Case-Based Reasoning and Evolutionary Computation Techniques for FPGA programming

Dominic Edward Job

A thesis submitted in partial fulfillment of the requirements of Napier University for the degree of Doctor of Philosophy

School of Computing,
Napier University

1st Oct 2001

Declaration

I declare that the work described in this thesis has not formed part of a submission for any other award, and that the work is entirely my own.

Signature

[illegible]

Dominic E. Job
1st October 2001

Notes on publications and contributions:

This information is given to clarify work done by individuals. All other significant work in these papers was collaborative.

The first paper covering the authors work was Miller J., T. Kalganova, N. Lipnitskaya and D. Job, (Miller et al., 1999b). This paper introduced the idea of using an evolutionary algorithm as a method of discovering new principles of design. It proposed that by examining programs of gradually increasing scale that new principles of design may be discovered. D. Job and J.F. Miller carried out this work. The work on Multivariate Logic was carried out by was J.F. Miller, T. Kalganova and N. Lipnitskaya. D. Job carried out the work on principle identification from evolved data and the reuse of evolved data and identified principles.

The next three papers were by Dominic Job, Venky Shankararaman and Julian Miller (Job et al., 1999a). The first paper takes a closer look at the problems involved in Evolving digital circuit programs from the point of view of software reuse. The problems of errors and scaling are introduced with some preliminary results. The second paper (Job et al., 1999b), looked at the problems involved in identifying and reusing principles from collections of evolved programs. The use of CBR technology to facilitate the solution to this problem was examined. The last paper of 1999 (Job et al., 1999c), looks at the problems of constructing a Case-Base from evolved data with no obvious modules and early work into error repair in evolved programs. Dominic Job was the primary author of these papers. V. Shankararaman and J. Miller contributed expertise in CBR and Evolvable Hardware, respectively.

The first paper of 2000 by J. F. Miller, D. Job, and V. K. Vassilev (Miller et al., 2000a, 2000b) was published in two parts. Part 1 (Miller et al., 2000a), gives and in depth view of an evolutionary algorithm as an engine for discovering new designs. It discusses the idea of identifying new, efficient, and generalisable principles of design. An in depth view of evolving designs for arithmetic circuits is given and the background to support the evolutionary approach to digital circuit design is given. The examination of conventional digital circuit design techniques was primarily by J.F. Miller. Part 2 (Miller et al., 2000b) presents a two fold approach to principle identification and reuse. These two approaches analyse the evolved data phenotypically and genotypically by processes of data-mining and landscape analysis. D. Job researched data mining from collections of evolved data and V.K. Vassilev examined the evolutionary process used to evolve the data, through landscape analysis.

The next paper Vesselin K. Vassilev, Dominic Job and Julian F. Miller (Vassilev et al., 2000), was primarily the work of V.K. Vassilev. Using landscape analysis techniques V.K. Vassilev showed that it was possible to estimate the size of the most efficient digital circuit for several problems. All three authors contributed towards a method that guarantees that the evolutionary algorithm produces a functionally correct circuit. This method is referred to in this thesis as 'seeding with the conventional solution'. J.F. Miller wrote the original software used to evolve digital circuit programs. D. Job subsequently made efficiency improvements to this software, and made changes to this software to enable the 'seeding with the conventional solution' and optimising for number of components experiments to be carried out for this paper. D. Job carried out the experiments involving the examination of the increase in time taken and computing power required by problems of increasing scale. V.K. Vassilev wrote the software for landscape analysis and also carried out the experiments to estimate the minimum possible size for each multiplier program examined.

The final paper covering material in this thesis was written by D. Job and V. Shankararaman (Job and Shankararaman, 2000). This paper looks at the use of CBR techniques to adapt evolved digital circuit programs to solve problems of increased scale. D. Job developed this work. V. Shankararaman contributed to the writing of the paper and expertise in CBR.

Acknowledgements

I would like to thank Dr. Venky Shankararaman for many years of valuable support and supervision, from my MSc years through to this PhD, even after leaving Napier, and for introducing me to Case-Based Reasoning. Thanks also to Dr. Les Morss, my current supervisor who took on the difficult task of supervisor when Venky left Napier in 1998. I would like to thank Professor Terry Fogarty for his support during my time at Napier. Thanks also to Professor David Benyon, my thesis chairman.

A great many thanks to Dr. Vesselin K. Vassilev and Dr. Julian F. Miller, with whom I worked closely for several years during my PhD, who inspired, corrected, supported and humored me with many hours of priceless discussions, incredible inventions, and relentless hard work.

Thanks also to all of my friends at home and at Napier (If any of you read further than this) for putting up with my vacant expressions and meaningless babbles for the past 4 years.

Thanks to Steve Corley at British Telecommunications for a great deal of patience, useful feedback and the fun times at BT laboratories, Ipswich (now Adastral Park).

And last but not least, thanks to Mum and Dad for the unquestioning support throughout my university years.

The author would like to thank British Telecommunications for supporting this research work.

Contents:

Abstract

A problem in Software Reuse (SR) is to find a software component appropriate to a given requirement. At present this is done by manual browsing through large libraries which is very time consuming and therefore expensive. Further to this, if the component is not the same as, but similar to a requirement, the component must be adapted to meet the requirements. This browsing and adaptation requires a skilled user who can comprehend library entries and foresee their application. It is expensive to train users and to produce these documented libraries. The specialised software design domain, chosen in this thesis, is that of Field Programmable Gate Arrays (FPGAs) programs. FPGAs are user programmable microchips that have many applications including encryption and control. This thesis is concerned with a specific technique for FGPA programming that uses Evolutionary Computing (EC) techniques to synthesize FPGA programs.

Evolutionary Computing (EC) techniques are based on natural systems such as the life cycle of living organisms or the formation of crystalline structures. They can generate solutions to problems without the need for complete understanding of the problem. EC has been used to create software programs, and can be used as a knowledge-lean approach for generating libraries of software solutions. EC techniques produce solutions without documentation. To automate SR it has been shown that it is essential to understand the knowledge in the software library. In this thesis techniques for automatically documenting EC produced solutions are illustrated. It is also helpful to understand the principles at work in the reuse process. On examination of large collections of evolved programs it is shown that these programs contain reusable modules. Further to this, it is shown that by studying series of similar software components, principles of scale can be deduced. Case Based Reasoning (CBR) is a problem solving method that reuses old solutions to solve new problems and is an effective method of automatically reusing software libraries. These techniques enable automated creation, documentation and reuse of a software library.

This thesis proposes that CBR is a feasible method for the reuse of EC designed FPGA programs. It is shown that EC synthesised FPGA programs can be documented, reused, and adapted to solve new problems, using automated CBR techniques.

1. Introduction

This thesis proposes that Case-Based Reasoning (CBR) is a feasible method for the reuse of Evolutionary Computing (EC) designed Field Programmable Gate Array (FPGA) programs. It is shown that EC synthesised FPGA programs can be documented, reused, and adapted to solve new problems automatically, using CBR techniques.

FPGA programs are a restricted form of software. An important objective of software development is to enable greater productivity and quality in the development process. One solution to the problem is to develop tools and techniques that reuse existing, established software components, analyses, designs and documentation (Mili, 1995), providing a level of automation.

A large cost in Software Reuse (SR) is the creation and maintenance of software libraries. Automated generation of software artefacts and an automated generation of the understanding of the potential reuse of these software artefacts are required to avoid having engineers manually encode software artefacts and information on their potential for reuse (Mili, 1995). Further to this, a retrieved software component may not exactly match the desired specifications. In this situation a browsing tool alone is not sufficient and tools for adapting the retrieved software component to meet the required specifications must be provided. This work looks at automated generation and comprehension of software artefacts and their automated reuse.

The aim of this research is to automatically generate a software library of FPGA programs and to produce an automated method for FPGA program reuse. This aim involves the development of a method for automatic extraction and application of principles that can be used for SR. SR is a domain that is characterised by complex examples. The complex system is that of software programs generated by the evolutionary computing technique, Cartesian Genetic Programming (CGP). In the case study presented here the software is intended for use on FPGAs. In this work CBR techniques have been applied to the problems involved with SR in this field.

Using EC techniques to generate software programs for the FPGA is computationally expensive. EC techniques grow prohibitively computationally expensive for larger FPGA programs. The FPGA programs in this research are limited in their mechanics compared to notion of traditional software programs, as they have no loops or states (memory). FPGA programs are a limited form of program consisting of a feed forward network of primitive logic functions. In this thesis evolved FPGA programs are reused to create an initial Case-Base to allow CBR techniques to create larger FPGA programs at reasonable computational expense.

An automated method for FPGA program reuse could be achieved by automatically identifying the principles of design that emerge from examination of large collections of small FPGA programs, and applying these principles to create larger, more complex programs.

The field of software reuse can be generally split into two areas of reuse. These two areas are the reuse of software *products* and *processes*. In this thesis an examination of the reuse of *products* generated by EC techniques is presented. It is shown that EC techniques can be used to create a software library that can be reused by the use of CBR techniques, in an automated fashion.

CBR is an alternative to rule-based and model-based reasoning and has several advantages: it can provide answers to problems in poorly understood complex domains; it does not require a domain model or domain rules; and it can provide an explanation of its reasoning. The reuse of old solutions to solve new problems is the problem facing software reuse, as many old solutions need to be identified and adapted to suit new problems. CBR can provide selection, retrieval and adaptation of old software solutions to solve new problems. Presented in this work is an investigation of CBR, EC and SR methods, which supports the creation, understanding, reuse, and adaptation of software artefacts in automated software programming.

This thesis shows the development of methods to enable the latest advances in CBR automated adaptation techniques to be applied to the reuse of software Cases that have been automatically generated by EC. These Cases have unstructured solutions and have no obvious reuse or design components encoded into them. Techniques for automated identification and application of reuse principles are presented as new techniques in software reuse. This is a significant advance over existing applications of these techniques to Cases with simple numerical atomic and linear solutions, or solutions that have a clear design and understanding built into them.

The high level aim of this thesis is to show how CBR could be applied to EC Software Reuse, its strengths and weaknesses. The ease with which these techniques can be applied to new fields is a subject for future work. This aim intends to illustrate how the evolutionary process could help automate a reasoning process, introducing the idea of evolutionary reasoning. It is of interest to determine how much effort may be required to prepare such a system to work in a new software domain as this gives a measure of portability and the generality of the technique.

In this thesis EC techniques are used to automate the synthesis programs for Field Programmable Gate Arrays (FPGAs). In Chapter 2 section 2 it is shown that there is no complete set of techniques for designing any FPGA program and that EC provides the most general technique for FPGA program design. These programs are processed to create a software library in the form of a Case-Base (CB). This CB is then subjected CBR techniques to provide retrieval, understanding, principle extraction, adaptation and reuse. Figure 1 shows how EC techniques are used to synthesise FPGA programs that are used to construct a CB of programs that can be retrieved and adapted in an automated way to produce programs that are too large to evolve by EC techniques alone. In this thesis a combination of EC and CBR are used to solve these problems.

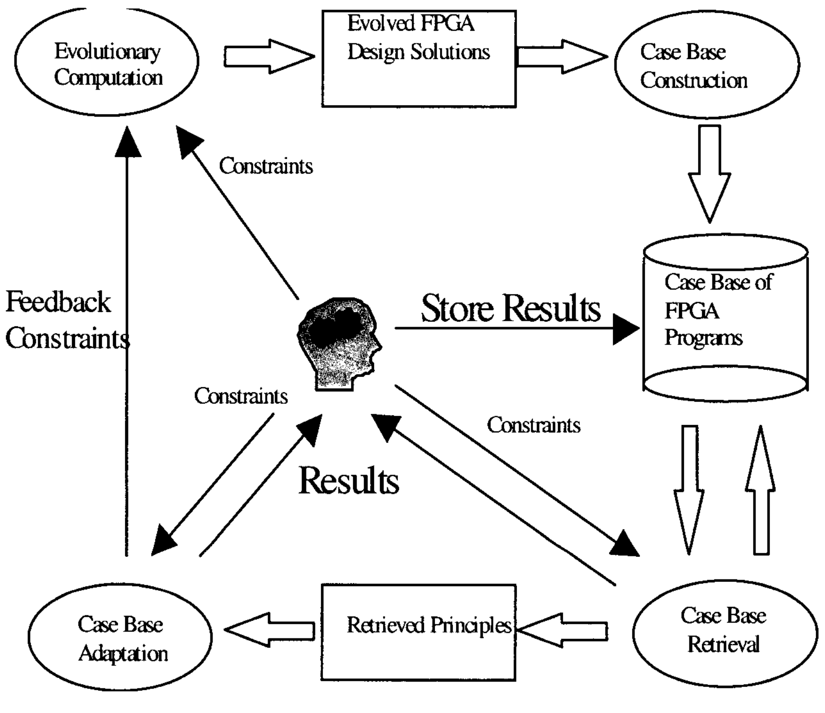


Figure 1. Reusing Evolved Designs using CBR.

1.1. Conclusion

This thesis shows that CBR techniques are a feasible way of reusing evolved FPGA programs. Through the construction and analysis of a CB it is shown that evolved FPGA programs are modular in design. It is argued in this thesis that by studying evolved designs of gradually increasing scale, it may be possible to identify new, efficient, and *generalisable* principles of design. It is shown that by studying evolved designs of gradually increasing scale, that it is possible to design new efficient programs through the reuse of existing solutions.

There are several reasons for why this research area is novel. The software reuse domain (evolved programs) is more complex than any domain that has had the technology of automated principle extraction and application, applied to it before. Specifically the Cases involved represent multi-attribute compound problems and solutions, where solutions can be based upon a single or multiple Cases, and the attributes of the Cases are highly interactive.

It is argued in this thesis that CBR is most suited to solving problems in the SR domain. The complexity of the SR domain provides the CBR researcher with a suitable testing ground for improving the adaptation in the CBR technique.

Chapter 2 introduces the subject areas involved in this thesis. A specific area SR, that of FPGA programming, is described. Current research and practices in SR are reviewed. Next the problem of knowledge acquisition is described to show the utility of EC to FPGA SR. The background of EC is discussed in Chapter 2 where it is shown how EC can be a more general design technique than conventional techniques. CBR is then introduced as a technology that when combined with EC can provide a mechanism for automated FPGA program reuse. The ideas behind combinations of EC and CBR are briefly described and then a summary of the thesis is given.

The Chapter 3 discusses a hybrid CBR and EC approach that is intended to improve the adaptation capabilities of the CBR technique. EC provides a knowledge-lean search facility that can produce solutions where CBR fails due to a lack of knowledge (Tanaka *et al.*, 1994).

In chapter 3, CBR is reviewed and examples of CBR and EC hybrids are given. Following this, conventional methods of FPGA programming are reviewed and then the use of EC for FPGA programming is reviewed. In Chapter 3 section 2, the ideas involved in CBR adaptation are reviewed showing the connection between automated principal identification, reuse and CBR adaptation. This is followed by an overview of CBR-EC hybrids.

In Chapter 4 details of EC and practical issues of FPGA programming are discussed. Chapters 5, 6 and 7 discuss and examine EC and CBR techniques for automated FPGA program design and reuse. Chapters 6 and 7 show advances made by this work in combining EC and CBR techniques for FPGA programming. Finally Chapter 8 gives conclusions and suggestions for future work.

2. From Software Reuse to automated FPGA programming

This chapter discusses the problems in software reuse (SR) and the specialised domain of FPGA SR and techniques and problems in designing FPGA programs are discussed. Conventional techniques and an EC approach to FPGA program design are discussed.

2.1. Software Reuse

SR encompasses any technique that reuses software development work that has already been done. When a high level of SR is achieved the costs of development decrease. Also, due to new software being based on tried and tested software, the quality of the software is high, yielding lower maintenance costs. The high quality of the software reduces testing and debugging. SR has been the subject of much research for the past thirty years. As yet no single SR approach has become standard due to the complexity of the problem. SR itself has not become standard practice in software engineering (Krueger, 1992). Most recent research work into SR has concentrated on organisational level issues, and on reuse in High Level Languages, or application specific software generators. Both of these techniques require the costly development of software libraries or knowledge bases to operate (SEKE, 1999).

Mili (1995) emphasises the reuse of both *products* and *processes* and analyses the SR problem from this perspective. Krueger (1992) examines each of these techniques through their reusable "*artefacts*" and the way in which these artefacts are "*abstracted, selected, specialised* and *integrated*".

The effectiveness of a reuse technique is judged via a rule-of-thumb that Krueger (1992) calls *cognitive distance* and is "an intuitive gauge of the intellectual effort required to use the technique". This judgement can then be compared to how much intellectual effort would be required to build the system from scratch.

In general, from a domain point of view, the narrower the domain, the more successful the reuse system; the more general the domain coverage the more reuse is sacrificed. Also, the larger the most suitable artefact for reuse is, the more efficient the reuse becomes. It is easier to reuse one large software artefact than it is to reuse many small artefacts that require assembly.

All of the approaches to reuse benefit as the techniques mature through use and experience.

SR encompasses many techniques. In general these techniques fall into one of the categories give in Table 1. Each technique aims to reuse existing knowledge as much as possible whilst minimising the amount of new work required to produce a satisfactory solution (Krueger, 1992).

SR covers a very wide area, so this review is limited to a discussion of reusing software artefacts themselves. This review does not cover organisational infrastructure or institution wide reuse policies. The following review is aimed at providing an overview of current reuse technologies from an individual or small team of software engineers' perspective.

Three major research papers on software reuse Krueger (1992), Mili (1995) and Biggerstaff (1992) provide an extensive and exhaustive examination of the subject.

The Table 1 summarises the techniques that have been applied to the problem of software reuse, as classified by Krueger (1992):

	Krueger (1992)	Mili (1995)	Biggerstaff (1992)
High-Level Languages	√		√
Design and Code Scavenging	√		
Source Code Components	√	√	√
Software Schema	√	√	
Application Generators	√	√	√
Very High-Level Languages	√	√	√
Transformational Systems	√	√	√
Software Architectures	√		

Table 1. Reuse technologies.

These categories are not clean-cut but give a general view of each area. Mili (1995) emphasises the reuse of both *products* and *processes,* and analyses the SR problem from this perspective. Krueger (1992) examines each of these techniques through their reusable "*artefacts*" and the way in which these artefacts are "*abstracted, selected, specialised* and *integrated*". Krueger compares the effort required to produce a solution using a SR technique to the effort required to produce a completely new solution, as a measure of the level of reuse. Biggerstaff (1992) views the subject from a scale and domain point of view, that the narrower the domain, the more successful the reuse system, the more general the domain coverage the more reuse is sacrificed. All of the approaches will benefit as the techniques mature through use and experience.

Details of the categories of reuse technologies given in Table 1 are given in Appendix 1.

There are several problems in making software reuse systems. The following points highlight the difficulties involved in any SR system. Two main problems with SR are that issues of scale have not been successfully resolved and that most effort has been put into small details and correctness, not into improving productivity and quality (Mili, 1995).

- Knowledge acquisition, specification and evaluation

  It is difficult to specify requirements, difficult to create complete libraries of code, architectures, reuse knowledge and it is difficult to represent this complex information.

- Domain generality trades-off with level of abstraction and performance

  The level of abstraction dictates the effectiveness of the reuse system, as there is a trade-off between horizontal (broad domain) versus vertical reuse, between defining *what* has to be done and *how* it must be done, and between *specification* and *implementation.*

- Finding reusable artefacts in good time

  There is a lack of assistance in finding the relevant artefacts.

- User and machine understanding of the software reuse system present problems:

  How much do users and machines need to know about the artefacts and reuse system? There is a lack of assistance in understanding the reusable artefacts and system, and most systems do not have a good system for describing the behaviour of the reusable software artefacts to the user and machine.

- Integrating and adapting the reusable artefact

  This problem is alleviated by reducing the need for integrating and adapting and by providing assistance in integrating and adapting the reusable artefact.

- Debugging the reusable artefact:

  Reducing the need for debugging by reusing fewer and larger artefacts, and by assisting in debugging the adapted reusable artefact, including the reuse of previous debugging experience.

- Acquisition of new knowledge as additional problems are solved.

This enables the solving of increasingly more complex reasoning tasks through the assessment and evaluation of new Cases and knowledge.

Applying SR to FPGAs

The best solution to the above problems would require a system that uses aspects of all of the reviewed techniques. The system would be able to reuse software artefacts from the code levels through to design levels and even analysis and debugging levels. The main problem would be to create sufficient machine knowledge to enable this. It is difficult to gauge when sufficient knowledge exists to achieve a high level of general SR. One approach would be to use an evolutionary system that can improve iteratively with use and experience of past problems. It would also be most beneficial if the system could justify any of its actions to the programmer, and the reuse system is easy to understand for any competent programmer.

