness Functions

Many of the experiments presented in Chapter 7 - Experiments 1, 3, 8, 9, 10 and *in particular* Experiment 2, all demonstrate a particular trait of GAs. The circuit produced by a GA upon convergence is *only* guaranteed to satisfy the specified criterion. If any aspect of the circuits implementation, behaviour or performance is not measured as part of the fitness function, those unmeasured aspects of the circuit are essentially left to chance.

Experiment 2, for instance, possesses a frequency response whose *shape* is very close to the target. The pass band attenuation, however, is nothing like that of the target. This is because only the *location* of the poles and zeros are measured by the fitness function, and these on their own do not convey information about attenuation. Another example of this is the shape-fitting experiment (Experiment 8) in which the frequency response was supplied as a target. This shows a pass band attenuation *very* close to that specified, but the overall shape of the frequency response was not as good, nor was the phase response.

The synthesis of real, practically useful analogue circuits requires many circuit parameters and specifications to be met. A multi-objective fitness function would therefore appear to be crucial. The fitness functions used in the GA system presented in Chapter 4 are single-objective.

The pole-zero fitness function helps in finding a viable topology quickly. Unfortunately this method is limited here due to the unreliability of SPICE, as reasonably complex circuits (such as those containing operational amplifiers as subcircuits) often caused HSPICE to return entirely inaccurate results of pole-zero analysis. Using a dedicated pole-zero calculator may yield better results.

8.2.3 Impact of SPICE Non-Convergence

The issue of SPICE non-convergence, and other issues with SPICE discussed in Chapter 5 did not prevent the GA from producing viable circuit topologies. The GA quickly selected against circuits which could not be simulated, and the proportion of the population affected by this was quickly reduced.

It is interesting to note that non-convergent circuits were *never* eliminated from the population in any of the experiments. This may in part be due to the predation operator which introduces randomly generated circuits into the population[2]. After a reduction of the initial peak, the percentage of non-convergent circuits remained relatively high - typically between 10-20% in most of the experiments using the pole-zero fitness function, and typically around 5% when the shape-fitting fitness function was used. This issue has two particular effects which are a concern:

1. The computational effort spent trying to simulate a circuit which does not converge is wasted. In addition to this, more effort will typically be spent on these circuits than convergent circuits due to that fact that in such situations SPICE goes through a series of attempts at producing a simulation. The proportion of execution time spent on non-convergent circuits will be greater than the proportion of non-convergent circuits in the population.

2. Much of the solution space will be off-limits to the GA, as it will not be able to explore these regions. Put another way, certain schemata will be untestable. Given the omni-present nature of non-convergent circuits in the population, it would appear that untestable regions of the solution space are *very* common, and are densely scattered throughout. This depends to an extent on the type of SPICE analysis being performed, which in turn depends on the fitness function being used.

During all experiments, detailed per-generation data was collected about non-convergent circuits. Every such circuit netlist was recorded, along with the corresponding SPICE output and error messages. This would allow a highly detailed analysis of SPICE convergence to be performed, including the incidence of particular failure mechanisms. Such an analysis is not be presented here due to time and space constraints.

[2] Non-convergent circuits occur with a markedly higher frequency in populations of entirely randomly generated circuits. This can be easily observed in the higher incidences of non-convergent circuits during early generations.

8.2.4 Topology Generation

The data presented in Chapter 7 shows that GAs are certainly capable of generating *viable* circuit topologies. The GA used in these case studies produced circuits which satisfied the fitness target, but these circuits were often different in some way to a design that a human engineer would produce. The Chebyshev filters produced in Experiments 1-3 roughly followed a typical topology for this kind of passive filter, but the more complex topologies produced by Experiments 9 & 10 were certainly not recognisable. What is quite evident from the results is that:

- **Circuits generated by the GA are very likely to contain redundant and/or replicated components.**

The impact that crossover and mutation are likely to have on circuit behaviour has been discussed in section 8.1.1, and the corresponding protective effects of redundant and replicated components discussed in section 7.6.7.

This may be a problem for things such as size, area and power dissipation of circuit, amongst other things. Although it may be possible to prune some of these components after the circuit has been evolved, that not only reduces the level of design automation afforded by the GA, but for larger, more complex circuits it may be extremely difficult to identify these components. Techniques such as *current-flow analysis* [50] may prove useful for this task, however.

If considered in terms of schema theory, circuits which contain redundant and replicated components will belong to slightly shorter, slightly lower order schemata. If considered in terms of the discussion presented in section 8.1.1, the existence of redundant and replicated components will mean that there will be a slightly stronger link between the size of physical circuit changes and the resulting change in circuit behaviour. Such circuits are less likely to be destroyed by the genetic operators and so have more chance of being the circuit converged upon by the GA.