A suitable system would use general and domain specific adaptation knowledge, and also learn products and processes (Mili, 1995). Completion rules and adaptation rules (Wilke *et al.*, 1996), are similar to the Adaptation specialists and strategies (Smyth, 1996), both of which suggest a two tier approach to the adaptation problem.

Lastly, it is expected that software reuse systems will benefit with the experience of use (Biggerstaff, 1992), but little research has been done into the evolutionary aspects of SR systems as opposed to the evolution of software systems in general.

This thesis examines a specific area of SR in the programming of FPGAs. The FPGA programs under examination are evolved using EC techniques, they are not designed by humans using conventional techniques. To understand and reuse the evolved programs CBR techniques will be applied to the problem. CBR is explained in detail in Chapter 3.

2.1.1 Field Programmable Gate Arrays and programs

This thesis examines a specific area of software reuse in the programming of Field Programmable Gate Arrays (FPGAs). An FPGA is a programmable microchip that takes as a program a representation of a digital logic circuit. The FPGA takes on the digital circuit configuration given to it as a program. FPGAs have the advantage over Gate Arrays in that they do not have to be manufactured for a specific purpose. This greatly speeds up research, development, testing and deployment of products. FPGAs can also be quickly reprogrammed to fulfil a new specification, whereas non-programmable Gate Arrays cannot (Xilinx, 1996).

FPGAs are used in a wide variety of applications from signal processing to security e.g. the Sbox, used in automated teller machines for encryption when transmitting financial information. FPGAs are also widely used for control systems such as robots and for providing a flexible interface to other hardware devices e.g. PCI cards, PCMCIA devices (Xilinx, 1997).

The FPGA programs under examination in this thesis are evolved using the EC technique of CGP (Miller *et al.*, 1999a), they are not designed by humans using conventional techniques. In order to understand and reuse the evolved programs, CBR techniques have been applied to the problem. CBR is explained in detail in Chapter 3.

An important application of this work is to produce a highly automated design method that produces more efficient digital circuits than those generated by conventional techniques, in terms of size i.e. the number of cells used on an FPGA, in design areas like signal processing circuits (digital filters). It is shown that this can be achieved by using a knowledge-lean technique from EC for designing the circuits and then CBR for understanding and reusing them. The EC techniques used to evolve the FPGA programs and the programs themselves are described in Chapter 4.

It is intended that this work will also overcome the limitations of CGP, in design areas like signal processing (digital filter circuits). The EC techniques used to evolve the FPGA programs and the programs themselves are described in Chapter 4.

2.2 Conventional logic synthesis

Logic programming can be seen as a specific kind of software programming. Gate Arrays are microchips that require a logic program to perform a function. FPGAs are a specific type of gate array that is user programmable, and can be reused by reprogramming it with a new program. Normally Gate arrays are programmed once and then discarded after use.

Conventional logic synthesis techniques have been used to partially create programs for FPGAs. It is beyond the scope of this thesis to give a complete description of Boolean algebra (Devadas *et al.*, 1994; Lala, 1996). PLA files (PLA stands for programmable logic array) commonly specify combinational logic functions. A PLA file is a truth table with additional information about the numbers of inputs, outputs and products of the target program, and uses the format shown in Table 2. A PLA file differs from a truth table in that a PLA file need not have all outputs or inputs specified.

Inputs	3
Outputs	2
Products	8
000	00
001	01
010	01
011	10
100	01
101	10
110	10
111	11

Table 2. An example PLA file for a three variable function, the 1-bit adder with carry.

In Table 2 the three inputs are A, B and Carry in. The two outputs are the sum and the carry. The eight products are the eight different sets of binary numbers that can be produced at the outputs (sum and carry) when each of the eight different sets of inputs are presented at each of the inputs (A, B and Carry in).

The main aim of logic synthesis is to represent a logic function in the simplest manner possible. There exists no complete method for synthesis of any logic function. A complete method for synthesis could synthesise any logic function using any primitive logic operations (AND, OR, NOT, EOR, NOR, NAND) and is not limited to synthesis of a specific function or set of functions, or limited to a specific set of primitive logic operations. The main reason is that the techniques suffer from exponential growth in the effort required to solve a synthesis problem as the number of inputs increases. The next most common limitation is that most methods for logic synthesis can only use a limited set of primitive logic gates to represent a given logic function leading to inefficient representations for many problems.

Canonical and Two-level Boolean functions only use AND, OR and NOT to synthesise logic functions, the ESPRESSO technique (Brayton *et al.*, 1984) only applies to two-level AND-OR representations. NAND-NAND and NOR-NOR representations allow any Boolean logic function to be constructed using either NAND or NOR gates. Methods like De Morgan's theorems can be used to convert NAND gates in expressions to OR gates, and also NOR gates to AND. These limited sets of logic gates make these methods very inefficient for some problems. In addition to this there are some functions e.g. The Achilles Heel function (Brayton *et al.*, 1984), parity functions and the n-bit multiplier that grow exponentially in difficulty with the number of input variables.

Karnaugh maps are a graphical technique used to simplify logic functions. This method can be used in conjunction with the Quine-McCluskey Algorithm, (Quine, 1952; McCluskey, 1956) but both techniques are only practical for functions with small numbers of input variables (Davio *et al.*, 1983).

Multilevel Boolean Functions enable multilevel representation of a logic function that allow factoring and decomposition into sub-functions. In general all of these classical representations are impractical, as their size is exponentially dependent on the number of inputs.

Binary decision diagrams (BDD) Lee (1959) and Akers (1978) suffer from the problem that their size is dependent on the variable ordering. There have been many heuristics devised to find a good ordering including evolutionary algorithms (Brace *et al.*, 1990; Friedman and Supowit, 1990; Fujita and Matsunaga, 1993; Drechsler *et al.*, 1996).

Many other types of decision diagrams have been proposed which can provide smaller more efficient representations of Boolean functions. Again, like the classical methods, representations like ordered Kronecker functional decision diagrams are limited to XOR and OR gates (Drechsler *et al.*, 1994a). Further to this it has been proven that certain functions have Ordered-BDDs (OBDDs) that have exponential numbers of vertices as functions of the number of input variables. The n-bit multiplier is an example of this (Bryant, 1991) and also the Devadas function (Devadas, 1993).

Depending on the design problem Exclusive-OR Logic including Reed-Muller form uses Exclusive-OR gates to implement Boolean logic efficiently where the canonical Boolean logic form does not. The worst case of this efficiency difference being the n-parity functions which can be implemented with n - 1 XOR gates only but which require $2^{n-1} - 1$ OR gates and a large number of AND gates. Evolutionary algorithms have also been used to improve the Reed-Muller representations, (Miller *et al.*, 1994; Drechsler *et al.*, 1994b; Sasao, 1993; Thomson and Miller, 1996).

It has been shown here that the conventional techniques of logic synthesis are limited by the range of functions they are applicable to; the set of primitive logic operations that they may use and by the number of input variables of the function to be synthesised. The next section shows how EC can be used for logic synthesis, and how it overcomes several of the limitations of the conventional techniques.

2.3. Evolutionary Computation for Programming Field Programmable Gate Arrays

Automated knowledge acquisition is an important issue in SR as the creation and maintenance of software libraries is expensive. It can be achieved by several methods. Experts can compile knowledge manually, automated techniques like Natural Language Processing (NLP) can extract knowledge from Natural Language documents and other data-mining techniques can produce knowledge e.g. by Filtering or mapping legacy databases to new uses. Many Artificial Intelligence techniques suffer from the 'knowledge acquisition bottleneck' – the difficulty of gaining enough knowledge to enable these technologies to work (Bramer *et al.*, 1996). Some techniques like EC are described as 'Knowledge-lean' i.e. they require little knowledge to operate.

Rule-induction systems, like ID3, suffer from scaling-up to large numbers of rules. Techniques such as the expanding window method reduce the scaling problem in ID3 but cannot be guaranteed to continuously produce good decision trees. They also suffer from the 'knowledge acquisition bottleneck', as they require carefully constructed training sets. Other approaches e.g. PRISM (Cendrowska, 1987) generates rules instead of decision trees, but still suffer from the above problems. Further problems with Induction methods are that they can be over-fitted, or applied to irrelevant attributes and noise in the data can lead to difficulties in selecting one decision against another (Bramer *et al.*, 1996).

EC can be used as an automated knowledge acquisition technique. EC has several advantages in Data Mining when compared to rule-based induction approaches. EC techniques produce a wider range of results, as they are not restricted by a search strategy, they are naturally parallel and require less user interaction. However, EC does not currently have the ability to directly exploit domain knowledge. EC can produce multiple answers for one data set and EC requires a lot of computing power (Bramer *et al.*, 1996). As EC techniques are power hungry they are limited by the available computing power. CBR has been used to enable EC to exploit domain knowledge, enhancing the performance of EC (Louis *et al.*, 1992).

Quantum theory provides an explanation for the diversity in the range of results produced by EC. Quantum theory is now accepted and it is no longer a question of correctness, but a question of why the theory is correct - John Wheeler (McEvoy *et al.*, 1996). The theory contends that everything in our universe is the result of random processes at the quantum level. The far reaching consequences of the theory can be seen as it explains e.g. the periodic table, the stability of DNA and the operation of lasers and microchips. So this diversity and sophistication emerges from randomness, creating a diverse environment. Even if quantum theory is upturned as theories often are, it is clear that the effects of random events can be traced back to the limits of what is understood about the universe today. The mutation operator explained in Chapter 3 section 3, central to the EC technique used in this thesis, is also a random process, in addition to this operator the concept of 'survival of the fittest' is also used in EC.

EC is based on biological systems and has three important components:

- A phenotype which represents a living individual organism in a biological system or a solution for a problem,
- A genotype, which is an encoding of the information (e.g. a Case in a Case-Base) which is used to produce the phenotype,
- A fitness function which is used to ascertain the quality of each individual.

EC techniques have been successfully used for knowledge-lean data mining (Maher *et al.*, 1996). For this reason EC techniques can be used to generate knowledge where the human expertise is not available. EC is therefore suitable for partially automating the reasoning process.

One method of programming a FPGA is to use EC. The conventional methods are limited automatic methods for designing digital circuits. The only fully automatic method (Quine -

McCluskey algorithm) can only use AND, OR and NOT gates as the components for the circuit, and this can lead to very inefficient designs, as for example it is inefficient to build an EOR (Exclusive OR) gate from AND, OR and NOT gates.

The symbols used to represent logic gates are given in Figure 2. The truth tables defining the function of the logic gates are given in Tables 3 and 4.

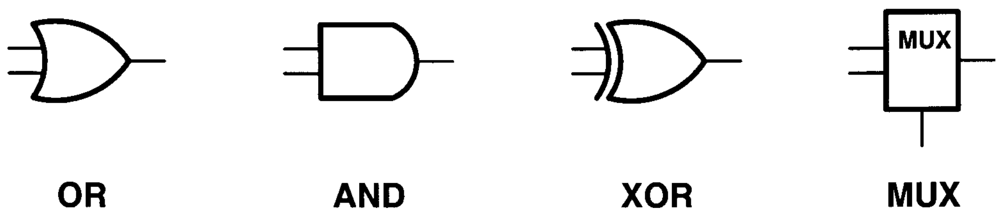


Figure 2. Binary circuit symbols used to represent logic gates in circuit diagrams. Note that the small circles that can appear on some of the inputs and outputs of these gates in figures throughout this thesis indicate inversion (logical NOT).

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

(a)

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

(c)

Table 3. Truth tables defining the logical operation of (a) OR, (b) AND, (c) XOR in Figure 2.

A	B	C	MUX e.g. A AND NOT C OR B AND C
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

Table 4. A truth table defining the logical operation of a MUX (Figure 2). The MUX effectively acts as a switch, the C input being used to select either A or B as an output.

These limitations mean that digital circuits which require gates other than AND, OR and NOT, must be designed by hand. EC is an alternative automatic method of designing digital circuits. Further to this EC is capable of producing circuits that are more efficient FPGA programs in terms of the number of two input gates used, than the best human-designed equivalent. An example of this efficiency is discussed in section 4.2.1.3.

It is important to note that a CBR-EC hybrid could be capable of producing many different kinds of programs, not just FPGA programs, this versatility is discussed in section 3.2 of this chapter.

What has become apparent in the field of EC being used for design purposes is that design principles, not previously known by human digital circuit designers, are being evolved by EC. Also, design principles that were previously known to human digital circuit designers have been evolved. An example is the 2-bit ripple-carry adder (Figure 3), where two 1-bit carry adder circuits are ripple-chained together. The ripple-chain is so named as many 1 bit carry adders can be connected together in a chain to produce an n-bit carry adder, where n is also the number of carry adder units required. The carry from the first 1 bit adder 'ripples' to the next. This process continues until the last 1bit adder unit in the chain. This ripple-chain principle of connecting two or more identical simple circuits together to solve a large problem has also been observed in the evolved design for a 2-bit cellular multiplier.

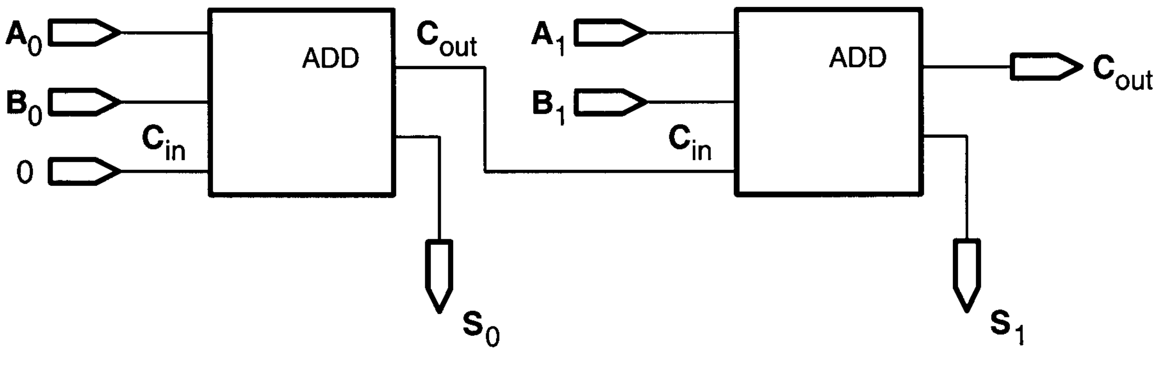


Figure 3. 2-bit ripple-carry adder.

The first problem with evolving digital circuits is the problem of scale. The examples used in this thesis (see Appendix 2 for example PLA files) involve a small number of input and output bits as larger numbers of bits require much larger circuits, and this gives an exponential growth in the number of computations required to evolve a correct circuit. Four sizes of carry-adder, from 1-bit to 4-bit, were designed to show the effect that increasing the number of input variables has on computational cost, for the EC method discussed in detail in Chapter 4. Table 5 gives the average number of generations required to evolve 100% functionally correct Carry-Adder units for Carry Adders of increasing scale. As the scale increases the number of computations required for one generation increases as the size of the program also increases. The 4-bit Carry-Adder takes approximately 10 times longer than a 3-bit Carry-Adder for each generation. So it is necessary to extract the evolutionary principals from collections of the best FPGA program solutions produced by a EC, to enable the CBR design of much larger circuits using much larger numbers inputs and outputs. The complexity of the problem also effects the size of the required circuit. The Sbox problem (Chapter 1 section 3) has a relatively small number of input and output bits (PLA file) but the circuit required to solve the problem appears to be many times the size of a multiplier circuit of a similar sized PLA file (see Appendix 2).

Carry-Adder Scale	**Average number of generations**
1-bit	**< 50 (sample 500)**
2-bit	**321,393 (sample 500)**
3-bit	**1,953,056 (sample 500)**
4-bit	**5,153,533 (small sample)**

Table 5. An Example of the Scaling Problem in Carry-Adders. As the scale increases the average number of generations required to achieve a 100% solution increases.

In Chapter 2 section 3.2 it is shown that an evolutionary algorithm provides a mechanism for a process referred to in this thesis as assemble-and-test, and is an engine for discovering new designs. These designs are often radically different from those produced by top-down, human, rule-based approaches. Here these ideas are tested in the context of designing digital circuits, particularly arithmetic circuits. It is shown in Chapters 6 and 7 that, by studying evolved designs of gradually increasing scale it is possible to discern new, efficient, and generalisable principles of design.

The concept of the evolutionary algorithm to gradually improve the quality of a design has been adopted in the field of Evolvable Hardware (Sipper *et al.*, 1997) where the task is to build an electronic circuit. Here the circuits are encoded in genotypes which can be simply translated into circuits or phenotypes. They are then tested in a computer simulation or in physical hardware. Note: the biological terminology used in the field of evolutionary computation serves as useful terminology and as a reminder of the ideas that it is based upon. Evolvable hardware research can be divided into two main categories: intrinsic evolution and extrinsic evolution. Intrinsic evolution refers to an evolutionary process in which each phenotype is built in electronic hardware and tested. Extrinsic evolution simulates a hardware model in software.

Each of these categories can be further sub-divided into analogue, digital or mixed analogue-digital domains. Intrinsic evolution in the analogue domain has recently become possible because of the availability of reconfigurable analogue devices (Motorola, 1997; Grundy, 1994). Researchers have begun to explore the possibilities for automatic design that reconfigurable analogue devices can facilitate (Murakawa *et al.*, 1998; Flockton and Sheehan, 1998; Zebulum *et al.*, 1998; Stoica *et al.*, 1998; Zebulum *et al.*, 1999; Stoica *et al.*, 1999; Flockton and Sheehan, 1999).

Thompson (1997) used a reconfigurable digital platform, the Xilinx 6216 FPGA. In his research Thompson produced a timer circuit that used fewer components than conventional design mechanisms stated as the minimum. Thompson discovered that this evolved design exploited the physical properties of the specific individual Xilinx 6216 FPGA that it had evolved on. Further to this Thompson showed that this design was not portable to every other Xilinx 6216, only to some, due to uncontrollable and minute physical differences between 6216s of 'identical' human design. And so it was seen that the evolutionary algorithm could produce circuit designs further outside the conventional design space than previously expected, that are more efficient than those produced by conventional techniques (Thompson *et al.*, 1999).

Koza (1994) has pioneered the extrinsic evolution of analogue electronic circuits. By using evolutionary algorithms (genetic programming, specifically) combined with the SPICE simulator Koza has automatically generated circuits which are competitive with those of human designers. Systems like the SPICE simulator require expert training to use, otherwise the simulator produces results that do not closely represent the behaviour of the a real physical circuit (Zebulum *et al.*, 1998).

Thompson (1997) and Kajitani et al. (1998) have pioneered intrinsic evolution for purely digital systems. However most researchers are content with extrinsic evolution (Miller *et al.*, 1997; Iba, 1997). An advantage of extrinsic digital evolution is that non platform-specific representations of circuits can be created which are then portable to many platforms. Extrinsic digital evolution overcomes the problem of chip-specific designs encountered by Thompson (1997). Extrinsic evolution also permits relaxing and constraining of the simulator parameters, which facilitates the study of the effect of specific parameters on the evolutionary process.

This thesis is only concerned with the extrinsic evolution of digital combinational (non-sequential) circuits. The thesis specifically examines arithmetic digital functions.

Arithmetic functions were chosen (addition and multiplication) for several reasons:
1) They are modular in conventional construction.
2) There are well-established conventional methods of building them.
3) They are fundamental building blocks of many digital devices, and as digital filters.

Even-parity functions have been studied in addition to arithmetic functions and together are sufficient to explore the efficiency of the techniques and the novelty of new designs.

Even-parity functions were chosen for two reasons:

1) It is well established that these functions are difficult to find by random search when the operators are constrained to the following set: {AND, NAND, OR, NOR} (Koza, 1992).

2) They have been used extensively to test the effectiveness of various algorithms (Poli, 1999).

Together they afford a study of The Fundamental Question (TFQ):

"By evolving a series of sub-systems of increasing size, is it possible to extract the general principle and hence discover new principles?"

It is argued that the general principle of scalable design can be automatically identified in the case of designing arithmetic circuits.

An example of such discoverable and scalable principles can seen in the way that the principle of the ripple-carry adder follows as a consequence of examining the best evolved designs for the one and two-bit adders with carry.

2.3.1. Applications of Evolutionary Computation

The application of EC to FPGA programming is examined in detail in Chapter 4. A full review of EC is beyond the scope of this thesis. EC techniques have been used for design and optimisation in fields including airfoil design, scheduling, nuclear reactor reload design, retail dealership relocation and oil production scheduling (Schoenauer, 1998). In this thesis a specific field of Genetic Programming (GP) (a branch of EC), Cartesian GP, is examined.

2.3.2. Exploring the Space of All Representations with EC techniques

This section shows how the use of an evolutionary algorithm implementing assemble-and-test could be used to explore over a much larger area of design space than that possible using a top-down rule-based design algorithm. Figure 4 shows a particular case of this for the problem of finding efficient representations of Boolean functions and it illustrates one of the fundamental concepts of this thesis.

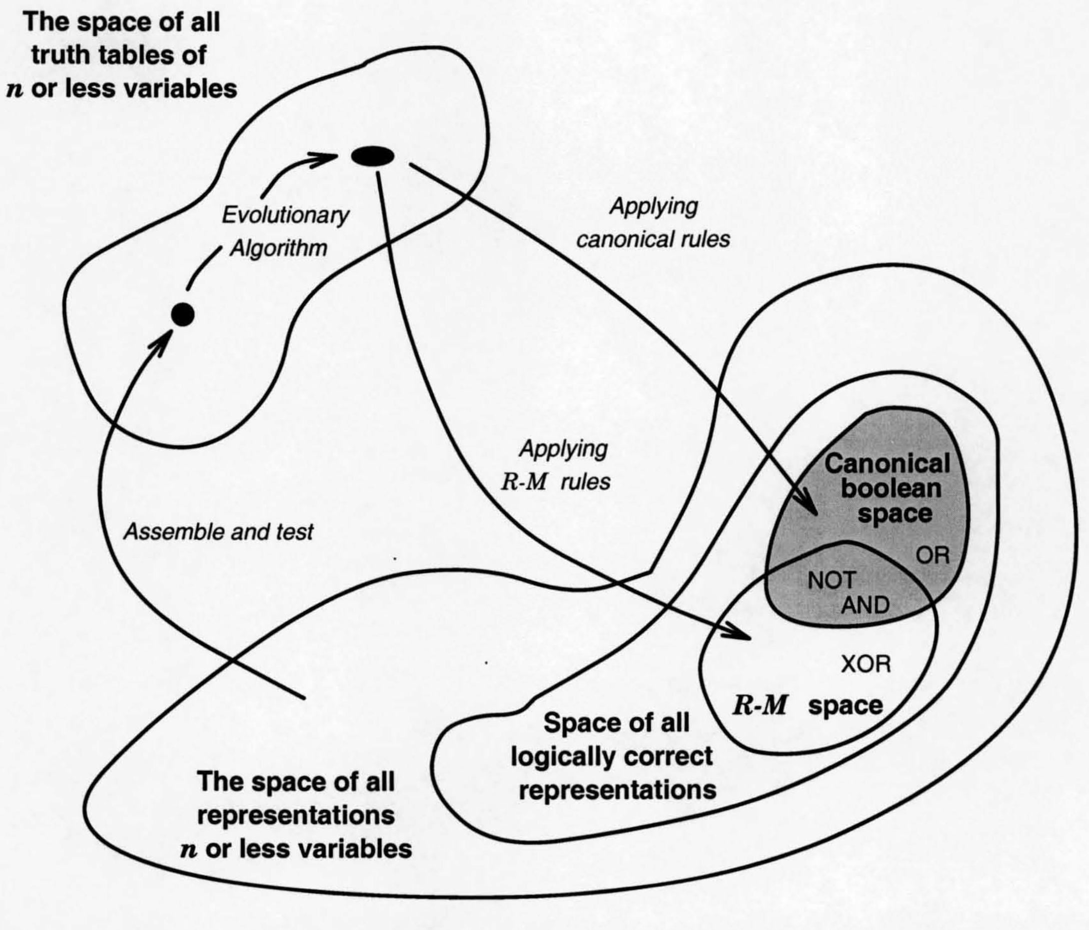

The space of all truth tables of n or less variables
Evolutionary Algorithm
Applying canonical rules
Applying R-M rules
Assemble and test
Canonical boolean space
OR
NOT
AND
XOR
R-M space
Space of all logically correct representations
The space of all representations n or less variables

Figure 4. How " assemble-and-test " reaches the space of all representations. Canonical boolean space only covers logically correct representations with NOT, AND, OR as can Reed-Muller, but with XOR instead of OR

The analysis in Figure 4 was developed in Miller *et al.*, (2000a) to show how " assemble-and-test " reaches the unknown regions of the space of all representations. Conventional logic design begins with a precise specification in the form of a truth table, PLA file, binary decision diagram, symbolic expression etc. The expression is manipulated by applying canonical Boolean rules (AND, NOT, OR) or Reed-Muller algebraic rules (AND

NOT XOR). It is not possible to escape from the space of logically correct representations. The methods though powerful in that they can handle large numbers of input variables are not adaptable to new logical building blocks and require a great deal of analytical work to produce small optimisations in the representation. Assembling a function from a number of component parts begins in the space of all representations and maps it into the space of all designs. The evolutionary algorithm then gradually pulls the specification of the circuit towards the target truth table (shown as a small dark ellipse). Thus the algorithm works in a much larger space of functions many of which do not represent the desired function. It is one of the contentions of this thesis that this is the only way to discover radically new designs.

It has been shown that by the process of assemble-and-test that EC can automatically produce novel and efficient designs. The use of assemble-and-test for producing novel designs is now commonplace in the practice of Evolvable Hardware (Sipper *et al.*, 1997; Thompson *et al.*, 1999). As these evolved designs can come from a much larger design space than conventional rule-based methods can cover, the diversity of the resultant design solutions is greater and they are therefore much more difficult to interpret.

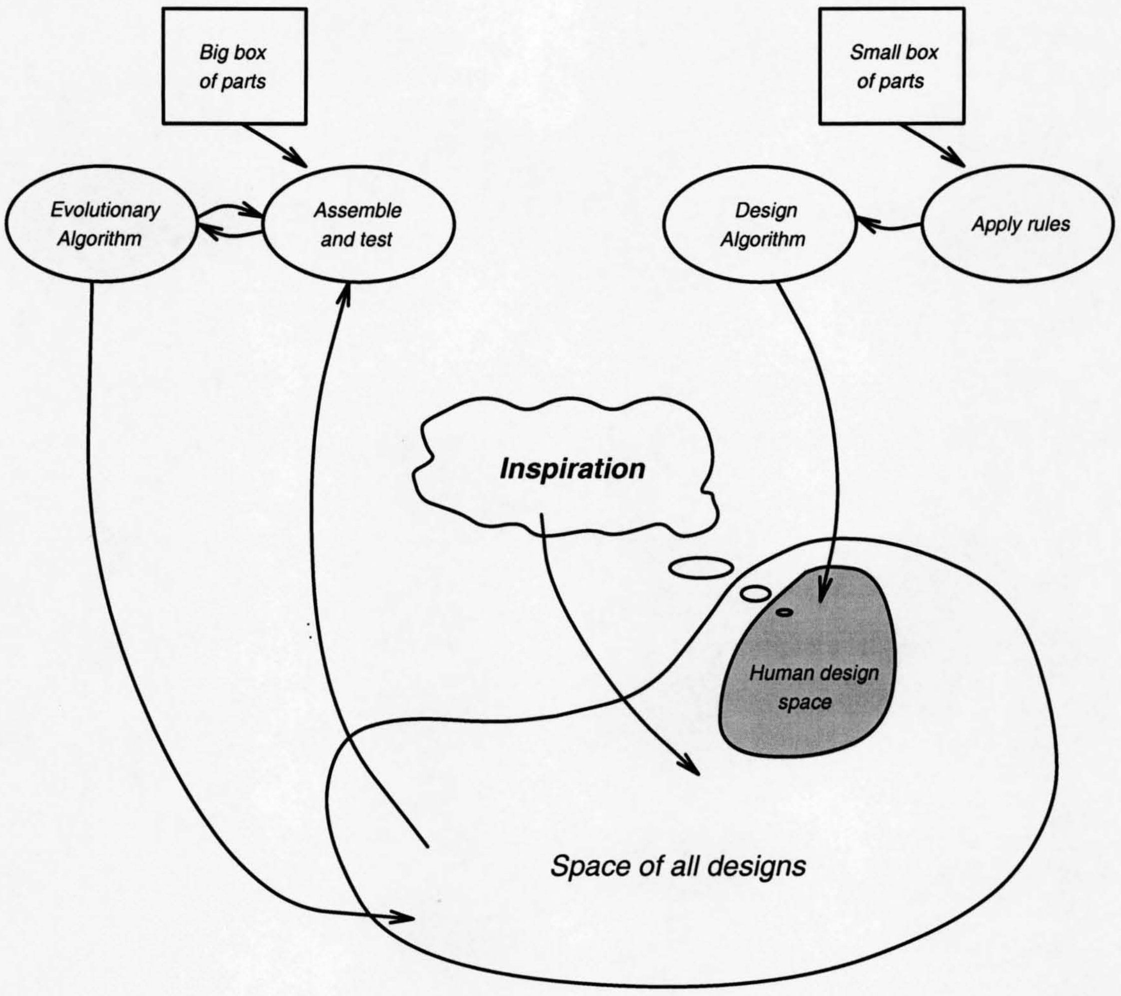


Figure 5. Conventional design versus evolutionary design with assemble-and-test.

It is argued that traditional design techniques take a top-down approach beginning with a precise specification and through the application of complex rules and principles the design is implemented. The top-down design approach is very different from the mechanisms that produced the stars, elements and life on earth. In nature an extraordinary diversity and sophistication in living creatures can be seen. There is evidently a natural mechanism that produces these complex designs.

For complex organisms, e.g. for humans, a process of natural selection comes into play. Natural selection provides the environment that allows such organisms to evolve.

Evolutionary computation is based on our understanding of this process. In evolutionary computation the design starts as a set of instructions encoded in a manner based on what is known of real DNA. In nature, DNA is translated into the assembly of building blocks upon which life forms are based. In EC DNA or chromosomes, are translated into the phenotype by applying a model of the problem. Whether or not the organism remains in existence and evolves further is dependent on the environment it lives in. As organisms evolved so larger numbers of building blocks were assembled together into more complex organisms. These organisms continue to exist if they can survive in their environment. The environment effectively tests the organism design. EC uses a fitness function to directly test a chromosome or organism. The chromosomes that are poor in fitness are discarded from the evolutionary process in favour of those chromosomes of higher fitness.

This evolution in an environment process is referred to as assemble-and-test. The idea of assemble-and-test fed back into itself produces Evolutionary Computation. Figure 5 illustrates this concept in the general space of designs. The top-down rule-based space of designs is shown in grey as a small sub-region in the much larger space of all possible designs. When humans discover a new design method this space can be widened to cover more of the general space. Top-down rule-based design will always give solutions in a sub-region of the general design space as this approach is inherently blind to alternative solutions e.g. they will usually give the same answer for any given data set regardless of whether or not there are other potentially better solutions.

The fact that conventional human design methods follow specific systems of rules and principles also limits the choice of building blocks that can be used in any of the conventional techniques. As these systems of rules and principles do not exist in natural evolution, evolutionary computation techniques can use any of the building blocks available.

2.4 Summary

Software reuse aims to minimise the effort required to create a new solution by reusing existing knowledge. FPGAs are programmable microchips that require programs that represent digital logic circuits. Conventional techniques for creating FPGA programs are limited in the size of the programs that they can create and by the logical components and rules that they can use to synthesise an FPGA program. EC is an alternative method of synthesising an FPGA program that is not restricted in the logical components it can use or by any system of rules.

EC is capable of synthesising FPGA program designs that cannot be synthesised by any other method. EC can explore the space of all designs, which includes the design spaces of the conventional techniques.

By combining EC and CBR it may be possible to create larger FPGA programs than those that can be synthesised with EC alone. Further to this, it is proposed that it may be possible to discern new, efficient and generalisable principles of design by studying evolved designs of gradually increasing scale for specific problem classes e.g. arithmetic multiplication.

3. CBR for Software reuse

Software Reuse (SR) is a domain where knowledge is represented by software artefacts and expert experience. These software artefacts make a good basis on which to make Cases for a Case-Base. Expert experience can be captured by a CBR-SR system as the system is used, thereby learning from use. Software reuse requires methods to find, retrieve, adapt and retain existing artefacts to solve new problems (Krueger, 1992; Mili, 1995; Biggerstaff, 1992). CBR provides methods to find, retrieve, adapt and retain Cases.

Additionally CBR is suited to domains where the domain theory is weak i.e. there are usually no explicit or best method(s) of solving any given software problem and large amounts of domain experiences exist e.g. software artefacts (Krueger, 1992; Mili, 1995; Biggerstaff, 1992).

CBR cannot however solve a problem if it does not have any suitable Cases in its Case-Base. Evolutionary Computation is a knowledge-lean problem solving technique that can be used to evolve solutions with little knowledge about solving any given problem.

This Chapter discusses how CBR can solve problems in Software Reuse. Next, an example of the CBR methodology is given. The example is followed by a discussion of hybrid CBR-EC techniques to show how EC can be used to solve problems in CBR and vice versa.

A review of current CBR literature has shown that CBR does not appear to have been applied to a domain with the complexities of FPGA Software Reuse with automated adaptation. CBR provides many of the facilities required by SR, and CBR mechanisms themselves are similar to those used in SR.

The following points illustrate the application of CBR to SR:

- CBR is good in a domain where there is plenty of experience representing the domain even if the domain theory is weak.

- CBR is a technique that can retrieve, specialise and learn with use.

Gibbs *et al.* (1990) compares reuse of past experience and continuously evolving effort in the legal field to the requirements of software development and maintenance. Case-Based Reasoning is an AI technique that uses past experience and captures new experience and so continuously learns as it is used. This comparison is made as two of CBR's primary components are retrieval of past knowledge and specialisation of this knowledge to solve a new problem. This means that CBR can provide a tool that can solve two of the major problems in software reuse, finding reusable artefacts and specialising those artefacts.

• CBR facilitates knowledge acquisition

Another problem in software reuse is that of knowledge acquisition. In CBR knowledge is stored as Cases, in a form a human would relate the knowledge to another human. This aids the problem of user understanding of the CBR system. The CBR system is an evolutionary one, facilitating knowledge acquisition as the system improves through use. This also facilitates maintenance and debugging. Knowledge acquisition is also superior in CBR over other AI techniques as it is easier to produce Cases in experience-rich domains that lack theories, than it is to formalise rules, for example Gibbs *et al.* (1990).

• CBR supports specification, maintenance and debugging

CBR has also been used extensively to support the descriptions of new problems, and can therefore be used to support the production of software requirements that allow the reuse of software artefacts (Maguire *et al.*, 1995). CBR has been used successfully in the retrieval of software components through the system CAROL, a Case Assisted Object Library reuse system (Maguire *et al.*, 1995). Many Help-desk applications have also successfully used CBR, which demonstrates CBR's applicability to the debugging problem.

• CBR is scaleable through and across multiple domains and improves with use

The problem of scale and domain coverage can also be greatly alleviated by using CBR as many different types of knowledge have been represented and used in CBR (Gibbs *et al.*, 1990). For example, MEDIC, a lung disease diagnosis system, CHEF, (Kolodner, 1993) a recipe creator, CLAVIER system used to generate autoclave layouts, and the legal domain where new Cases are developed from previous ones (Giraud-Carrier, 1996). Much work has been put into using large numbers of Cases, indexing and retrieval methods are simple to implement in a parallel or distributed fashion. It has also been shown that much of the CBR processing can be moved away from the user-time and Cases can be pre-processed to prepare them for use beforehand (Leake, 1996).

- CBR can use previous experience to solve new problems

Hierarchical CBR, which supports representations of Cases at various levels of abstraction, has shown success in the combination of Cases to provide a solution (Smyth, 1996), and so the integration and adaptation problem in software reuse can also be tackled.

- Problems that CBR cannot solve in SR

CBR is limited by the knowledge in the case-base. If the case-base contains no knowledge pertaining to a problem, it cannot solve that problem. This problem is where the facilities of EC can be applied. EC techniques can produce a solution to a problem without specific knowledge of the domain that it is searching for a solution. EC techniques achieve this by guessing solutions, testing them to see how good they are and then recombining these solutions and re-testing until a satisfactory solution is found.

3.1. CBR

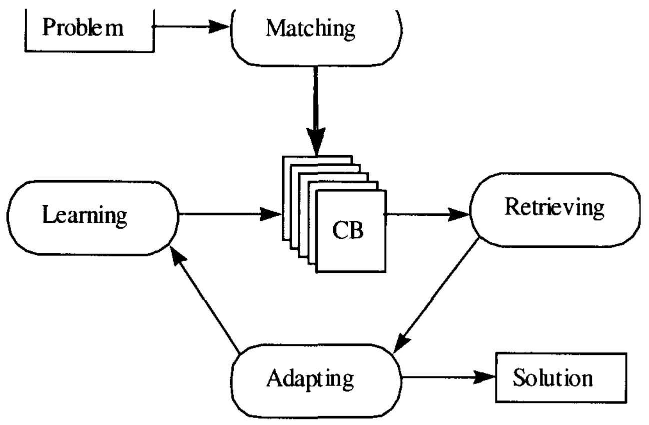


Figure 6. The basic CBR process.

The basic Case-Based Reasoning process shown in Figure 6 begins with a Problem description. The user describes their new problem, which could be a free text description or a complex design incorporating textual and diagrammatic specifications or partial/complete filling in of a template/form. For example, cases can be simple *attribute-value* pairs such as:

Attribute	Value
Food	Turkey
Weight	4kg
Cooking time	????
Temperature	????

A problem Case would have one or more attribute(s) without value(s), such as temperature or cooking time.

In CBR terminology, a Case usually denotes a problem situation. A Case is a previously experienced situation, which has been characterised and stored in a way that it can be reused in the solving of future problems. An unsolved Case is the description of a new problem to be solved. Case-based reasoning is a cyclic and integrated process of solving a problem, learning from this experience, solving a new problem, and so on (Barletta, 1991; Kolodner, 1996; Richter, 1998).

Problem solving is not necessarily the finding of a concrete solution to an application problem, it may be any problem put forth by the user. For example, to justify or criticise a solution proposed by the user, to interpret a problem situation, to generate a set of possible solutions, or generate expectations in observable data are also problem solving situations (Barletta, 1991; Kolodner, 1996; Richter, 1998).

Next, **Matching** (Figure 6), is the process of comparison of the problem Case with existing Cases in the Case Base. This can be simple, word counting, numerical value matching, or complex matching based on e.g. OO structure. Techniques like Hierarchical Case Base Reasoning (HCBR) (Smyth, 1996) are designed to facilitate indexing and retrieval by organising the Cases into a hierarchy where specific Cases are indexed under more general Cases. This means a general match can quickly be found and then further specialised Cases are found under the general match.

Any Case-Based Reasoning system is dependent on the structure and content of its collection of Cases, known as its Case-Base. As CBR problem solving is achieved by retrieving previous Cases, the search and matching processes must be accurate and time effective. As new solutions are generated they must be retained to support the learning process, and so the integration of a new Case into the Case-Base must also be effective and time efficient (Barletta, 1991; Kolodner, 1996; Richter, 1998).

It is clear that the primary problems in CBR are: the problem of deciding what to put in a Case; finding an appropriate structure for describing Case contents; and deciding how the Case memory should be indexed for effective retrieval and reuse. An additional possibility involves the problem of integrating general domain knowledge into the Case-Base (Barletta, 1991; Kolodner, 1996; Richter, 1998).

The next stage of CBR (Figure 6), **Retrieving**, involves selecting the Cases that you have matched to the problem Case. Two example Retrieval methods are Standard retrieval that simply selects the closest match and Adaptation Guided Retrieval (AGR) (Smyth, 1996). AGR finds existing Cases in the Case-Base that can be best adapted to solve the problem, rather than simply retrieving the closest Case to the problem Case.

A new problem is solved by finding a similar past Case, and reusing it in the new problem situation. This can involve **Adapting** (Figure 6), the retrieved Case(s) using knowledge of the domain and knowledge from other Cases. Adaptation can be for example: Substitution, which in turn can be Simple, where the substituted component is independent of Case under adaptation or Complex, where the substituted component is dependant on context and Case under adaptation; Model guided, where a structured system is used to adapt; or adaptation can be performed using domain knowledge that could be for example, Rules or Cases (Barletta, 1991; Kolodner, 1996; Richter, 1998).

The last stage in the basic CBR model is **Learning** (Figure 6). Learning involves storing the new problem-solution pair (Case) in the Case-Base. Two major advantages of CBR are that a new Case can be added without concern for existing Cases and that CBR systems can evolve with use, in the workplace.