8.2.5 Genetic Algorithms As A Useful Design Tool

Answering the question of whether GAs can form the basis of a realistically useful automated analogue circuit design tool is one of the primary goals of this research. The issues discussed so far all play an important role in this. It is useful to now review the criteria identified in section 1.3.1, which such a tool must satisfy:

- **Automation:** It is possible to implement GAs in such a way that they can produce a circuit *topology* with virtually no starting information. A tool based on such GAs satisfy the definition of *synthesis* given in section 3.1.2[3]. In this respect, GAs offer a high level of design automation.
- **Robustness:** A synthesis tool must be robust both in terms of the circuit it produces (a good Quality Of Results) and in terms of its ability to reliably produce a circuit. The QOR of a GA is highly dependent on the comprehensiveness of the fitness function. A single objective fitness function is unlikely to produce good QOR. GAs exhibit high sensitivity to their control parameters, and significant trial and error is often required to find suitable values. Even when suitable values have been found, GAs will frequently *not* converge, they do not exhibit a high level of reliability. Therefore, in this application domain, GAs are not robust.
- **Scope/Generality:** The high level of design automation offered by GAs means that there are very few restrictions on the types of analogue circuit which can be synthesised. As a synthesis technique, GAs offer a very high level of generality. However, the range of circuits which may be synthesised with a given GA implementation may be limited by its available fitness functions.
- **Execution Time:** As discussed in section 8.2.1, typically, the time taken for GAs to converge (if they do converge at all) is very high. An experienced human designer is likely to be able to design a suitable circuit is less time than it takes the GA to converge. The run time of the GA during the synthesis of some of the simple passive filters presented

[3] The input to a synthesis system is in the functional domain and the output is in the structural domain.

in Chapter 7 was of the order of a few *days*. The execution time of GAs is likely to be prohibitive with typical, currently available computing resources.

A realistically useful design tool must satisfy *all* of the above requirements, GAs *currently* satisfy *at most* two. As discussed, this has much to do with the fact that analogue circuits are highly sensitive to modification which, ultimately, means that:

- **Genetic Algorithms are not *immediately* well suited to the task of entirely, or almost entirely unconstrained analogue circuit synthesis.**

It is certainly true that GAs can produce analogue circuits which satisfy a given fitness function. However, the resource requirements, control parameter tuning and quality of results (both in terms of circuits produced and likelihood of GA to produce anything useful at all) mean that GAs are not *currently* a practically useful design tool for the open-ended, unconstrained synthesis of analogue circuits. Careful consideration must be given to how GAs and the circuit representations they process may be constrained in order to make use of known-good analogue subcircuits and other important knowledge of analogue circuits.

8.3 Practical Significance Of Schema Theory

The Fundamental Theorem Of Genetic Algorithms (section 4.5) yields some important results, and is perhaps still one of the best explanations of GA behaviour. The concept of schemata works well for fixed length strings of a given cardinality. It is harder to apply it to a non-string encoding scheme and harder still to apply it to a variable-length encoding scheme.

Ultimately, the concept of the length and order of schemata is just a way of expressing how sensitive an individual in the GA population is to change. There is an implicit link in this idea of length and order of schemata between changes to an individual and a resulting change in the *performance*, *quality* or *suitability* of that individual. If an individual is a member of a schema which is long and of a high order, there are many parameters that the individual possesses which are important in producing a solution which possesses desired characteristics. Changing one of those parameters will result in a significantly different solution.

It is this implicit link which is important, and therefore the *concept* of schemata may still be useful if applied to complex, variable-length encoding schemes. It may not be a particularly easy fit however, but an individual being sensitive to change would certainly seem to satisfy the idea of a long and high order schemata. Analogue circuits are in general highly sensitive to change, and so therefore may be considered to belong to long, high order schemata.

As highlighted by Zebulum ([58] and section 4.5), schema theory is incomplete. In particular, the following points should not be ignored:

1. Only the destructive effects of the three main GA operators are taken into account, but it does not consider the role these operators play in building good schemata.
2. The theorem suggests that encoding schemes should be used where fit individuals belong to short and low order schemata.
3. It also suggests that a low alphabet cardinality should be used as this will increase the number of schemata and therefore increase the implicit parallelism of the GA.

8.3.1 Constructive Effects Of Genetic Operators

The first point in the list above is certainly true. The *constructive* effects of the main GA operators are difficult to quantify, even when considering a simple encoding scheme and application. It is quite clear that the effects of the crossover operator, for example, are not only destructive. Crossover is vital to the way a GA operates and provides its primary means of traversing the solution space.

One of the primary constructive effects of the genetic operators is their ability to test out new points of the solution space. The discussion presented in section 8.1.1, concluded that new solution points which lie very far away in the solution space from the solution points they are derived from have the effect of hindering the search. If this happens often this is true.

Testing *some* solution points very far away from current solution points is not *necessarily* a bad thing. If the current solution points are very poor, it is desirable to explore entirely new regions of the solution space. Early on in the search, when the population of solution points is likely to be poor, *only* sampling nearby points may result in a very slow search. While a *guided* search should incrementally build up a path to the final solution, it may often be useful to sample many entirely separate regions of the solution space. For each of these distant solution points, nearby solution points should be sampled in an attempt to incrementally build up a path, at least until a promising region is found. Even a guided search may therefore at least *partially* resemble a brute-force search *early on.*

It is perhaps helpful to visualise this process in terms of *clusters* of solution points. An initial population of solution points is likely to be scattered randomly throughout the search space. As the GA is much more likely to select the best of these for crossover, solution points will start to cluster around them as the search progresses. The distance between current and newly sampled solution points should be dependent on the quality of the current solution point. Initially, the clusters will be very large and ill-defined, but once a good region of the solution space is found, then these clusters should *ideally* become smaller and more focused.

Occasionally sampling far away solution points, even when the search is quite advanced may also be useful, as it may reduce the chance of becoming stuck in a local optimum. The majority of new solution points still need to be close to the current points - in other words, they need to be within the cluster - in order to incrementally seek out a path to the final solution.

8.3.2 Encoding Schemes

The second point, that encoding schemes should be used where fit individuals belong to short and low order schemata, is also an important result from Schema Theory. Zebulum [58] states GA users rarely follow this guideline, as in many cases it will not be obvious if a particular encoding scheme is likely to result in an abundance of low and short order schemata. More complex applications typically require more complex encoding schemes.

However, as discussed in section 7.5, the encoding scheme plays a relatively insignificant role in determining the schemata available to the GA. Far more important is the application itself, as is the fitness function. What the user is asking of the GA chiefly determines the available schemata.

Re-stated in other terms, if individuals in the GA population belong to long and high order schemata, they are sensitive to change. The sensitivity of these individuals to change depends upon what the individuals represent and what is required of them.

When applied to analogue circuits, therefore, the nature of the circuits being evolved will predominantly determine the schemata available. The user does not have a great deal of choice in this, the user can do very about little the sensitivity of analogue circuits to change.

8.3.3 Low Alphabet Cardinality

The third point, that a low alphabet cardinality should be used, is only directly applicable to particular encoding schemes, as the concept of alphabet cardinality may not make much sense in the context of more complex encoding schemes. The reason for recommending a low alphabet cardinality, however, is to increase the number of schemata available to the GA which will maximise the implicit parallelism. Again, Zebulum [58] points out that for a great many problems, binary strings do not allow for a natural encoding scheme and would lead to an overly complex mapping.

What low alphabet cardinality actually achieves is a high number of *'don't care'* points. A 'point' in this context is a parameter or characteristic that defines an individual in the GA population. This is a property that allows a high level of implicit parallelism, and it is possible that more complex encoding schemes could achieve a similar effect.

There are two ways in which a large number of 'don't care' points lead to high implicit parallelism. The first is that by sampling a single solution point, the GA is in effect sampling all the *individuals* that belong to that schemata. The second is that the GA will also sample other *schemata* that contain the current one as a subset, and each of those schemata represents a group of solutions. So, for each solution point sampled, many other points will be sampled simultaneously if there are many low order schemata, that is, if there are a lot of 'don't care' points. It will sample a set of sets of solution points.

Increasing the number of ‘don’t care’ points will also reduce the sensitivity to change. If an individual in the GA solution possesses a large number of parameters or characteristics which have little or no impact on the quality of solution that individual provides, then it is possible to change that individual in many ways while maintaining the quality of that solution.

8.4 Future View

The successful *synthesis* of analogue circuits, while highly desirable, does not address the whole problem. Before an IC containing an analogue circuit can be manufactured, the circuit must be laid out. While this is strictly outside the scope of synthesis, it is the next natural step in automating analogue design. It is also true that there is a less distinct separation between synthesis and layout for analogue circuits than for digital, as the function of analogue circuits is more likely to be affected by their layout.

8.4.1 Dynamic GA Parameter Variation

Mutation, and in particular crossover, play a vital role in finding a viable circuit topology. However, circuits are highly sensitive to change and once a good topology is found these same operators can be highly destructive. A possible solution to this would be to modify GA parameters during its execution. When a viable topology has been obtained, greatly reducing the crossover rate may result in faster GA convergence. Other possibilities include halting the GA and switching to a more conventional optimisation algorithm, or even switching to a fixed-topology GA implementation.