The **Solution** (Figure 6), may be a single specific solution with explanation, if necessary an interpretation of a problem situation or a set of possible solutions, or a list of expectations in some observable data. The Solution is only stored if the new Case has new information that does not already exist in the Case-Base.

Maintenance of the Case Base is necessary to remove old or unusable material, e.g. the removal of duplicate Cases, or the removal of errors or out of date Cases. Maintenance also involves the addition of new Cases, perhaps from another CBR system and the rebuilding of indexes or storage structures e.g. examining the history of the systems use to spot problem areas, such as bottlenecks and areas with insufficient Cases (Barletta, 1991; Kolodner, 1996; Richter, 1998).

There are two main types of CBR systems, problem solving and problem interpretation. Problem solving is used in design and planning where a solution is derived from retrieving approximately matching Cases and adapting them to the new problem. Problem interpretation is used, for example, in legal and diagnosis fields where new problems or situations are explained and illuminated using closely matching Cases of previous experiences (Barletta, 1991; Kolodner, 1996; Richter, 1998).

An example of problem solving with Cases in a Case-Base for solutions to cooking problems follows. This example is adapted from an example of Case-Based Reasoning rule extraction given by Hanney (1996).

The food cooking problem.

Attribute	Value
Case Id	A
Food	Turkey
Weight	2
Cooking time	1
Temperature	200
Form	Whole

Attribute	Value
Case Id	B
Food	Turkey
Weight	3
Cooking time	2
Temperature	200
Form	Whole

Attribute	Value
Case Id	C
Food	Salmon
Weight	3
Cooking time	2
Temperature	250
Form	Steaks

Attribute	Value
Case Id	D
Food	Salmon
Weight	3
Cooking time	2
Temperature	300
Form	Whole

Rule generation could be implemented as follows:

- First find Cases that are almost identical:
  - A and B, C and D
- Comparison of Case A with Case B gives rule R1:
  - If Weight changes from 3 to 2
  - Then reduce Cooking time by 1
- Comparison of Case C with Case D gives rule R2:
  - If Form changes from Whole to Steaks
  - Then reduce Temperature by 50

Given a new food to evaluate Cooking time and Temperature for:

Attribute	Value
Case Id	X
Food	Salmon
Weight	2
Cooking time	?
Temperature	?
Form	Steaks

Matching: Match this problem Case X to existing Cases and rank them in order (Nearest Neighbour matching).

Case A	1 – A match with Attribute Weight = 2
Case B	0 – No match
Case C	2 – A match with Attribute Food = Salmon, Form = Steaks
Case D	1 – A match with Attribute Food = Salmon

Retrieval: Case C is the closest match, this gives an estimated cooking time of 2 and an estimated temperature of 250.

Adaptation: A rule exists that can cope with a weight change from 3 to 2, R1:

If weight changes from 3 to 2 Then reduce Cooking time by 1

As Case X has a weight of 2 and Case C has a weight of 3 apply R1 to adapt Case C's cooking time of 2 to a cooking time of 1. This value (1) could be derived from a more complex formula, for example a formula that is scaled or proportional to the weight change could mean that we half the Cooking time as the Cooking time has halved in the matching cases, giving a Cooking time of 1. For simplicity more complex methods are not discussed further in this example.

Domain knowledge could be added in the form of rules e.g. If it is known that Beef generally cooks faster than Turkey, then add the rule R3 to the system:

If Food changes from Beef to Turkey Then increase cooking time by 1

Learning: Now there is a new Case, Case X that could be added to the Case-Base. However this Case contains no new information that was not already in the Case-Base so it would not be an advantage to add it to the Case-Base.

Before this particular Case is discarded it is necessary to confirm that the estimated cooking time and temperature are good estimates. For example, if this food is cooked for the estimated time of 1, but during cooking it required an extra time of 0.5, then Case X is updated to reflect this. In this event Case X would be added to the Case Base as it now contains new information

Then a new rule could be generated, R4:

If Food changes from Turkey to Salmon Then increase cooking time by 0.5

CBR can also make use of Cases of failed examples. An unsuccessful cooking experience could be used to avoid repeating the same mistake.

These rules are not guaranteed to be applicable in every situation. These rules are based on the differences between two very similar Cases, not on every Case that may contain information on a particular rule. These rules rely on their context to an extent, therefore it is important that they are derived from Cases that closely match the problem Case, to increase the likelihood that the rule will be applicable. The idea of principle identification and reuse extends the idea of generating rules from Cases, explored in Chapter 5.

CBR differs from other Artificial Intelligence-Machine Learning technologies in several ways. It does not rely solely on general knowledge of a problem domain, or making associations along generalised relationships between problem descriptors and conclusions. CBR is able to utilise the specific knowledge of previously experienced, concrete problem situations (Cases). And lastly, a new problem is solved by finding a similar past Case, and reusing it in the new problem situation (Barletta, 1991; Kolodner, 1996; Richter, 1998).

Some of the organisations using CBR include: the US government for automated text analysis; IBM for customer support e.g. Help Desks; VISA International for quality assurance; British Telecom for design and fault diagnosis; British Airways for aircraft design & maintenance; NASA for process planning e.g. Autoclave layout plans for high performance aircraft parts, and for decision support. Many Companies provide or use CBR software, services and commercially available CBR tools. These companies include: AknoSoft; Cognitive Systems Inc; Inference Corporation; Tecinno; Daimler-Benz; BMW.

A review of CBR-SR literature shows that there is a close relationship between the applications that CBR has been used for and the Software Reuse problem. The following points cover most of the significant areas pertinent to the EC - CBR software reuse problem:

- CBR can be used as an adaptive reuse system in poorly understood domains, see CLAVIER in Kolodner (1993).

- A successful software reuse system must be able to evolve and learn from use. CBR evolves and learns from use (Mili, 1995).

- CBR is suitable for performing adaptation in complex systems with high variance in problems e.g. software reuse (Krueger, 1992).

• Cases are easier to create than e.g. rule bases, and provide better justification than rules, alleviating the Knowledge acquisition problem.

• Knowledge of failed solution attempts can be instructive. Failures can be learned to enable the CBR system to avoid them in the future (Kolodner, 1993).

• Knowledge should be represented at several levels in CBR and SR (Mili, 1995). Several researchers have achieved results by splitting knowledge into two main types: *Domain specific* adaptation knowledge e.g. Specialists (Smyth, 1996), specialisation (Bergmann *et al.*, 1996), rules (Leake, 1993) and *General* adaptation knowledge e.g. Strategies (Smyth, 1996; Leake, 1993), Generalisation (Bergmann *et al.*, 1996). Further knowledge that may be included is knowledge of software engineering techniques, an extension of Strategies (Smyth, 1996).

An important approach to CBR adaptation that appears to be arising from present research is to reduce the amount of adaptation that is done. This is achieved through approaches like Adaptation Guided Retrieval (Smyth, 1996). These approaches reduce the number of steps taken to adapt a Case. This means that fewer possibilities for introducing errors or failures occurring, increasing the probability of a successful solution being produced.

Learning is the best way to successfully deal with adaptation as it enhances the quality and flexibility of the process (Fuchs *et al.*, 1996). This is particularly true for more complex domains where it is difficult to assess the scope of the knowledge coverage, and therefore learning is essential as it facilitates knowledge acquisition.

3.2. Adaptation and CBR

The recent work on CBR adaptation, relevant to this project, has been carried out by Hanney (1996), Smyth (1996) and Giraud-Carrier (1996). Hanney (1996) presents two algorithms, the first for automatically learning adaptation rules from Cases and the second for automatically applying these adaptation rules to new problems. Giraud-Carrier (1996) and Hanney (1996) have both achieved automated rule extraction and adaptation in simple domains. To date the algorithms in Hanney (1996) have not been applied to a domain as complex as software reuse proposed in this project. Smyth (1996) has achieved adaptation in structured software, but has not automated adaptation rule extraction and application. This is an area of focus of this Ph.D. Hanney (1996) has applied the above mentioned algorithms to Cases with numerical atomic solutions and achieved good results. This thesis shows how these ideas can work with complex structured solutions.

An example based on Case-Based rule extraction methods devised in Hanney (1996) was given in Chapter 3 section 1.

Using ideas similar to the algorithms given by Hanney (1996), it may be possible to find principles of FPGA program design that can be extracted from sets of example adaptations from the field of FPGA programming. It is shown in Chapters 5, 6 and 7 that the scaling problem might be overcome using the combinations of EC and CBR techniques. The aim here is to select the most generally applicable of the extracted principles and to produce a mechanism for handling the complexity of the task. The primary method for handling the complexity seen in current research is to use multiple levels of abstraction over the knowledge being used, and to decompose and recompose the problem Cases (Louis, 1993; Smyth 1996). This involves creating and using abstract representations for the code artefacts themselves as well as using general and specific adaptation principles.

It has been shown that where possible, domain knowledge should be used to create initial rules and Cases, where this knowledge is simple to encode as Cases (Hanney, 1996). This is partly the case in FPGAs and digital circuit design, as domain knowledge is weak and it has been shown that existing rules and methods are not as useful as EC. Therefore it is of interest to determine the mechanisms by which EC produces solutions.

This thesis investigates the following questions:

- How much human intervention is necessary, what advantages does this technology have?
- How Scaleable is the approach?
- How portable (to other domains) is the approach?
- Do EC techniques improve the performance or technology?

CBR-SR involves all types of classification knowledge, target elaboration knowledge, role substitution knowledge, sub-goaling knowledge and goal interaction knowledge (Hanney, 1996).

Measuring how the size of the CB affects rule production is an important exercise to determine if rules can be generated, and then to reduce the case-base. If this can be done then this could facilitate adaptation in domains where only a small number of Cases are available, if these rules are domain independent enough to be applicable to the new domain (Hanney, 1996).

Hanney (1996) aims to reduce the CB size and recalculating the rules from the reduced CB. If the principles are portable to many domains this could drastically reduce the knowledge elicitation bottleneck for implementing new domains. A number of Cases versus number of rules experiment can be done to determine how many rules are general to software reuse and how many rules can be reused. If there are a small number of rules for a large number of cases then the knowledge acquisition bottleneck is reduced for new domains with few Cases.

Hanney (1996) shows the correlation between the number of Cases and the error rate, but recalculates the rule base for the new reduced CB each time, and does not measure the effect of keeping the same rule base, which is important for re-scaling to a large scale system. As a point of efficiency a Case may be removed from the CB if it provides no additional information. Once rules have been extracted it may be useful to examine which Cases contribute to the CB and rule base to refine the domain coverage.

It has been shown that adaptation seems to have a far stronger relationship to Case Base size than retrieval does (Hanney, 1996). This finding may be domain specific.

Another possible exercise is to determine what rules can be created from exact matching features in highly similar Cases, e.g. where the independent rules have been extracted, what dependency information is there? This is already handled partly by the fact that Cases are matched in the first place i.e. their context is similar. Context can be imposed by adding to a rule that it can only be applied if the subject Case for adaptation matches the context (the bits that made the original Cases that the rules were extracted from similar or matched).

Hanney (1996) says that in the property domain the rule set can be reduced without significant drop in accuracy, because combinations of rules can be used to solve target problems where no single rule exists. Hanney (1996) uses two different methods of selecting Cases from which to generate rules and concludes that generalisation does not improve adaptation accuracy in the test domains.

The experimental approach of Hanney (1996) is as follows:

- Rule generalisation (not abstraction)
- Different methods of selecting Case pairs for comparison
- Rules with short antecedents versus long antecedents
- Case based size versus prediction accuracy
- Methods of adaptation rule learning
- Guidelines for the method of adaptation rule learning

In this thesis the experimental approach examines principle identification, a more general version of rule extraction, and methods of comparing case pairs.

EC techniques often produce solutions that do not have obvious components. As they do not have a mechanism for exploiting domain knowledge, they require additional mechanisms to produce an automated reasoning method. One technology that uses domain knowledge is Case-Based Reasoning (CBR). CBR has the advantage that it can use domain knowledge in the form of problem-solution pairs, called Cases; no additional domain knowledge e.g. rules, models are required, so EC results can be fed directly into a CBR system without user interaction. This simple knowledge capture mechanism produces a highly automated process.

CBR is a problem solving method that reuses old solutions to solve new problems. CBR is an alternative to rule-based and model-based reasoning as rules or models do not need to be explicitly defined. CBR has several advantages: it can provide answers to problems in poorly understood complex domains; it does not require a domain model, domain rules, or general domain knowledge; and it can provide an explanation of its reasoning.

The main advantage CBR has over other techniques is that its knowledge is represented by Cases that represent specific experiences of experts working in any given domain. This means that expert knowledge can be entered almost directly into the Case-Base, no rules or models are required. Further to this new Cases can be added to the Case-Base simply, without the need for updating of the existing data in the Case-Base. So knowledge acquisition, updating and maintenance are simpler than other techniques.

3.3. CBR and EC Hybrids

CBR-EC hybrids are systems that combine CBR and EC methods to solve problems, either technology being used to support the other. Existing CBR-EC hybrid systems are reviewed to examine the ways in which these technologies have been combined and applied. The most significant work in this area is that of knowledge-lean techniques. This enables the generation of knowledge where little or no knowledge exists for the target domain.

This section examines systems that combine CBR and EC to solve problems in each technology. Some of the following techniques can be applied to software reuse by increasing the capabilities of CBR technology using EC.

There are several possible combinations of CBR and EC. Firstly CBR can be used to initialise the EC population before the execution of the EC (Figure 7). In this procedure CBR is used to select potentially useful data to create a starting population for EC instead of the EC using the conventional technique of random initialisation of the starting population (Maher *et al.*, 1996, Tanaka *et al.*, 1994). CBR selects a cases from its case-base that provide the best solutions to the EC problem by evaluation the EC fitness function (problem to be solved) and these cases become the starting population for the EC.

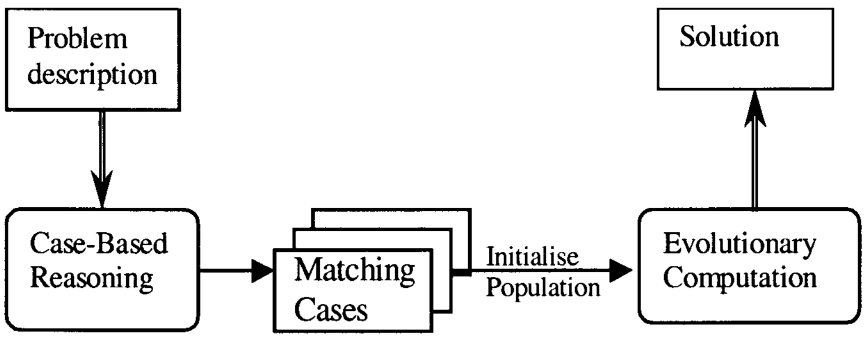


Figure 7. Use of CBR for initialising an EC population based on a problem description.

Louis (1993) combines CBR with EC for digital circuit design. In Louis (1993) CBR is used to monitor the EC during the EC process, and extract and re-inject potentially useful genetic information (Figure 8). This method keeps track of the successes and failures of these injections of material into the EC population, and the context in which these occurred. This information is then used to improve the accuracy of the extraction and re-injection process, thereby improving the performance of the EC.

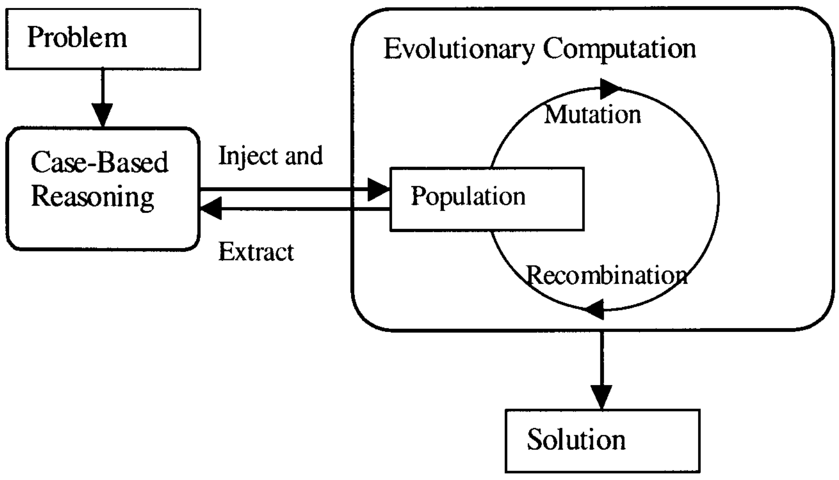


Figure 8. CBR interacts with the population of the EC during evolution.

In Figure 8 cases or individuals can be injected or extracted from and to the Case-Base to improve the EC population. Individuals from a population intended to solve a single problem can then be stored for use on different problems. This reuse can be implemented by an inject-and-test mechanism so that the effectiveness of injecting a given individual into a population for a particular problem can be stored as part of that individuals' Case. The system can then learn when it is best to inject which Cases as individuals into the population for a new problem. This form of hybrid has been investigated by Louis (1993).

A solution to the problem of knowledge acquisition is to use a 'knowledge-lean' method of solution generation, e.g. the use of EC Algorithms for the sub-task of adaptation (Maher *et al.*, 1996), (see Figure 9). EC algorithms require less knowledge than other AI approaches (see Chapter 1 section 4) to produce an acceptable result. Maher *et al.* (1996) uses a simple EC Algorithm to provide this solution.

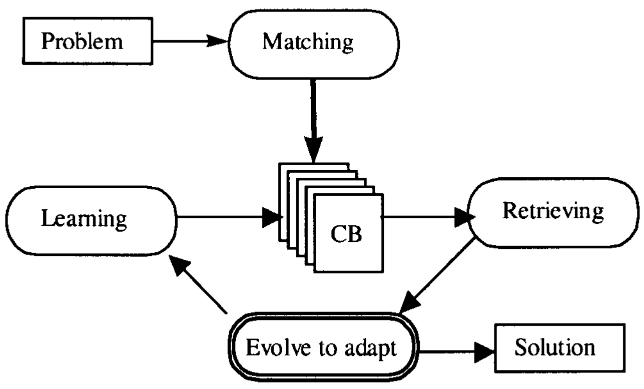


Figure 9. Using EC as a knowledge lean method of achieving adaptation in CBR.

EC techniques are 'knowledge lean' because they can produce new solutions from small bits of solutions without the need for knowledge of how to generate a solution in that particular domain. The only knowledge that is required is that of the fitness function, which gives the evolving population a target to compete for.

To use EC for engineering design problems the representation of the object to be designed (phenotype) must be converted into a genotype representation. The traditional EC operators of crossover and mutation can then manipulate this genotype representation. The result of this is a population of individuals that represent solutions to design problems. Each of these individuals then becomes the basis of a Case in the CBR system. The Case design is discussed in Chapter 6. In this way Cases in a Case Base Reasoning system can be represented as individuals in an evolving population. The EC process proceeds as in Figure 8.

Another hybrid approach is to use EC to create a Case-Base for CBR. Figure 10. gives an example of how CBR could be integrated with EC in this way. This last hybrid is the approach taken in this thesis. In this approach EC is used as a knowledge-lean technique to provide CBR with a Case-Base, similar to the use of EC as a knowledge-lean adaptation mechanism as seen in Maher *et al.* (1996).

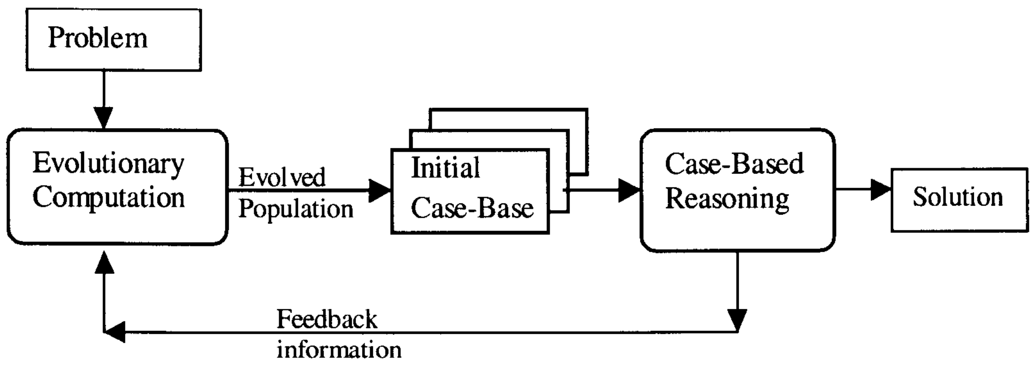


Figure 10. EC is used to generate an initial Case-Base for CBR.

In Figure 10 Information gained from the CBR process can be fed back into the EC to improve its capabilities. For example, an analysis of the initial case-base could show that the evolved solutions are modular, and these modules could then be used by EC to reduce the processing time required to design solutions.