However, in order to implement any of these ideas, there must be some mechanism in place to determine *when* a viable topology is found. This is likely to be a very difficult problem.

8.4.2 Predation Operator

It is not clear whether the predation operator was particularly beneficial in helping to guide the search and get the GA out of local minima. Certainly, the new, random circuits produced by the operator were poor and a great many of them could not even be simulated. When used at a low rate, as it was in the experiments presented in Chapter 7, it is unlikely that the new circuits will in general be any better than the old ones. Circuits which SPICE could not analyse were always present in the GA population, and the vast majority of the circuits replaced by predation indeed would be these circuits. If used at a much higher rate, it is very likely that this operator would hinder the search, by effectively undoing much the work that the GA had already done, particularly in later generations.

The point of this operator is to introduce 'fresh blood' into the population. But, because it does so randomly, most of this fresh blood will be very poor and so swiftly be rejected. Any automatically generated brand new material added (by any means, method or genetic operator) would almost certainly be random in some way. However, replacing entire circuits is a rather coarse way of doing this. *Partially* replacing individuals may less destructive, for example an operator similar to Koza's implementation of the mutation operator [26] where random sub-trees within an individual are selected for deletion and a randomly generated subtree is inserted in place of it. Or, possibly a 'more active' version of the mutation operator presented in section 6.5.3 could be developed whereby several functions or attributes are mutated each time the operator is executed, rather than simply changing one.

8.4.3 Topology Generation

The automatic *synthesis* (topology generation) of analogue circuits remains an unsolved problem. Certainly, the completely or almost completely unrestricted synthesis of these circuits faces many problems. It is perhaps possible that better results may be obtained from *restricted* synthesis. Providing the GA with information about known good topologies and subcircuits is not a completely new idea [50], however there has been no detailed research in this area.

A series of experiments could be carried out that begin with the completely unrestricted synthesis of a well-understood type of analogue circuit. Subsequent experiments could then be carried out, with each one being increasingly restricted, giving the GA both more predefined knowledge of good topologies, and less freedom to explore the design space. The GA system presented in chapter 6 would be well suited to this and would allow for a very fine grained series of experiments. Although this may greatly limit the potential of the GA to discover truly novel circuits (a characteristic of GAs advocated by Thompson [53]), it may possibly allow for a much more *practically useful* analogue synthesis system.

8.4.4 Fitness Functions

The pole-zero fitness function could be used in the evolution of more complex circuits than passive filters. For example, circuits which would otherwise be very difficult to simulate or measure with traditional fitness functions, such as oscillators, may benefit from this approach. Unstable circuits can be very difficult to simulate with SPICE-like simulators, and often need a human user to 'kick-start' the circuit by setting appropriate initial conditions. In addition to this, the more traditional shape-fitting fitness functions used by GAs would be unsuitable in determining a measure of error. A more direct measure of error would be required, one which could at least more directly determine the frequency of oscillation, for example.

A pole-zero representation of the circuits behaviour would be much easier to deal with, however. Unstable circuits have poles on the right hand side of a pole-zero plot. This may therefore form the basis of a fitness function which could be used in the evolution of much more complex circuits such as voltage controlled oscillators (VCOs), or phase locked loops (PLLs).

Multi-objective fitness functions would also seem to be a practical requirement for GAs to produce realistically useful circuits. There are many aspects of an analogue circuit which are important, and the GA needs to be aware of all of them. However, it seems likely that such fitness functions would further increase the sensitivity of circuits to change, and in doing so further reduce the effectiveness (increase the resource requirements) of the GA. A series of experiments which evolve a given circuit, but which make use of an increasingly detailed and demanding fitness function could be carried out.

Further to this, the reduction of redundant components could be specifically targeted. Sripramong [50] makes use of a multi-objective fitness function, and uses a pruning technique to remove unused components. However, if the fitness function was specifically designed to penalise the presence of such components, this may have a serious impact of the operation of the GA.

Redundant components have the effect of introducing shorter, lower order schemata into the population. The importance of this to the operation of the GA could be investigated - what effect would eliminating them have?

Appendices

Appendix A

EXPERIMENT RUN-TIME

A.1 Computing Hardware Used

The Genetic Algorithm system presented in this thesis was run on two separate machines.

- Dual Intel Xeon CPUs, 2.8GHz, 2GB RAM, running Fedora Core 2 Linux.
- An Intel Core 2 Duo (dual core), 2.13GHz, 512MB RAM, running Fedora Core 5 Linux.

A.2 Execution Times

The execution time for some of the experiments was not recorded.

A.2.1 Initial Experiments

Chebyshev							
	Run	**Machine**	**Generations**	**Seconds**	**Minutes**	**Hours**	**Days**
4th Order	1	Core 2	113	19684	328.07	5.47	0.23
	2	Core 2	961	157707	2628.45	43.81	1.83
	3	Core 2	113	19589	326.48	5.44	0.23
	4	Core 2	124	21523	358.72	5.98	0.25
	5	Core 2	633	106109	1768.48	29.47	1.23
						Total:	3.76
5th Order	1	Core 2	46	7549	125.82	2.10	0.09
	2	Core 2	241	40145	669.08	11.15	0.46
	3	Core 2	1000	158601	2643.35	44.06	1.84
	4	Core 2	515	110676	1844.60	30.74	1.28
	5	Core 2	122	19696	328.27	5.47	0.23
						Total:	3.90
6th Order	1	Core 2	1000	164604	2743.40	45.72	1.91
	2	Core 2	1000	159866	2664.43	44.41	1.85
	3	Core 2	353	109743	1829.05	30.48	1.27
	4	Core 2	1000	148942	2482.37	41.37	1.72
	5	Core 2	88	14782	246.37	4.11	0.17
						Total:	6.92

Table A.1: Summary of Initial Experiments Run Times

A.2.2 GA Sensitivity Experiments

Population Size							
	Run	**Machine**	**Generations**	**Seconds**	**Minutes**	**Hours**	**Days**
50	1	Xeon	1000	8471	141.18	2.35	0.10
	2	Xeon	1000	8480	141.33	2.36	0.10
	3	Xeon	1000	19727	328.78	5.48	0.23
	4	Xeon	1000	19774	329.57	5.49	0.23
	5	Xeon	1000	19766	329.43	5.49	0.23
						Total:	0.88
100	1	Core 2	1000	35051	584.18	9.74	0.41
	2	Core 2	1000	31847	530.78	8.85	0.37
	3	Core 2	1000	34150	569.17	9.49	0.40
	4	Core 2	1000	31757	529.28	8.82	0.37
	5	Core 2	1000	31814	530.23	8.84	0.37
						Total:	1.91
150	1	Core 2	1000	59125	985.42	16.42	0.68
	2	Xeon	1000	38671	644.52	10.74	0.45
	3	Core 2	1000	48303	805.05	13.42	0.56
	4	Core 2	1000	48575	809.58	13.49	0.56
	5	Core 2	1000	50103	835.05	13.92	0.58
						Total:	2.83
200	1	Core 2	905	56848	947.47	15.79	0.66
	2	Core 2	1000	65157	1085.95	18.10	0.75
	3	Core 2	1000	61209	1020.15	17.00	0.71
	4	Core 2	1000	60178	1002.97	16.72	0.70
	5	Core 2	361	27406	456.77	7.61	0.32
						Total:	3.13
250	1	Core 2	290	24679	411.32	6.86	0.29
	2	Xeon	790	51122	852.03	14.20	0.59
	3	Core 2	419	30913	515.22	8.59	0.36
	4	Core 2	1000	69823	1163.72	19.40	0.81
	5	Core 2	1000	80093	1334.88	22.25	0.93
						Total:	2.97

Table A.2: Summary of Population Size Experiments Run Times

Predation							
	Run	**Machine**	**Generations**	**Seconds**	**Minutes**	**Hours**	**Days**
Normal Cull	1	Core 2	1000	309707	5161.78	86.03	3.58
	2	Core 2	1000	307575	5126.25	85.44	3.56
	3	Core 2	1000	338217	5636.95	93.95	3.91
	4	Core 2	1000	330938	5515.63	91.93	3.83
	5	Core 2	1000	339301	5655.02	94.25	3.93
						Total:	18.82
Single Cull	1	Core 2	984	333205	5553.42	92.56	3.86
	2	Core 2	317	107148	1785.80	29.76	1.24
	3	Core 2	224	72788	1213.13	20.22	0.84
	4	Core 2	100	31817	530.28	8.84	0.37
	5	Core 2	100	31531	525.52	8.76	0.36
						Total:	6.67