3.3.1. Phenotype to genotype mapping

The following example of a phenotype to genotype mapping is based on the representation used in Miller *et al.* (1998a). The phenotype in this example is a digital logic circuit (Figure 11). This phenotype implements the truth table given in Table 6 and is represented in a Case by the genotype shown in Table 7. This representation is a simplification of that used in this thesis.

The digital circuit in Figure 11 is an example of the phenotype of an FPGA program created by EC:

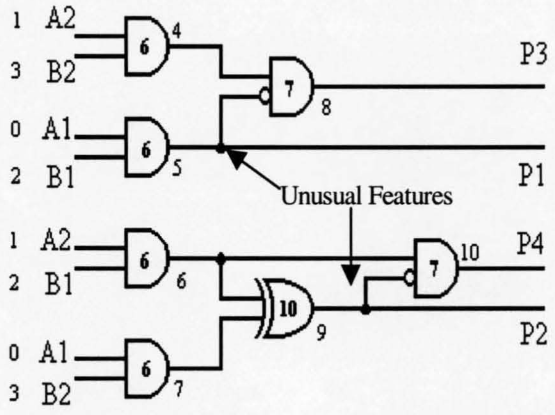


Figure 11. An evolved 2-bit multiplier (a Phenotype) (Miller *et al.*, 1998a).

The unusual features shown in Figure 11 are examples of why the evolved solutions are difficult to understand. In the conventional 2-bit multiplier P1 would is not used as an input to P3 and P2 is not used as an input to P4. These features illustrate some unconventional design properties.

Inputs Binary	Outputs Binary	Input Decimal	Output Decimal
0000	0000	0×0	0
: :	: :	.	
1111	1001	3×3	9

Table 6. Illustrated truth table used for evaluation of potential solutions.

The genotype representation of the phenotype shown in Figure 11 is obtained as follows. The genotype (Table 7) is an ordered array of cells, each containing 3 integers, followed by cells with a single integer that represent output connections (Table 7). The structure of an individual cell is represented as a series of three integers. The first two integers in each cell represent the cell inputs and the third represents the cell function, see Table 7. Truth tables represent the functionality of a program, (see the shaded area of Table 6). Here, the third integers in Table 7 (in bold), can be seen inside the gates in Figure 11. This number represents the gate types: - logical A AND B, A AND NOT B and A XOR B, (**6**, **7** and **10** respectively). The shaded cells in Table 7 represent the cell numbers from which outputs P1-P4 (Figure 11) are taken.

The phenotype in Example one is represented by the genotype in Table 7:

1 - 3 - **6**	0 - 2 - **6**	1 - 2 - **6**	0 - 3 - **6**
4 - 5 - **7**	6 - 7 - **10**	6 - 9 - **7**	5
9	8	10	

Table 7. Genotype for an evolved 2-bit multiplier, an example of a program for a FPGA (Miller *et al.*, 1997).

This ordered array of cells in Table 7. is indexed left to right, top to bottom, the inputs from the truth table are numbered as indices 0 to 3 (as there are 4 inputs, see Figure 11). So, this genotype in Table 7. has cells with indices 4 to 15 (see the numbers outside each gate in Figure 11), where cells 11 to 14 (Shaded) represent output connections and cell 15 is unused. These output connections are represented by P1 to P4 respectively, in Figure 11.

Table 8 illustrates the meaning of the gate types used in Table 7. The above genotype is a slight simplification of that used by Miller *et al.* (1998a) as the gates in this circuit only have two inputs, not three as in Miller's representation, so the third input in each cell is irrelevant in this case, and has been removed from this example.

Representation	Logical statement
6	a AND b
7	a AND NOT b
10	a Exclusive-OR b

Table 8. Example Adapted from Miller *et al.* (1997).

The numbers 6,7 and 10 are fixed for each logical gate type and are assigned their values from a simple table giving a number for each operator. These numbers are taken from a larger set of logical statements available in Miller's representation.

Many evolutionary computation techniques implement two operators, crossover and mutation. Each of these operators is used to change genotypes, in order that they might improve in some desired quality.

Crossover is a matter of taking two genotypes, selecting one or more crossover points, and swapping over the resulting sections of the genotypes to create a new genotype. Here is a simplified example using two cells:

1	3	6

^Crossover point

and

4	5	7

With a single crossover point, produce two more genotypes:

1	3	7

And

4	5	6

Mutation

Mutation is a simple operation, an element is chosen at random and that element is changed to a randomly chosen number from the allowed set of numbers, e.g. the following cell (The first cell in the genotype in Table 7).

1	3	6

mutated could become:

2	3	6

So the original cell representing an AND gate (the number 6) connected to inputs 1 and 3 (see the top-left gate in Figure 11) has now been mutated to an AND gate with inputs 2 and 3. The mutation could equally have been the second input or the function itself, e.g. the AND gate (the number 6) could be mutated to an 'a AND NOT b' (gate number 7, Table 8).

Crossover and mutation are used to randomly change the cells of the genotype. In this way the functionality of the FPGA program can be changed. The new genotype is then evaluated (see section 3.3.3) to determine whether or not it has improved.

Miller's representation, algorithm and the constraints involved in the operation of the EC technique employed are discussed further in Chapter 4.

3.3.2. Genotype to Phenotype mapping

Genotype to Phenotype mapping is the same as the above but in the opposite direction, so a program (phenotype) is created from the genotype e.g. the genotype in Table 7. would be used to create the example program in Figure 11.

3.3.3. Evaluation and selection of genotypes

There are several ways in which evaluation of the genotype can be achieved. The simplest method is to compare the phenotype solution to the required solution. In the FPGAs the functionality of the phenotype solution is compared to a truth table, e.g. a truth table representing 4-bit multiplication (see appendix 2.1), allowing different circuits (different structure and behaviour) to be compared. The functionality, behaviour and structure are explained in Chapter 6.

In the case of FPGAs an attempt is made to evolve a circuit that has the same functionality as the conventional solution, but one that is much more efficient, either in size, speed or both. The genotype shown above in Table 7, is an example of a software program used to program an FPGA.

There may be more than one evaluation criterion. In addition to functional correctness, size, speed and cost may also be of interest. Additional evaluation criteria can be applied during the evolutionary process or after the first evaluation criterion of functionality has been met. A discussion of these additional costs is presented in section 4.2.1.5.

A genotype is selected if it is evaluated to be better or equivalent to the genotype before mutation and crossover. The selected genotype can then be subjected to mutation and crossover, potentially improving it further. The processes of crossover, mutation, evaluation and selection are repeated until a solution is achieved. Chapter 4 further explains in detail how standard EC ideas are applied to the problem of logic design in this thesis.

3.4. CBR and SR

The SR domain is complex, as there are potentially an infinite number of software problems and solutions. Approaches to SR range from application specific generators to general programming languages e.g. Pascal, C++, KL-1. These reuse systems themselves can vary greatly in their underlying model e.g. sequential, parallel. The complexity of the SR domain means that it is difficult to build a model of the domain. This is where CBR, specifically representing knowledge as Cases wins over other Knowledge Based (KB) techniques. This is also where EC can be used, as it is possible that not all of the knowledge required to solve any problem will be in the Case-Base, a knowledge lean approach to the problem is required. EC has been successfully applied to CBR systems to facilitate knowledge acquisition in this manner (Maher *et al.*, 1996; Tanaka *et al.*, 1994; Lenart *et al.*, 1994).

A goal of this work is automated CBR adaptation. Hanney (1996) and Kolodner (1993) give taxonomies of adaptation transformations and adaptation methods, as discussed in Chapter 3 section 2. These taxonomies are general although not every application uses every adaptation technique, many do not perform any adaptation. The forefront of adaptation is in being able to automatically extract rules from Cases, and apply them to new Cases. Hanney (1996) gives two algorithms to achieve this, but they are simple and do not cover the more intricate aspects of adaptation. Table 9 shows the areas of importance to this project and which research has covered these areas.

Researcher Name	System Name	Adaptation	Hierarchical Case-Base structures	Software reuse	Automated rule extraction & application	Complexity of Case
Hanney		√			√	Simple
Smyth	Deja Vu	√	√	√		Structured
Wilkie	INRECA	√	√			OO Structures
Giraud-Carrier	FLARE	√			√	Nominal and linear
Fuchs		√				Avoidance
Fox Leake	ROBBIE	√				Simple
Bergmann	PARIS	√	√			Limited

Table 9. CBR-SR and CBR adaptation techniques research.

These categories are not clean-cut but give a general view of each area.

3.5. Summary

A hybrid CBR-EC system gives an opportunity to develop a new method of designing digital circuits. EC is necessary, as the existing methods for digital circuit design are insufficient. CBR techniques of matching, retrieving, adapting and learning are required in conjunction with EC techniques, as EC alone cannot produce large digital circuits due to the limits of computational power. CBR can overcome this EC limitation by extracting and applying the principles involved in the EC solutions. CBR is able to repair solutions to a given problem using parts of other solutions that are better, or perfect.

Hunt (1995) has observed that solutions that have the same quality can be very different in structure, behaviour, and/or functionality. This provides a good source of knowledge for CBR principle extraction.

This study of CBR applied to EC-produced software programs for the FPGA shows that CBR can be used to solve the problems of understanding, scaling and optimising solutions using all of the information gained from many individuals in the population.

In Chapter 4 the encoding of a digital circuit into a genotype and the characteristics of the evolutionary algorithm are given. This evolutionary algorithm designs fully functional circuits. This evolved data is discussed in Chapter 4. Chapter 5 discusses the techniques of landscape analysis developed by Vassilev *et al.* (1997a/b, 1999a/b, and 2000) that show the principles by which an effective evolutionary search may be conducted. Vassilev's examination of the evolutionary search illustrates the *processes* involved.

The process of discerning design principles from the evolved data can be seen as a form of data mining and is an examination of the *products* of the evolutionary process. This examination can make recommendations about useful components and sub-structures that feed back into the evolutionary algorithm and so improve the evolvability of the circuits in question and enhance our ability to understand the new designs. Further to this it is shown how this data mining can overcome the scaling problem in an automated manner.

Chapter 4 shows how much human intervention is required in EC, namely the definition of the PLA file, the target geometry and a parameter to tell the evolutionary algorithm when to stop. Chapter 4 shows that EC techniques alone can out perform the conventional techniques and also cope with different classes of problems within the domain of FPGA programming. Chapter 5 examines the scalability of the evolutionary algorithm alone, showing limitations in the method.

4. FPGAs and digital circuit program design

This chapter discusses an EC technique for digital circuit program design. It shows how EC can be used for designing FPGA programs and then discusses the practical aspects of program implementation.

4.1. Digital Circuit Evolution

The Section 4.1.1 shows how a FPGA program can be represented as a graph for Cartesian Genetic Programming (CGP). Then an evaluation method is given followed by a description of the CGP algorithm.

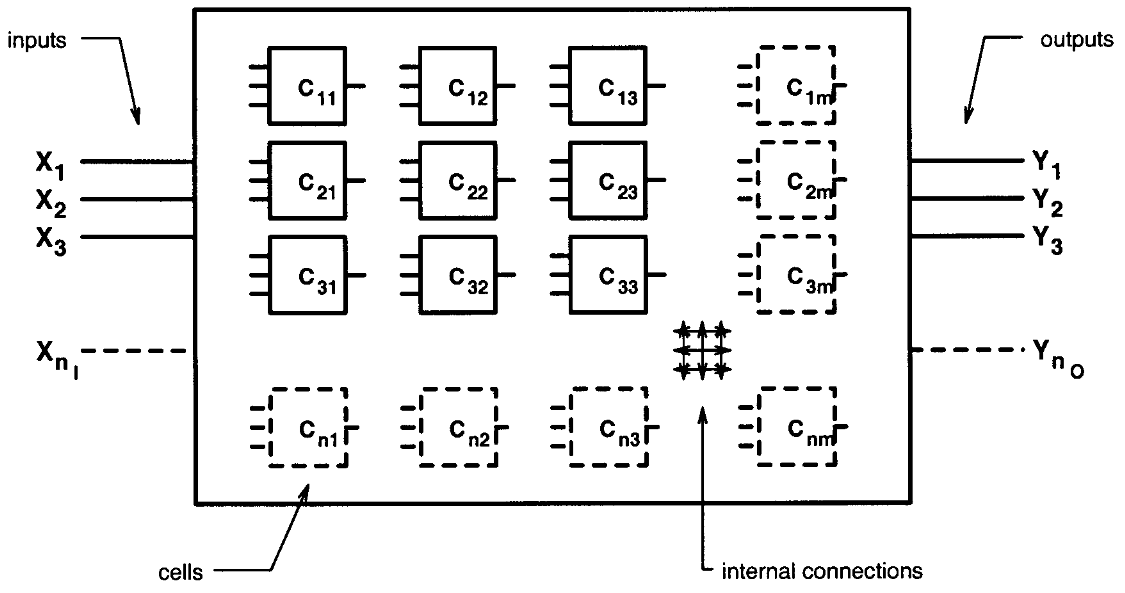
4.1.1. Encoding a Digital Circuit as an Indexed Graph

The encoding of a digital combinational circuit into a genotype, which is presented in this thesis, is based on earlier models that can be found in Miller *et al.* (1997); Miller and Thomson (1998b); Miller and Thomson (1998a). A digital logic circuit is encoded as a more general graph based computational model called CGP (Miller, 1999a). CGP is a graph-based form of genetic programming. Other graph based genetic programming forms are Parallel Distributed GP (PDGP) proposed by Poli (1997) and Parallel Algorithm Discovery and Orchestration (PADO) (Teller and Veloso, 1995). CGP represents a data-flow graph (Banzhaf *et al.*, 1998).

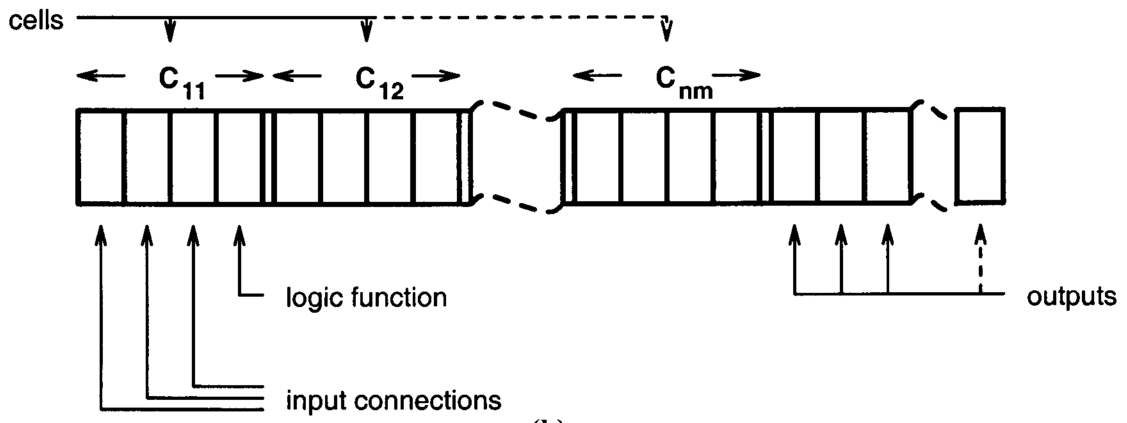
In CGP a digital electronic circuit is encoded as an instance of a program in which functional nodes are connected together to perform any given computational task on binary data. A CGP program is a rectangular array of nodes. These nodes each represent an operation on the data at its inputs or on the outputs of other nodes. Each node may implement any convenient programming construct (e.g. if, switch, OR, $\times$ etc.).

The genotype is a linear string of integers and is characterised by three parameters: the number of columns, the number of rows, and levels-back. The first two are the dimensions of the rectangular array and the last is a parameter that controls the internal connectivity. It determines how many columns of cells to the left of a particular cell may have their outputs connected to the inputs of that cell. This parameter is also applied to the program outputs. The cells and outputs are maximally connectable when the number of rows is one and levels-back is equal to the number of columns. Minimal connectivity occurs when the number of columns is one and levels-back is 1.

In this thesis a particular form of CGP is adopted in which all cells are assumed to have three inputs and one output and all cell connections are feed-forward. In general CGP the cells may have multiple inputs and outputs and the numbers of these would be encoded into the genotype for the cell. Also in general, primary outputs could be treated as clocked inputs thus allowing the CGP programs to possess internal states. The genotype and the mapping process of genotype to phenotype are illustrated in Figure 12 (a and b).



(a)



(b)

Figure 12. The genotype-phenotype mapping: (a) an $n \times m$ geometry of logic cells with n_i inputs and n_o outputs, and (b) the genotype structure of the array.

Letter	Function	Letter	Function
0	0	10	a EOR b
1	1	11	a EOR NOT b
2	a	12	a OR b
3	b	13	a OR NOT b
4	NOT a	14	NOT a OR b
5	NOT b	15	NOT a OR NOT b
6	a AND b	16	a AND NOT c OR b AND c
7	a AND NOT b	17	a AND NOT c OR NOT b AND c
8	NOT a AND b	18	NOT a AND NOT c OR b AND c
9	NOT a AND NOT b	19	NOT a AND NOT c OR NOT b AND c

Table 10. Available cell functions.

Functions 16 to 19 in Table 10 are all binary multiplexers with various inputs inverted. The multiplexer (MUX) implements a simple IF-THEN statement (i.e. IF c = 0 THEN a ELSE b). It is important to note that multiplexers can be considered to be atomic both formally and from an implementation point of view. They are atomic in that they are universal logic modules (Chen and Hurst, 1982) so that they can be used to represent any logic function. They are atomic in that some modern FPGAs now use a multiplexer based architecture so that all two input gates are synthesised with multiplexers. The specific FPGA that is used as a reference in this thesis, the Xilinx XC6216, uses a multiplexer based architecture and supports only the cell functions given in Table 10. It should be noted that several functions include inverters within the same cell, see functions 7, 8, 9, 11, 13, 14, 15 and the multiplexers 16 to 19.

The genotype is a list of connections and cell functions shown in Figure 12 b. In general the connections can be thought of as addresses in data, thus provided the function set is appropriate for a particular data type, the genotype is data independent.

It can be seen in Table 10 that only functions 16 to 19 use all three inputs and that some functions are actually constants with an output independent of the inputs (letters 0 and 1). Thus the genotype can contain completely redundant genes. This type of redundancy is referred to as input redundancy. Cells may also not have their outputs connected in the operating circuit between the primary inputs and outputs; these collections of genes (3 connections, 1 function) are also redundant. This is called cell redundancy. Another form of redundancy called functional redundancy is more typical of genetic programming. This is where a number of cells implement a function that can be implemented using fewer cells. A specific instance of this kind of redundancy is behavioural redundancy, where two or more cells implement identical behaviour.

It is important to emphasize that cell outputs may be re-used and when a program is used to evolve the genotypes the amount of re-use of sub-calculations is determined entirely automatically.

4.1.2. Calculating the Fitness of a Genotype

All functions are specified by a truth table. The fitness of a genotype is the number of correct output bits. Thus for the one-bit adder with carry seen in Figure 13 there are 8 input cases and 4 output cases, each output case having 2 bits, this gives 16 output bits, shown in Table 11 (Cout and S in shade). A fully correct circuit would have fitness 16. In practice the fitness of a circuit is calculated using 32-bit arithmetic. Thus the binary data is handled as 32-bit unsigned integers and all the operations defined in Table 10 are 32-bit operations. A truth table with 3 input variables is then represented as a single line (Poli, 1999). For example the truth table of the 1-bit adder with carry e.g. Table 11 is represented as Inputs: 170 224 240 Outputs: 232 150.

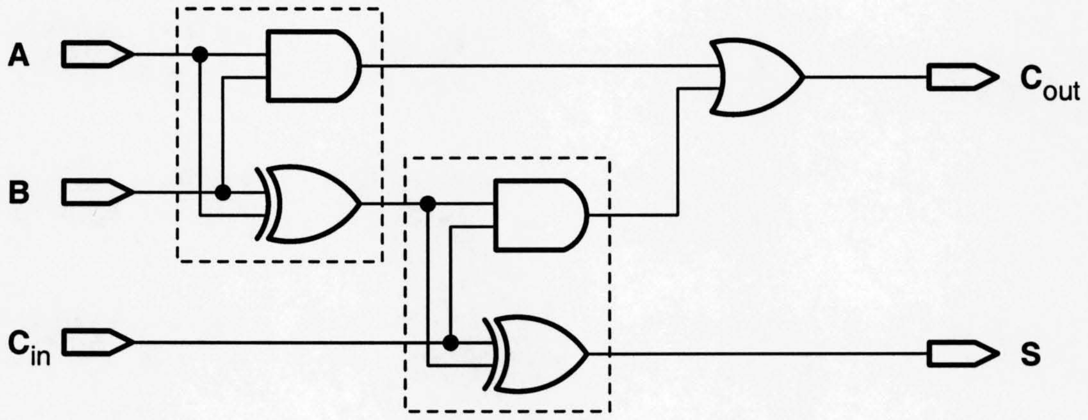


Figure 13. One-bit adder with carry.

A	B	Cin	Cout	S (sum)
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

Table 11. Truth table for a 1-bit carry adder.

Each column of a Truth table for the required function is divided into 32-bit sections, which are then represented by 32-bit integers. In Table 11 there are only 8 bits in each column, so each column is one 32-bit integer. For example, the first bit in column A becomes the least significant bit in the 32-bit binary number, the last bit being the most significant bit. In this case column A is now represented by a 32-bit number (170 decimal), where only the first 8-bits of the 32-bit number are used. Then for example the AND operation using 32-bit operands would be 170 AND 224 the result of which is then compared to the required output 232, effectively evaluating 32 binary input – output combinations simultaneously.

Additional fitness functions can be added to further refine the evolutionary design. For example, to reduce the size of the program the number of gates in each program can be counted and a score based on the number of gates used could be combined with the main fitness function that favours functional correctness.

This size scoring fitness function is used in this work to optimise the size of programs with respect to the number of gates used. Once 100% functional correctness has been evolved the number of gates in the evolving program is used to give an additional score to the score for 100% functional correctness. During evolution with this additional fitness function many of the attempts to improve (mutate) a program fail, producing a less than 100% functional solution. In this case the failed programs are discarded. Only programs that remain 100% functionally correct after a mutation receive an additional score for their size. This ensures that smaller programs achieve a higher fitness score and so the sizes of the programs are minimised.

4.1.3. The evolutionary Algorithm

The evolutionary algorithm, developed by Miller, used to produce all of the evolved circuit designs in this thesis is a simple form of $(1 + \lambda)$-ES evolutionary strategy (Schwefel, 1981; Bäck *et al.*, 1991), in this work λ is 4. $(1+\lambda)$ represents the size of the population in the ES strategy and λ can be any integer from 0 upwards. Experiments that were reported in Miller (1999a) indicated the efficiency of this approach. The algorithm is as follows:

Step 1 Randomly initialises a population of genotypes *.
Step 2 Evaluate fitness of genotypes. Stop if criterion reached.
Step 3 Copy a fittest genotype into new population.

Step 4 Fill remaining places in population by mutated versions of fittest genotype *.
Step 5 Replace old population by new and return to step2.

* Subject to constraints that ensure the feed-forward nature of circuits and levels-back connectivity.

The mutation rate was defined as a percentage of the genes in a single genotype that were to be randomly mutated *. It was necessary to adjust the mutation rate if the genotype length was too small, to prevent zero mutation. In this work a mutation rate which resulted in 3 genes being changed in each genotype was found to be suitable. A suitable population size was found previously by experiment using a two-bit multiplier circuit. The experimental parameters were as follows:

- Number of rows - 1
- Number of columns - 10
- Levels-back - 10
- Mutation rate - 8% (3 genes)
- Number of generations - up to 150,000
- Gates used – 6, 7, 10 (Table 10.).

When the number of rows is 1 and the number of columns equals the 'levels-back' the FPGA program is minimally constrained with respect to the connectivity between gates and between program inputs and outputs and gates. This means that any gate can use any program input or gate that precedes it in the genotype as an input, and can connect it's output to the input of any gate that follows it in the genotype, or a program output. In contrast, if the number of rows was equal to 2 and the number of columns equal to 5 then no gate could connect directly to another gate in the same row. If the 'levels-back' was equal to 1 then no gate could connect to another gate or program input/output, that was more that 1 column from that gate, e.g. a gate in column 5 could only have gates in column 4 as potential inputs. These constraints exist to allow FPGA programs to be designed to fit into specific areas of an FPGA. The need for constraints is discussed in Section 4.2.

The minimum number of evaluations required to obtain a 0.99 probability of successfully obtaining a 100% functionally perfect solution (fitness equal to 64 in this case) was calculated by Miller *et al.* (2000a). Millers results, obtained for the 2-bit multiplier, show that the optimal population size was 4 and minimum number of evaluations was 81,608. This number of evaluations (81,608) was the number of evaluations required to give a 0.99

probability of successfully obtaining a perfect solution. This is a measure that gives the minimum amount of computational effort that is required to ensure a probability of 0.99 that a 100% functionally perfect solution will be achieved (Koza, 1992). A probability of 1 means that a 100% functionally perfect solution is guaranteed, whereas a probability of 0.99 almost guarantees a 100% functionally perfect solution. A probability of 1 can never be achieved as evolutionary algorithms depend on random processes. Further measures of evolvability are given in Vassilev *et al.* (2000).

4.2. Practical Aspects of Circuit Implementation

One of the objectives of this thesis is to aim to evolve as novel and efficient digital logic circuits as possible. The table of logic functions Table 10 that has been used is modelled on the resources that are available on modern FPGA platforms. The experiments described have assumed that there are no practical constraints imposed by wiring. In practise the routing of connections between components is a significant factor in the successful implementation of a circuit (See Section 4.2.1.5.). Other representations of digital circuits in which the routing is explicitly taken into account have been devised (Miller and Thomson 1998b, 1998a). To improve the potential routability of circuits evolved using the techniques described here one can adjust the levels-back parameter so that it takes much lower values. The complete investigation of the influence of this on circuit routability is a subject for further work. It was shown by Miller and Thomson (1998b, 1998a) that the dominant factor in the evolvability of the circuits is the amount of functional resources that are available, however increasing this tends to produce less efficient circuits. Conventional logic synthesis techniques minimise the symbolic representation of a circuit and then carry out technology mapping. This is a process of trying to rewrite the symbolic logic into a form that can be implemented with whatever gates are available on the chosen platform. To do this efficiently is a non-trivial exercise. Such a process is unnecessary when evolving a circuit using the gates available on the device.

4.2.1. Evolved Data and Interesting Problems

It is clear that the number of input combinations in a truth table grows exponentially with the number of inputs. Thus it is not practical to evolve very large truth tables (> 25 input variables). Conventional logic synthesis techniques (See Section 2.2) can handle hundreds of input variables. Thus the question arises: what is the point of evolving solution programs for truth tables by assemble-and-test? The answer is that interesting functions could be evolved. These interesting functions may be more efficient, using fewer components or by being faster than their conventional equivalents. These are useful functions which can be series of functions of increasing scale but similar function. These functions can be reused to build larger circuits. Classic examples of this are arithmetic functions, namely, binary adders and multipliers. These smaller functions can then be combined to create larger circuits (e.g. digital filters).

Another useful application of the use of evolution for digital circuit design is that of re-engineering. Often in industrial situations, existing solutions have to be replaced by new solutions that take into account small changes in the specification of the problem. In this case only a small area of the solution requires re-design, something that EC can probably achieve more efficiently using reuse techniques such as Lockdown discussed in Section 5.2, and CBR discussed in Chapter 6. Conventional methods often require that the old solutions are completely re-engineered (Scherr, 2000). This manual re-engineering is a very inefficient approach that becomes more difficult as the differences increase between the original hardware platform and the new hardware platform.

One further useful application stems from the fact that EC techniques can also carry out the technology-mapping phase of digital circuit programs. In an industrial situation programs are developed to fit onto a specific hardware platform. As technology progresses new hardware platforms replace the old ones and the existing programs do not map onto the new platforms. Conventional techniques require that a completely new program is developed from scratch and then mapped to the new hardware platform (Scherr, 2000). This again is inefficient whereas EC techniques could reuse existing programs and adapt them to the new platforms.

As Digital circuit evolution suffers from exponential growth in complexity as the number of inputs increases, research has also been undertaken to reduce the complexity of the problem and to improve the efficiency of the evolutionary search. These ideas are presented in Section 5.6.2.

If a particularly efficient adder or multiplier can be evolved this could be used as a building block for adders of any size. However there is another interesting reason to try to evolve arithmetic functions. A series of examples with increasing numbers of inputs could be evolved and then it may be possible to deduce the general design principle. If this is possible then by using this principle it may be possible to obtain new designs for arithmetic functions of any number of input variables. It is these principles that are employed in the design of large arithmetic circuits. It is interesting to contrast conventional with evolved designs as the modularity of the evolved circuits can be examined.

A number of key questions emerge:

1. Can more efficient designs for arithmetic functions be found by evolution?
2. Can general principles be extracted?
3. How modular are the evolved circuits?

Next, evolved circuits for one and two-bit adders with carry, and two and three-bit multipliers are shown. The even four-parity function was also studied as parity functions have received much attention from the genetic programming community and it is an interesting function to study as its fitness landscape changes dramatically with the choice of gates used to build it.

In a simplistic view, term 'fitness landscape' refers to the idea that genotypes can be seen as points on a landscape from the point of view of its fitness. In this simplistic view the highly fit genotypes are points on mountain peaks and the low fitness genotypes are points in the valleys of the landscape.

4.2.1.1. One-bit Adder with Carry

Some FPGA manufacturers are adopting novel designs that have been evolved, e.g. the one-bit adder with carry Figure 13 (Miller *et al.*, 1997).

Any size of carry adder can be built with cascaded one-bit carry adders. This one-bit carry adder and the ability to cascade it, is an example of a general principle. It was seen in the paper by Miller *et al.* (2000a) that an evolved two-bit adder with carry is in fact the conventional two-bit adder with carry, through comparison of the evolved two-bit adder with the evolved one-bit adder with carry. In this way it was shown that it has been possible to re-discover the well-known principle of the ripple-carry adder Figure 3. Thus, in principle, an adder of any size could be constructed.

4.2.1.2. Two-bit Multiplier

Two-bit and three bit multipliers are shown here as examples of how and why evolved circuits can be 20% more efficient, in terms of the FPGA cells given in Table 10, than those designed by conventional techniques. The two-bit multiplier takes two two-bit numbers and multiplies them to produce a four-bit number. The three-bit multiplier takes two three-bit numbers and multiplies them to produce a six-bit number. These can be implemented in block form by the 2-bit cellular multiplier shown in Figure 14. The AND gates carry out elementary one-bit multiplication and two one-bit adders with carry are required to calculate the product bits. The 2-bit cellular multiplier is *cellular* because it is composed of *cells*, two 1-bit adders in this case. One-bit adders with a carry-in of zero can be reduced and one of the AND gates connecting to output P 3 can be eliminated and thus the final most efficient conventional circuit is obtained. It requires seven two-input gates.

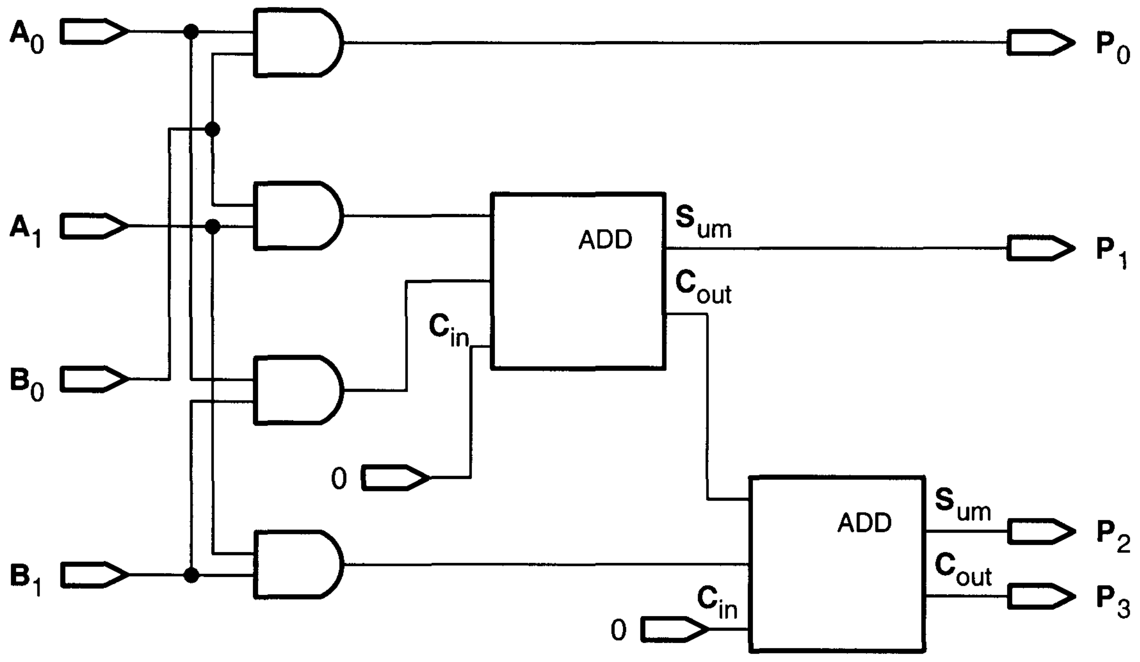


Figure 14. Two-bit cellular multiplier.

Some interesting circuits were evolved in Miller *et al.* (2000a). One circuit of particular interest, shown in Figure 15 a, uses only a single XOR gate yet still carries out two elementary additions. It re-uses sub-calculations in an unusual way. To create the second highest product (P 2) it re-uses the lowest product (P 0) and to create the highest product bit (P 3) it re-uses the second lowest product (P 1). The whole circuit sub-divides into two unconnected parts. The circuit is elegant but also counterintuitive which is more apparent when comparing it with the conventional two-bit multiplier Figure 15 b. It is clear that it is modeling multiplication in an unusual way. The choice of gates that are used to evolve circuits can have a dramatic effect on the ease of evolution. The effects of gate choice are reported in Section 5.4.

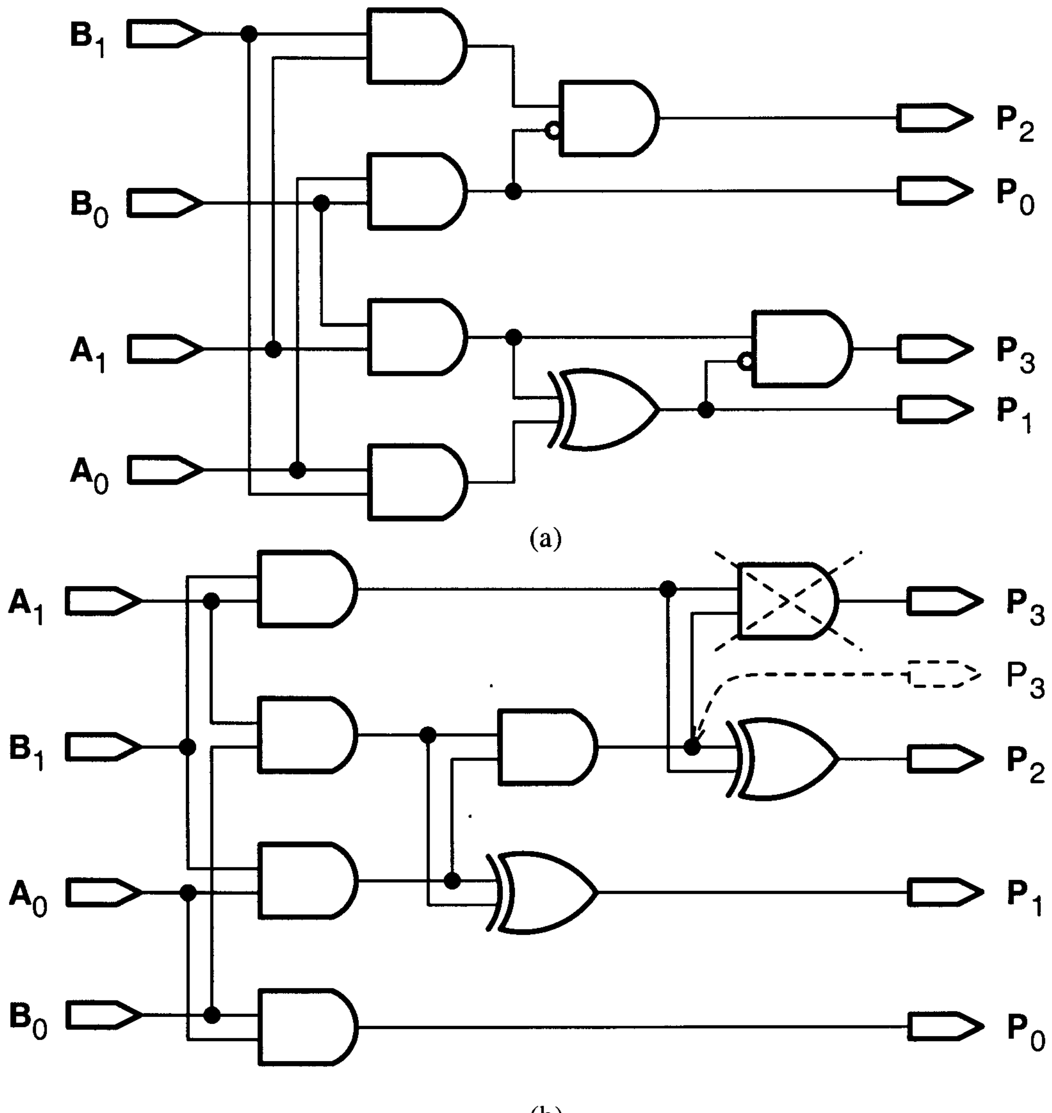


Figure 15. Most efficient (a) evolved and (b) conventional two-bit multipliers.

4.2.1.3. Three-bit multiplier

The conventional three-bit multiplier is again modeled using the familiar process of long multiplication and is built as a cellular array of adders with the nine elementary products being implemented with AND gates.

The evolved circuit uses only 21 gates (Figure 16). This is again 20% more efficient in gate usage than the best conventional alternative (see Figure 17) but is 30% better than the conventional as MUX gates are counted as elementary for the FPGA cells in Table 10. The circuit is difficult to understand and on sight it is not obvious whether it consists of identifiable sub-modules which are useful in building larger systems. It departs radically from conventional principles in that it does not directly synthesise the nine elementary products of the inputs.

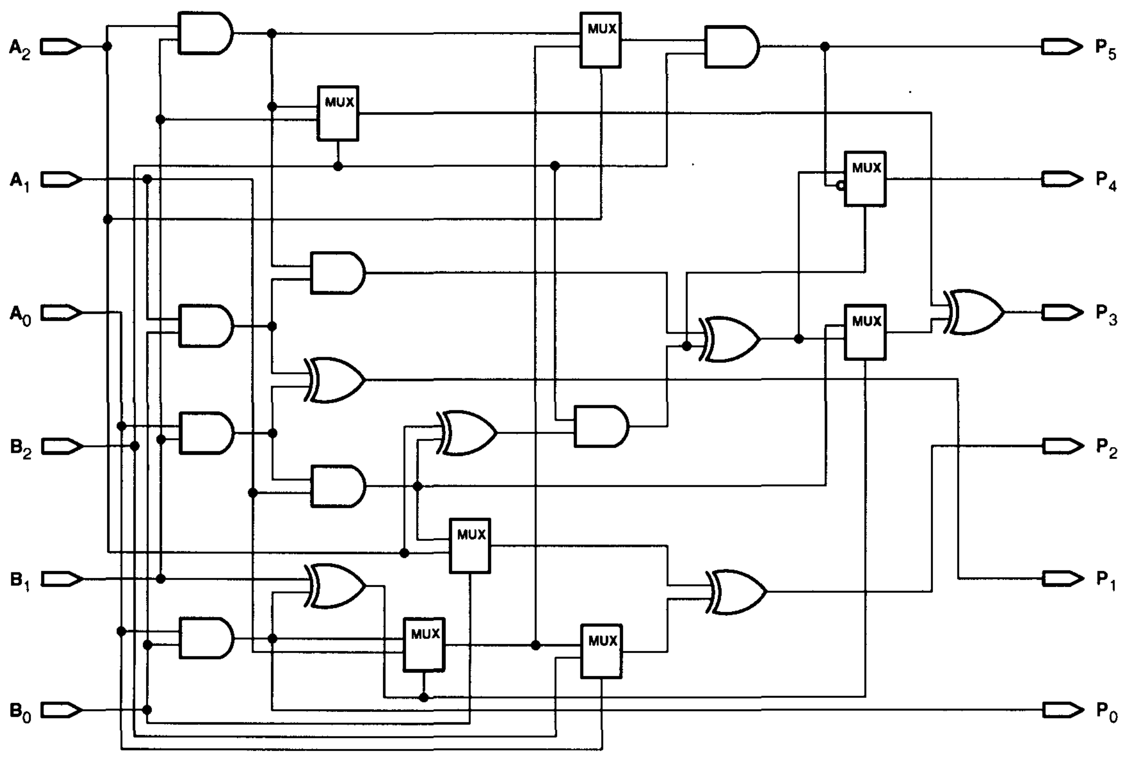


Figure 16. Evolved 3-bit multiplier (21 gates – 14 two-input gates and 7 MUX).