Table A.3: Summary of Predation Type Experiments Run Times

Crossover							
	Run	**Machine**	**Generations**	**Seconds**	**Minutes**	**Hours**	**Days**
0%	1	Xeon	1000	137989	2299.82	38.33	1.60
	2	Xeon	1000	176987	2949.78	49.16	2.05
	3	Xeon	1000	205031	3417.18	56.95	2.37
	4	Xeon	1000	204904	3415.07	56.92	2.37
	5	Xeon	1000	205987	3433.12	57.22	2.38
						Total:	10.77
10%	1	Xeon	106	9097	151.62	2.53	0.11
	2	Xeon	1000	87617	1460.28	24.34	1.01
	3	Xeon	225	31803	530.05	8.83	0.37
	4	Xeon	1000	130585	2176.42	36.27	1.51
	5	Xeon	1000	-	-	-	-
						Total:	3.00
90%	1	Core 2	1000	150403	2506.72	41.78	1.74
	2	Core 2	1000	143944	2399.07	39.98	1.67
	3	Core 2	1000	144515	2408.58	40.14	1.67
	4	Core 2	1000	229051	3817.52	63.63	2.65
	5	Core 2	135	21363	356.05	5.93	0.25
						Total:	7.98
100%	1	Core 2	109	20125	335.42	5.59	0.23
	2	Core 2	1000	137305	2288.42	38.14	1.59
	3	Core 2	1000	-	-	-	-
	4	Core 2	82	12521	208.68	3.48	0.14
	5	Core 2	56	11011	183.52	3.06	0.13
						Total:	2.09

Table A.4: Summary of Crossover Experiments Run Times

Mutation	Run	Machine	Generations	Seconds	Minutes	Hours	Days
0%	1	Core 2	1000	131528	2192.13	36.54	1.52
	2	Core 2	1000	142942	2382.37	39.71	1.65
	3	Core 2	1000	175058	2917.63	48.63	2.03
	4	Core 2	1000	119858	1997.63	33.29	1.39
	5	Core 2	1000	172063	2867.72	47.80	1.99
						Total:	8.58
100%	1	Core 2	214	25688	428.13	7.14	0.30
	2	Core 2	1000	164597	2743.28	45.72	1.91
	3	Core 2	55	6431	107.18	1.79	0.07
	4	Core 2	1000	167884	2798.07	46.63	1.94
	5	Core 2	125	15016	250.27	4.17	0.17
						Total:	4.39

Table A.5: Summary of Mutation Experiments Run Times

A.2.3 Fitness Function Comparison Experiments

Shape Fitting	Run	Machine	Generations	Seconds	Minutes	Hours	Days
Chebyshev	1	Core 2	1000	153389	2556.48	42.61	1.78
	2	Core 2	1000	154235	2570.58	42.84	1.79
	3	Core 2	1000	165680	2761.33	46.02	1.92
	4	Core 2	514	87916	1465.27	24.42	1.02
	5	Core 2	1000	166575	2776.25	46.27	1.93
						Days Total:	8.42

Table A.6: Summary of Fitness Function Comparison Experiments Run Times

A.2.4 Active Filter Experiments

Active Filter	**Run**	**Machine**	**Generations**	**Seconds**	**Minutes**	**Hours**	**Days**
Sallen-Key (Transistors)	1	Core 2	1000	208666	3477.77	57.96	2.42
	2	Core 2	1000	158847	2647.45	44.12	1.84
	3	Core 2	1000	-	-	-	-
	4	Core 2	1000	171064	2851.07	47.52	1.98
	5	Core 2	1000	171021	2850.35	47.51	1.98
						Days Total:	8.21
Sallen-Key (Op-Amps)	1	Core 2	19	2805	46.75	0.78	0.03
	2	Core 2	13	1893	31.55	0.53	0.02
	3	Core 2	11	1433	23.88	0.40	0.02
	4	Xeon	13	2275	37.92	0.63	0.03
	5	Xeon	20	3535	58.92	0.98	0.04
						Days Total:	0.14

Table A.7: Summary of Active Filter Experiments Run Times

References

[1] Intel Corporation (No date). The Intel 4004 Microprocessor [online]. Available from: http://www.intel.com/museum/archives/4004facts.htm [Accessed 13th April 2009].

[2] Intel Corporation (No date). World's First 2-Billion Transistor Microprocessor [online]. Available from: http://www.intel.com/technology/architecture-silicon/2billion.htm [Accessed 13th April 2009].

[3] Rob Storey (No date). IBM's Standard Modular System (SMS) cards [online]. Available from: http://members.optushome.com.au/intaretro/SMSCards.htm [Accessed 13th April 2009].

[4] Texas Instruments Corporation (No date). The Chip That Jack Built [online]. Available from: http://www.ti.com/corp/docs/kilbyctr/jackbuilt.shtml [Accessed 13th April 2009].

[5] Computerworld (July 20, 2009). NASA's Apollo technology has changed history [online]. Available from: http://www.computerworld.com/s/article/9135690/NASA_s_Apollo_technology_has_changed_history?taxonomyId=11&pageNumber=2 [Accessed 12th Jan 2010].