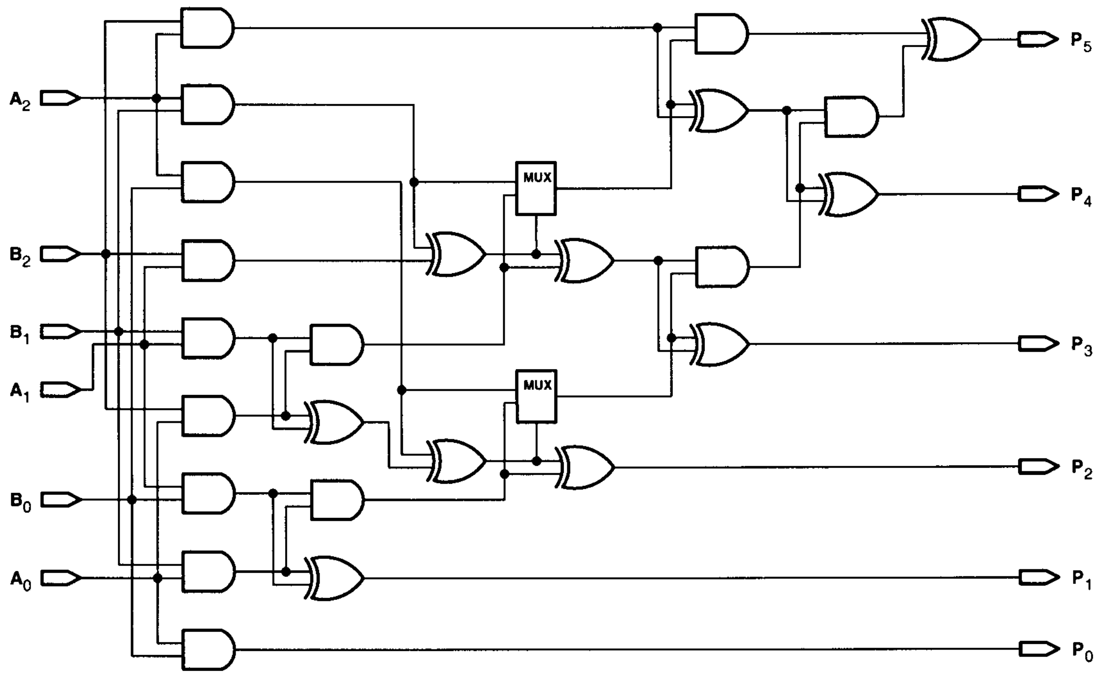


Figure 17. Most efficient conventional 3-bit multiplier using 30 two-input gates (26 gates including 2 MUX).

4.2.1.4. Even Four-parity

The even-parity functions are difficult to evolve when using the logic gates AND, NAND, OR, NOR. The even-parity function returns a 1 if there are an even number of '1s' input to the function. Even-parity functions are difficult to evolve because even-parity functions are most easily implemented using XNOR gates and it is difficult to synthesise XNOR function using this set. The most efficient implementation of even four-parity requires 3 XNOR gates (see Figure 18 a and b). This is an example of how gate choice can dramatically effect circuit evolution.

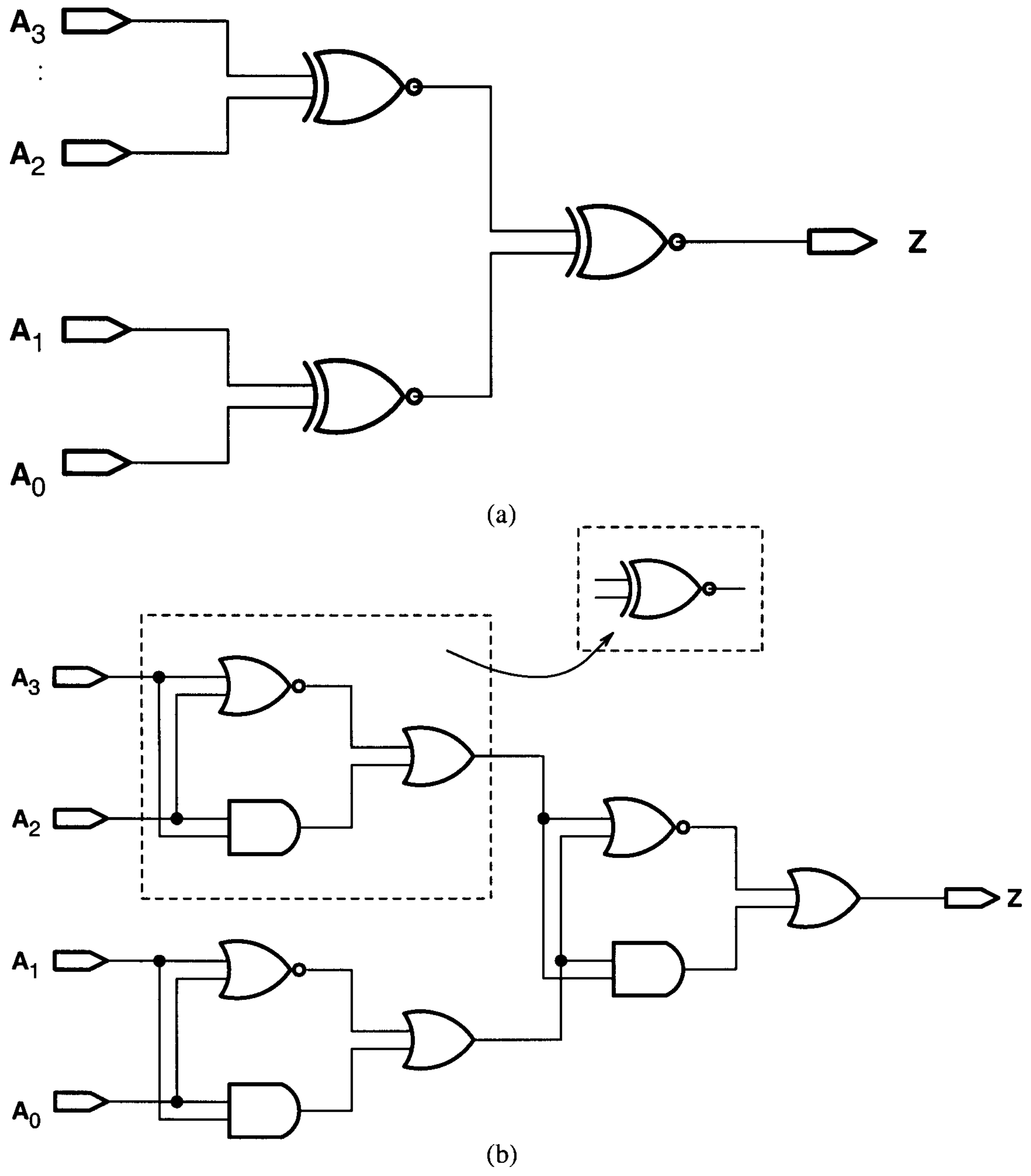


Figure 18. Two representations of the four-bit parity function with (a) gate XNOR and (b) gates AND, OR and NOR.

4.2.1.5. Application to hardware platform

If conventional logic synthesis techniques are used to create a program this program must then be mapped onto a specific hardware platform. This is a non-trivial exercise. Whilst the Cartesian Genetic programs presented here are general and platform independent, CGP is capable of producing platform specific programs, without the need for a separate process for mapping a design to the space available on a specific hardware platform. To make CGP specific to a platform the fitness function requires additional cost criteria. The hardware platform would impose specific costs (Davio *et al.*, 1983) e.g.

- Routing: the physical layout of an FPGA constrains the available interconnectivity of cells.

- Gate times: each type of gate requires a specific amount of time to function.

- Delay criterion: Gate times lead to propagation delays. Clocking ensures that gate times are accounted for, but it can effectively make every gate as slow as the slowest.

In addition to costs that depend on the physical platform, there are financial costs:

- Wire costs: Wires may have to be used to connect two gates together over areas of the platform, using additional materials and silicon area.

- Gate costs: the number of gates used and the number of inputs to each gate.

- Total silicon area, not just the number of devices in the area.

- Additional Costs: Development, Maintenance and Testing.

The cost functions that are used may also depend on design criteria in addition to the target platforms physical constraints, e.g. speed, size, and fault tolerance. If speed is important then it could be necessary to minimise number of gates in series.

4.3. Summary

It was shown in this chapter that a feed-forward digital circuit can be encoded as an indexed graph. The function of the target circuit is encoded as a PLA file with which the evolutionary algorithm calculates the fitness of a genotype. The algorithm developed by Miller (1999a) was given and the human inputs (experimental parameters) to the algorithm that are required in addition to the PLA file were discussed.

The evolutionary process of design is very time consuming and circuits larger than the four-bit multiplier require unreasonable amounts of computing power at present. Some evolved designs were 20% more efficient than the best conventional design for an FPGA, see Appendix 2 (The 4×3 – bit multiplier circuit evolved from the conventional design.)

The challenge here is to evolve large enough circuits to enable the design principles to be identified through comparison of larger circuits to smaller circuits of the same problem class e.g. multiplication. Also larger circuits have greater potential for reuse within the circuit and so efficiency may increase with size. Further to this, if the evolved circuits are modular in nature then the larger circuits should contain a greater number of larger more efficient building blocks, making their identification simpler. The specific problem with designing larger digital circuits using evolutionary techniques is that as the number of inputs grows the time taken for fitness evaluation increases exponentially.

In this thesis one of two different aspects of this problem was examined.

One aspect examines the nature of the digital circuit fitness landscapes and attempts to understand the structure of these landscapes in terms of their smoothness, ruggedness and neutrality (Vassilev *et al.*, 2000). It has been shown how these landscape characteristics should effect the evolutionary search and this has led to improvements in the efficiency of the evolutionary search itself.

The second aspect, the subject of this thesis, which examines the nature of the phenotypes themselves and attempts to discover useful sub-structures and methods that can be reused to create larger circuits. This also facilitates understanding of the novel and efficient designs through derived explanations of how to design larger circuits by reuse of smaller circuits.

These two investigations are an important part of the cycle of evolutionary discovery discussed further in Chapter 8 and illustrated in Figure 37.

The techniques of landscape analysis developed by Vassilev *et al.* (1997b, 2000) are used in finding principles that should lead to a better understanding of the nature of the problem of evolving digital circuits, and hence, effective evolutionary search. The process of discerning design rules and principles from the evolved data can be seen as a form of data mining, thus enabling recommendations to be made about useful components and sub-structures that can also be fed back into the evolutionary algorithm and hence improve the evolvability of the circuits. How this in turn enhances our ability to understand the nature of new designs is discussed in Section 7.5. How design principles could be identified and reused to solve the scaling problem using CBR, is the subject of the next section.

5. Evolutionary CBR for Automated Design of digital circuit programs.

It is argued that it might be possible to identify and reuse new, efficient, and generalisable principles of design by studying evolved program designs of gradually increasing scale. In this chapter this theory is explored in the field of digital arithmetic circuit programs. The design knowledge that is discovered can be reused to improve the evolutionary algorithms search capabilities and hence increase the likelihood of identifying new principles. These principles could explain how to build systems that are too large to evolve without using larger modules in place of the current atomic logic gates. The knowledge discovery process is realised through the combination of EC with Case-Based Reasoning (CBR).

Arithmetic circuits evolved using the EC described in the previous Chapter 4, are examined specifically because conventional design approaches exploit the modular nature of the arithmetic functions to build increasingly large functions from smaller building blocks. If these building blocks also occur in the evolved designs then it might be possible to identify them and their potential uses to build functions of any size. Further to this, by examining functions of increasing size, and comparing them it may be possible to identify a general design principle.

Firstly, in Section 5.1 the research set-up is described, then an explanation of the modularity of digital circuit program evolution is followed by details of an attempt at error correction in flawed evolved programs. The effects of function choice on evolvability are then given, showing how the modular analysis can be fed back into the evolutionary process. Further techniques for enhancing the capabilities of the evolutionary technique are then given. Next an overview of the problem of identifying principles in evolved circuits is given followed by a description of how CBR can be used to solve these problems and the experiments carried out to illustrate these techniques are described.

5.1. Introduction

A population of FPGA programs is evolved using EC, to meet a given functional specification, which is used to assess the quality of the programs being evolved. This functional specification is represented by a truth table in the form of a pla file (see Table 2), allowing exact calculation of functional correctness (some problems e.g. signal processing, have no precise functional specification since they involve the conversion of analogue signals to digital signals). Researchers have been successfully evolving electronic circuit programs by adopting constrained or unconstrained methods.

Constrained methods either temporarily ignore the need for robustness, or constrain the available choice of sub-programs and interconnection topologies. Unconstrained methods give evolution maximum freedom to exploit the full repertoire of behaviours that the device can produce (Miller and Thomson, 1998a; Thompson, 1996).

A common problem with evolving digital circuits is that of errors. The EC method does not always produce perfect solutions, and even if it does, perfect solutions are difficult to understand. As the size of the programs increases, the EC produces a lower percentage of perfect solutions. For a limited class of tasks, FPGAs do not require perfect solutions, e.g. a Digital Signal Processor may process Analogue signals, so there is no precise input specification. However, other tasks do require perfect solutions, e.g. in addition programs, a 'best' solution produced by EC may require repair, and the information required to do this may exist in another of the EC solutions.

The remainder of this thesis describes on-going attempts at overcoming the above problems with evolving circuit programs by effective integration of EC and CBR techniques. This research aims to adapt and reuse genetically evolved FPGA programs, and the sub-programs within these programs, to create larger programs at a reasonable computational expense.

5.2. Digital Circuit Evolution and modules

The evolved circuit designs are produced by Cartesian Genetic Programming with truncation selection and mutation. The latter is defined as a percentage of the genes in a single genotype which are to be randomly mutated. The population consists of $1 + \lambda$ genotypes where λ is usually about 4. Initially the elements of the population are chosen at random (see Section 4.1). To update the population, the operator for mutation is applied to the fittest genotype, and thus an offspring is generated. The offspring together with the parent constitute the new population. This mechanism of population update has some similarities to that employed in other evolutionary techniques such as $(1 + \lambda)$ Evolution Strategy (Schwefel, 1981; Bäck *et al.*, 1991) and the Breeder Genetic algorithm (Muhlenbein and Schlierkamp-Voosen, 1993). This algorithm has enabled the automatic discovery of highly efficient circuits that are unusual in construction. The one-bit adder that was used as an example, Figure 19, was evolved and it required two gates less than the conventional design. The MUX gate occurs in an unfamiliar configuration, implying that these gates are very useful building blocks for the construction of adder circuits.

Interestingly, this one-bit adder automatically emerged as a building block in an evolved two-bit adder. This suggested that it would be worth while attempting to evolve larger and more complex circuits, such as the two-bit and three-bit multipliers. Indeed it was found that some of the evolved three-bit multipliers were 20% more efficient than the most efficient conventional design.

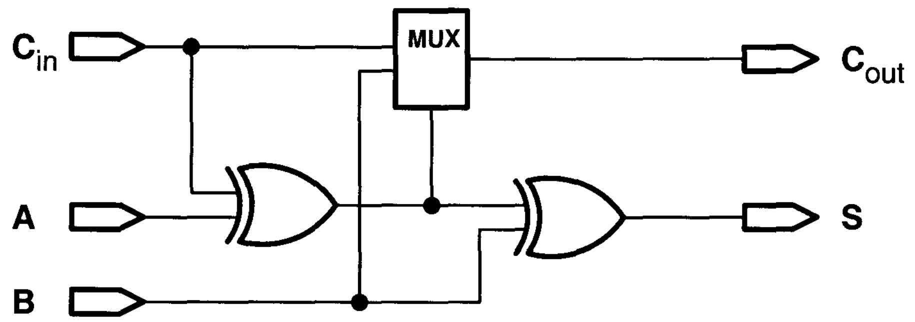


Figure 19. Evolved one-bit adder with carry.

5.3. Errors in evolved solutions

The prospect of error repair was investigated. This was done with the intention of developing a method to repair the best of the faulty solutions that resulted when the problem was too large and complex for CGP to handle. In the case of multiplier circuit programs repair was possible in the 3x3-bit multiplier class when a perfect solution was known. By replacing the faulty areas of a non-100% fit program, with the relevant areas from a 100% program, a faulty solution could be fixed. This was done to see how difficult it was to repair faulty solutions. This approach worked but resulted in inefficient solutions. Faults tended to be the same in each program. This is because specific outputs are significantly more difficult to synthesise than any other part of the solution. This means that it was unlikely that a fix for one faulty solution could be found in another faulty solution, so it followed that creating one perfect program from two faulty programs was not a solution to the scaling problem in multiplier circuits.

If a fix for a faulty solution was evolved especially to fix a specific fault, the small error fix tended to result in an increase in the size of the program by approximately 50%. This is again due to the faults tending to be in the most difficult to synthesise areas of the program. This may not be the same for all problem classes e.g. adders, filters.

Repair involved retrieving the best of the faulty programs from the Case-Base and repairing it with the evolved fix to create a perfect solution. The result is shown for a 100% correct 3-bit multiplier program (Table 12).

0-5-6	1-5-6	2-3-6	1-4-6
1-3-6	2-5-6	2-4-6	6-9-10
0-4-6	6-8-10	0-3-6	10-14-10
9-11-7	4-9-6	13-15-7	11-20-6
15-18-10	15-19-6	17-20 -6	9-24-10
16-23-10	17-21-10	23-27-10	20-25-6
7-12-10	26-29-7	16-25-6	32
31	28	22	30
11			

Table 12. The genotype for a repaired evolved 3-bit multiplier program.

In Table 12 the dark shaded cells represent the new cells of the repair and the light shaded cells represent existing cells reused by the repair. The white cells are cells from the original program that were unchanged. In this genotype each cell contains three integers. The first two of these integers represent the input connections and the third integer 6, 7, or 10 represents the function type AND, AND NOT and Exclusive-OR, respectively. The cells containing one integer only represent output connections. An explanation of the genotype representation was given in chapters 3 and 4. The cells in the table are indexed 6 to 32, starting at the top left (index 6), numbered from left to right and top to bottom. The output cells (containing single integers) to not have index numbers. Index numbers 0 to 5 represent the indices for the inputs to the program.

5.4. An Analysis of the effects of Function choice

An examination of how the functions made available to the Cartesian Genetic Programming technique affect the evolutionary process was carried out. Expert recommended choices of gates were 6, 7 and 10. These were chosen because of their use in the conventional design, and because they simplify understanding of the evolved circuits. They were compared to selections of allowed gates. These selections were made based on the frequency occurrence of each gate. The frequencies of gates were counted over large collections of solutions, in this case, solutions for the two-bit carry adder. The frequencies of gates were not calculated simply from the number of times each atomic gate appeared in the solutions. Instead, the modularity of the evolved designs was allowed to influence the calculation. To capture this influence frequencies were counted from the number of times each gate appeared in a 2-into-1 principles e.g. Figures 31, 32 and 33. 2-into-1 principle is a general description of one of the simplest modules. 2-into-1 principles are simply two gates of any type connected into one other gate of any type. (2-into-1 principles are discussed in detail in Section 6.4 and Chapter 7). These selections (Table 13.) were made as testing all possible combinations of allowed functions is unfeasible.

Set number	Allowed functions
1 (Expert recommended functions)	6,7,10
2	10,11,13
3	6,10,11,13
4	6,8,10,11,13
5	6,8,10,11,13,14
6	6,8,9,10,11,13,14
7	6,7,8,9,10,11,13,14
8	6,7,8,9,10,11,12,13,14,15

Table 13. Allowed functions in each test. Functions selected were based on the functions appearance in a frequency count of 2-into-1 principles for the 2-bit Carry Adder with allowed functions 6 to 15 inclusive.

There were several tests:

The first test was to examine the effect of allowed functions on the number of 100% solutions generated. As the number of different functions made available to the CGP increased it became easier to achieve a 100% solution, and so more 100% solutions are achieved (see Figure 20). This effect reached a plateau if 4 (sets 3 to 8) or more functions were allowed from the complete list of functions given in Table 13. This result simply illustrates the fact that some circuits are difficult to synthesise with certain allowed gates, as explained in Section 2.2 on conventional circuit design. A significant effect observed here was that the expert recommended set (set 1) of allowed gate choices gave rise to a large number of behaviourally duplicate 100% solutions. Approximately 50% were duplicates, compared to set 2 that gave approximately 10% duplicates (see Figure 21).