[6] National Aeronautics and Space Administration (NASA) (December 21, 2004). The Apollo Flight Journal. Available from: http://history.nasa.gov/afj/compessay.htm [Accessed 12th Jan 2010].

[7] Institute of Electrical and Electronics Engineers (IEEE) (September 8, 2008). Integrated Circuits and the Space Program and Missile Defense. Available from: http://www.ieeeghn.org/wiki/index.php/Integrated_Circuits_and_the_Space_Program_and_Missile_Defense [Accessed 12th Jan 2010].

[8] Synopsys, Inc. (No date). HSPICE [online]. Available from: http://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSPICE /Pages/default.aspx [Accessed 13th April 2009].

[9] P. E. Allen. *CMOS Analog Circuit Design*. Saunders College Publishing/Harcourt Brace, 1stedition, 1987.

[10] P. J. Ashenden, G. D. Peterson, and D. A. Teegarden. *The System Designer's Guide To VHDL-AMS*. Morgan-Kaufman, 2003.

[11] P. J. Ashenden, G. D. Peterson, and D. A. Teegarden. *The System Designer's Guide To VHDL-AMS*, pages 5–7. Morgan-Kaufman, 2003.

[12] W. F. Brinkman, D. E. Haggan, and W. W. Troutman. A history of the invention of the transistor and where it will lead us. *IEEE Journal Of Solid-State Circuits*, 32(12), December 1997.

[13] A. Doboli, A. Nunez-Aldana, N. Dhanwada, S. Ganesan, and R. Vemuri. Behavioral synthesis of analog systems using two-layered design space exploration. Technical report, University of Cincinnati, 1999.

[14] A. Doboli and R. Vemuri. A vhdl-ams compiler and architecture generator for behavioral synthesis of analog systems. Technical report, University of Cincinnati.

[15] A. Doboli and R. Vemuri. Behavioral modeling for high-level synthesis of analog and mixed-signal systems from vhdl-ams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(11):1504–1520, November 2003.

[16] G. Doménech-Asensi, T. J. Kazmierski, J. D. Ruiz-Marin, and R. Ruiz-Merino. Architectural synthesis of high-level analogure vhdl-ams descriptions using netlist extraction from parse trees. Electronics Letters, Vol. 36, No. 20.

[17] F. El-Turky and E. E. Perry. Blades: An artificial intelligence approach to analog circuit design. *IEEE Transactions on Computer-Aided Design*, 8(6), June 1989.

[18] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Springer, 2006.

[19] R. Harjani, R. A. Rutenbar, and L. R. Carley. Oasys: A framework for analog circuit synthesis. *IEEE Transactions on Computer-Aided Design*, 8(12), December 1989.

[20] J. Holland. *Adaptation In Natural And Artificial Systems.* The University Of Michigan, 1975.

[21] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1076-1997 edition, 1987.

[22] IEEE. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, 1364-1995 edition, December 1995.

[23] T. J. Kazmierski and F. A. Hamid. Analogue integrated circuit synthesis from vhdl-ams behavioural specifications. In *Proceedings of the 23rd International Conference on Microelectronics (MIEL 2002)*, volume 2, pages 585–588, 12-15 May 2002.

[24] R. KielkowSki. *Inside SPICE: Overcoming The Obstacles Of Circuit Simulation.* McGraw-Hill, 1994.

[25] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598), May 13 1983.

[26] J. R. Koza. *Genetic Programming.* MIT Press, 1992.

[27] J. R. Koza, I. F. H. Bennett, D. Andre, and M. A. Keane. Evolution of a 60 decibel op amp using genetic programming. In *Proceedings of the First International Conference on Evolvable Systems*, volume 1259, pages 455–469, 1996.

[28] J. R. Koza, I. F. H. Bennett, D. Andre, and M. A. Keane. Reuse, parameterized reuse, and hierarchical reuse of substructures in evolving electrical circuits using genetic programming. In *Proceedings of the First International Conference on Evolvable Systems*, volume 1259, pages 312–326, 1996.

[29] J. R. Koza, I. F. H. Bennett, D. Andre, and M. A. Keane. Genetic programming: Biologically inspired computation that creatively solves non-trivial problems. In *Evolution as Computation, DIMACS Workshop, Princeton, Jan 1999*, pages 95–124, 2001.

[30] J. R. Koza, I. F. H. Bennett, D. Andre, M. A. Keane, and F. Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*, 1(2), July 1997.

[31] E. Kreyszig. *Advanced Engineering Mathematics*, pages 1084–1087. John Wiley & Sons, 7$^{\text{th}}$edition, 1993.

[32] E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, 1993.

[33] V. J. Kublin. The micro module program. In *14th Annual Symposium on Frequency Control*, pages 217–241, 1960.

[34] C. Mattiusi and D. Floreano. Evolution of analog networks using local string alignment on highly reorganizable genomes. In *Proceedings of the 2004 NASA/DoD Conference on Evolution Hardware (EH'04))*, 2004.

[35] C. Mattiusi and D. Floreano. Analog genetic encoding for the evolution of circuits and networks. *IEEE Transactions on Evolutionary Computation*, 11(5):596–607, October 2007.

[36] C. Mattiussi. *Evolutionary Synthesis Of Analog Networks*. PhD thesis, Institute Of Systems Engineering, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2005.

[37] C. A. Mead and L. Conway. *Introduction To VLSI Systems*. Addison Wesley, 1980.

[38] C. A. Mead and B. Hoeneisen. Fundamental limitations in microelectronics - i. mos technology. *Solid-State Electronics*, 15(7):819–829, 1972.

[39] P. Mitros. A framework for analog circuit optimization. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994.

[40] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 19 1965.

[41] A. Nunez-Aldana and R. Vemuri. An analog performace estimator for improving the effectiveness of cmos analog systems circuit synthesis. In *Proceedings of Design, Automation & Test in Europe (DATE 1999)*, 1999.

[42] E. S. Ochotta, L. R. Carley, and R. A. Rutenbar. Analog circuit synthesis for large, realistic cells: Designing a pipelined a/d converter with astrx/oblx. In *IEEE Custom Integrated Circuits Conference*, pages 365–368, 1994.

[43] E. S. Ochotta, R. A. Rutenbar, and L. R. Carley. Astrx/oblx: Tools for rapid synthesis of high-performance analog circuits. In *31stACM/IEEE Design Automation Conference*, pages 24–30, 1994.

[44] E. S. Ochotta, R. A. Rutenbar, and L. R. Carley. Synthesis of high-perfomance analog circuits in astrx/oblx. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(3), March 1996.

[45] L. T. Pillage and R. A. Rohrer. Asymptotic waveform evaluation for timing analysis. *IEEE Transactions on Computer-Aided Design*, 9(4):352–366, April 1990.

[46] R. J. Plassche, J. H. Huijsing, and W. M. C. Sansen. *Analog Circuit Design: High-speed Analog-to-Digital Converters, Mixed Signal Design, PLLs and Synthesizers*, pages 177–181. Springer, 2000.

[47] V. Raghavan, R. A. Rohrer, L. T. Pillage, J. Y. Lee, J. E. Bracken, and M. M. Alaybeyi. Awe-inspired. In *Proceedings of the Custom Integrated Circuits Conference (IEEE 1993)*, volume 18, pages 1–8, May 1993.

[48] P. Seibel. *Practical Common LISP*. Apress, 2005.

[49] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 2$^{\text{nd}}$edition, 1999.

[50] T. Sripramong and C. Toumazou. The invention of cmos amplifiers using genetic programming and current-flow analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11), November 2002.

[51] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In *Proceedings of the First International Conference on Evolvable Systems*, volume 1259, pages 390–405, 1996.

[52] A. Thompson. Through the labyrinth evolution finds a way: A silicon ridge. In *Proceedings of the First International Conference on Evolvable Systems*, volume 1259, pages 406–421, 1996.

[53] A. Thompson, P. Layzell, and R. S. Zebulum. Explorations in design space: Unconventional electronics design through artificial evolution. *IEEE Transactions on Evolutionary Computation*, 3(3):167–196, September 1999.

[54] M. P. Vecchi and S. Kirkpatrick. Global wiring by simulated annealing. *IEEE Transactions on Computer-Aided Design*, CAD-2(4):215–222, October 1983.

[55] R. Walker and N. Tersini. *Silicon Destiny - The Story Of Application Specific Integrated Circuits And LSI Logic Corporation*, page 15. C.M.C. Publications, 1992.

[56] R. Walker and N. Tersini. *Silicon Destiny - The Story Of Application Specific Integrated Circuits And LSI Logic Corporation*, pages 151–152. C.M.C. Publications, 1992.

[57] S. Winder. *Filter Design.* Newnes, 1997.

[58] R. S. Zebulum, M. A. C. Pacheco, and A. M. B. R. Vellasco. *Evolutionary Electronics.* CRC Press, 2002.

[59] R. S. Zebulum, M. A. C. Pacheco, and A. M. B. R. Vellasco. *Evolutionary Electronics*, pages 41–44. CRC Press, 7th edition, 2002.