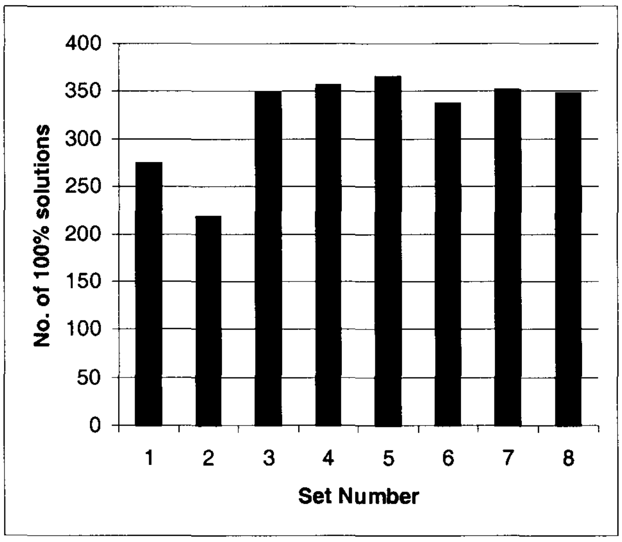


Figure 20. Graph of the number of 100% correct solutions produced by each set in Table 13.

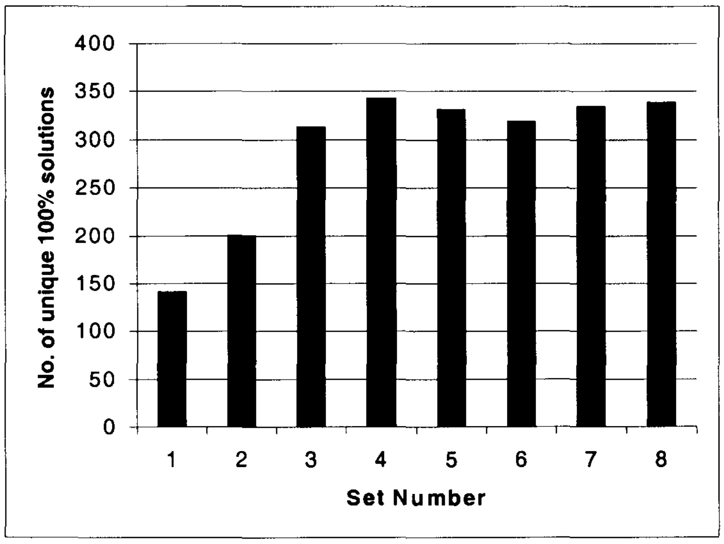


Figure 21. Graph of the number of unique 100% correct solutions produced by each set in Table 13. Unique solutions are those solutions left after behavioural duplicates have been removed.

In the other tests it was discovered that set 2 was out performed by set 1 in the average fitness of all solutions produced (see Figure 22), and set 1 also required fewer generations (see Figure 24), but sets 3 to 8 out performed sets 2 and 1. Sets 3 to 8 gave a slightly higher average fitness by approximately 1%. Sets 2 to 8 gave rise to far fewer behavioural duplicates. In general the number of generations required to achieve 100% solutions decreases as the number of allowed gates increased.

Set 1 (the expert recommended set of allowed functions) was chosen for its use in the conventional design and to ease understanding of the resulting circuit. The only significant results were that Set 1 produced many more duplicate solutions than sets 2 to 8 (see Figure 23), and that it was expected that the number of behavioural duplicates would decrease as the number of allowed functions increases, due to the greater number of possible combinations. However this is not the case. This is possibly because the amount of duplication created by sets 2 to 8 (See Figure 23) is not high enough to show this effect. So, overall, Set 1 favours the study of digital arithmetic circuits, as they are easier to understand and compare to conventional designs and it also gives rise to a smaller range of solutions. These results suggest that Set 1 produces a smaller search space; further examination is required to determine such a result.

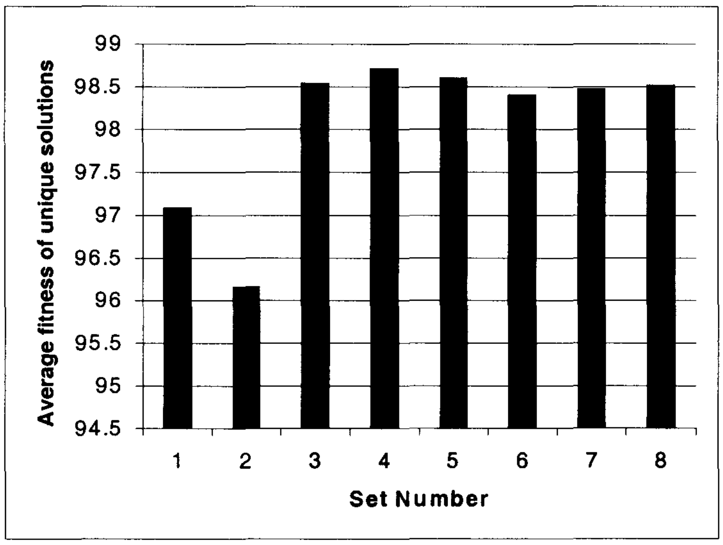


Figure 22. Graph of the average fitness of the number of unique 100% correct solutions, produced by each set in Table 13.

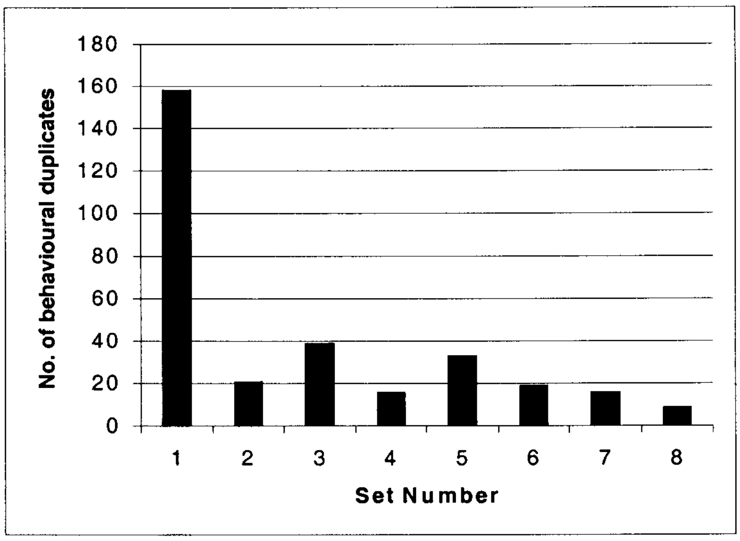


Figure 23. Graph of the number of 100% correct solutions that were behavioural duplicates, produced by each set in Table 13.

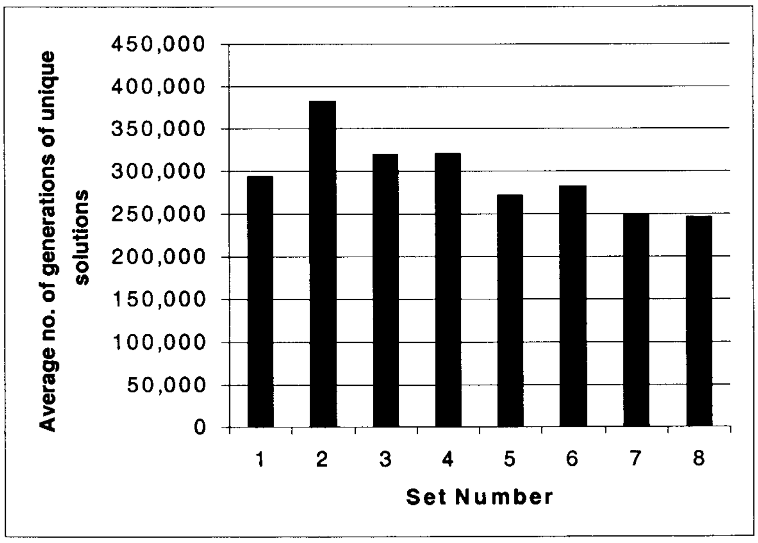


Figure 24. The average number of generations used by unique solutions, using each set in Table 13.

5.5. Seeding

Seeding is an approach to CGP where instead of using a randomly initialised population at the beginning of the execution of the CGP, a predetermined program is given and the CGP attempts to improve upon this existing program. This seed program can be a partial solution e.g. known optimal program parts for the given problem, or it could be a complete program of similar but not identical function. A particularly interesting method of seed selection is to seed the CGP with a known conventional solution to the given problem.

5.5.1. Seeding with a conventional solution

An effective way of obtaining the optimum solution for a circuit is to use a solution created using conventional techniques to seed the genotypes for the CGP. To generate an optimal circuit for a given problem, e.g. 3x3 multiplication, the CGP can be seeded with a conventional design for a 3x3-bit multiplier. This means that the CGP does not have to evolve a solution from scratch and can simply optimise the conventional solution. Solutions produced by this technique cannot be differentiated from purely evolved solutions.

The CGP technique could be used to re-map an existing solution for a current hardware platform to a new hardware platform. An existing solution could also be optimised for size or speed.

The conventional seed technique allows CGP to examine the entire search space (Figure 4). In this thesis the only limitation to the search is that only improvements in size improve the fitness (Since the conventional seed is 100% functionally correct). In this way the space of 100% correct solutions that contains the conventional solution can be specifically searched, looking for more optimal circuits. Evolving large circuits from scratch rapidly becomes impossible as their size increases, so this is one effective method of obtaining evolved solutions of increasing size with which the study of the principles of scale can be further explored.

A 4-bit multiplier has been evolved from the conventional design. It consists of 57 two-input logic gates and is 10.93% more efficient (in terms of the number of two input gates used, see the FGPA cells in Table 10) than the most efficient known conventional design (64 two-input logic gates). This evolved design can be seen in Figure A 2.5.

In some cases it is known that some parts of a seed solution cannot be optimised further. For this reason it is much more efficient if the CGP is not permitted to attempt to optimise these parts of the seed. One technique to prevent this is 'Lock-down'.

5.5.2. Lock-Down

It is possible to reduce the amount of work done by CGP by fixing the position, inputs and fitness of cells of the genotype to a configuration that is known in advance to be optimal. This is referred to as 'lockdown' and avoids CGP wasting valuable processing time. Lockdown is a technique that improves efficiency and is used in addition to seeding. With Lockdown, one or more of the cells of the seed chromosome are 'locked down' or fixed, so that the CGP does not attempt to mutate these cells. This means that these cells never change during the execution of the CGP. This speeds up the processes of the CGP as time is not spent selecting these cells for mutation or on rejecting a chromosome where one of these cells has be detrimentally mutated producing a less fit chromosome (Miller *et al.*, 1997).

The lockdown technique is very effective as larger circuits have larger areas that have obvious lockdown potential. A technique similar to lockdown was first shown by Miller *et al.* (1997). Miller pre-calculated the truth table into a form where the products of inputs were already assumed. This pre-calculation of the behaviour of the locked cells can further speed up the process. In this case the evaluation function does not have to recalculate the behaviour of the locked cells. This leads to a significant speedup. This specific method (Miller *et al.*, 1997) cannot be applied to the general seeding technique, as it requires that the cells to be locked down start with the first cell in the program and are in a contiguous unbroken block. However, this is convenient, as the cells that are most obvious candidates for lockdown are those at the beginning of the program. In the case of multiplier problems these cells are nearly always AND gates giving the products of inputs. It can be observed from the statistics on input triples that this is nearly always the case in evolved solutions and conventional solutions. Further to this it is apparent, in the case of multiplier programs that P0 and P1 do not have more optimal representations than those shown in Figure 15 b do, and so these too can be locked-down.

To examine the effects of lockdown an experiment was conducted. Firstly, one hundred 4x3-bit multipliers were evolved from scratch (Table 14.) and secondly, one hundred 4x3-bit multipliers were evolved using lockdown of ten cells (Table 15.). The ten cells locked-down in the seed used here were taken from an evolved 3x3-bit multiplier. They were the nine AND gates connected to program inputs and one additional cell required for producing output P1. An evolved 3x3-bit multiplier was used to obtain the seed for this experiment to show the potential for reusing existing evolved material to overcome the scaling problem.

It can be seen in Tables 14 & 15 that lockdown increases the number of 100% functional solutions. Lockdown also gives more optimal solutions (with respect to the number of two-input logic gates used). A locked-down program on average uses 38 cells whereas without lockdown the average number of cells used is 43 in this case. The number of 'generations' is defined to be the number of generations taken by an evolving solution to get to a given functionality-based fitness score, until 100% functionality is achieved, then increases in the fitness score reflect reductions in the number of cells used. The number of generations required to reach 100% functionality remains almost unchanged, but evolving using lockdown wastes no generations evolving or changing the already optimal 'locked down' cells. This means that it takes fewer generations to evolve a 100% functionally correct program so a greater number of generations can be spent optimising the size of the program. More generations are reported by 'all solutions' without lockdown as it takes a greater number of generations to converge towards a 100% solution.

100 chromosomes (structural) reduced to 100 chromosomes
There were 0 structural duplicates
There are 9 100% fit chromosomes
The average fitness is: 99.2556% standard deviation: 0.447214
The average number of all solution generations is 24,780,863 with a standard deviation of 5,305
The average number of 100% solution generations is 25,681,579 standard deviation of 21,555
100 chromosomes (behavioural) reduced to 100 chromosomes
There were 0 behavioural duplicates
There are 9 100% fit chromosomes
The average fitness is: 99.2556% standard deviation: 0.447214
The average number of all solution generations is 24,780,863 with a standard deviation of 5,305
The average number of 100% solution generations is 25,681,579 standard deviation of 21,555
The smallest number of gates used was 42 by chromosome #24

Table 14. Statistics for 100 4x3-bit multipliers evolved without using Lockdown.

100 chromosomes (structural) reduced to 100 chromosomes
There were 0 structural duplicates
There are 23 100% fit chromosomes
The average fitness is: 99.625% standard deviation: 0.2
The average number of all solution generations is 23,346,202 with a standard deviation of 5,321
The average number of 100% solution generations is 25,487,594 with a standard deviation of 1,558
100 chromosomes (behavioural) reduced to 100 chromosomes
There were 0 behavioural duplicates
There are 23 100% fit chromosomes
The average fitness is: 99.625% standard deviation: 0.2
The average number of all solution generations is 23,346,202 with a standard deviation of 5,321
The average number of 100% solution generations is 25,487,594 with a standard deviation of 1,558
The smallest number of gates used was 37 by chromosome #25

Table 15. Statistics for 100 4x3-bit multipliers evolved using Lockdown.

The increase in computing power required to evolve multipliers of increasing scale can be seen in Figure 25. Figure 26 shows the number of generations required to evolve multiplier circuits of increasing scale.

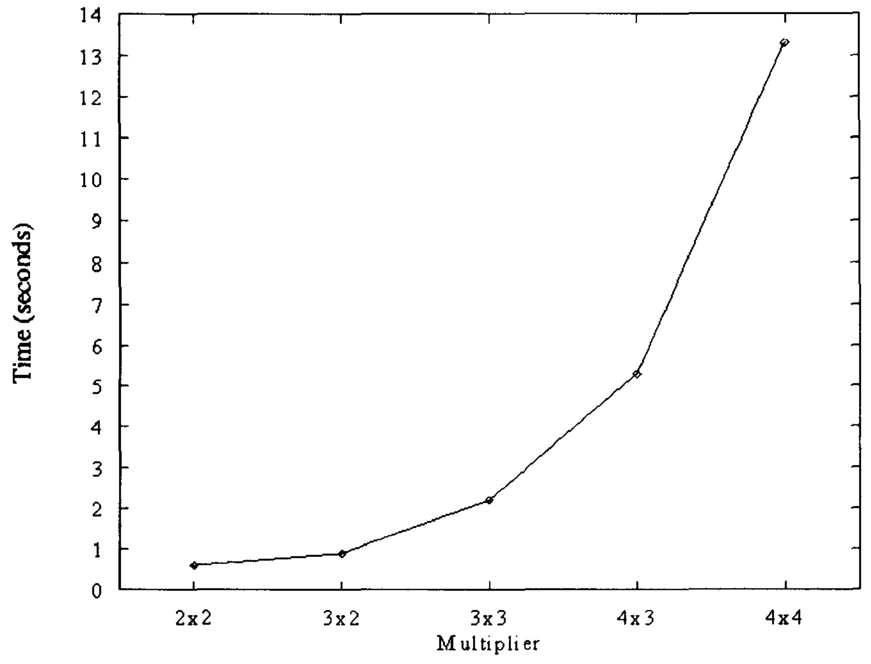


Figure 25. Time taken by a Pentium 200MHz computer to perform 10,000 generations with a population of five elements for various multiplier circuits.

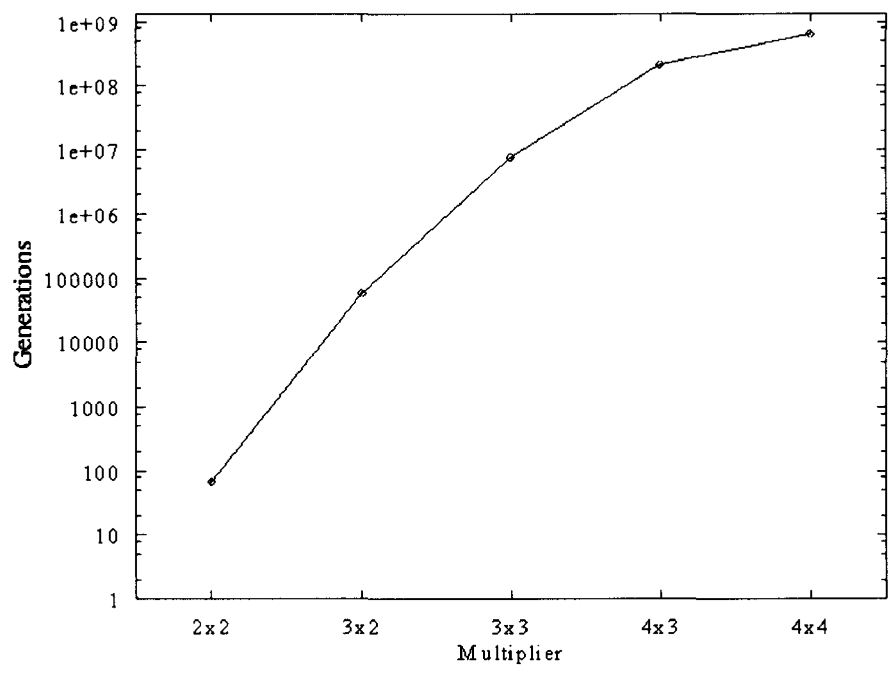


Figure 26. Number of generations required to evolve multiplier circuits of increasing scale.

Lock-Down is also useful for producing pairs of increasingly large circuits that can be matched. Seeding the CGP with the smaller of the two required circuits and locking-down the complete circuit ensures that the resulting larger circuit contains the smaller circuit. This enables the evolution of expansions for the multiplier circuits.

However, locking down a complete small circuit (e.g. a 2x2-bit multiplier) to use as a seed for a larger circuit (e.g. 3x2-bit multiplier) results in very inefficient results for the larger circuit. This is because parts of the smaller circuit need to be replaced by sub-circuits with additional functionality in order to produce efficient results. This is explained in detail in Chapter 7.

5.6. Identifying Principles in Evolving Circuits

The study of evolutionary design of digital circuits involves the examination of *products* and the *processes*. The *processes* can be considered as a search on a fitness landscape. The next section gives a brief outline of the current research into the processes involved in circuit evolution landscapes, and the following sections give a detailed analysis of the products produced by this process.

5.6.1. Fitness landscapes

Miller *et al.* (2000b) showed that circuit evolution landscapes are quite different from many recently studied landscapes. The notion of a fitness landscape is an important concept in evolutionary computation. The metaphor is taken from biology and it expresses the idea that Evolution can be considered as a population flow on a surface in which the altitude of a point qualifies how well the corresponding organism is adapted to an environment. In addition to this Miller *et al.* (2000b) examined the role of neutrality and the importance of its role in the evolutionary search was progressed. The difference originates in the structure of the genotypes which are defined by internal connections, functions and outputs, not just one alphabet, but three (Miller *et al.*, 2000b; Vassilev *et al.*, 2000). This gives rise to complicated relationships between the genes within the genotype which makes the study of the landscapes much more convoluted.

In Vassilev *et al.* (1999a and 1999b) a model for studying the structure of circuit evolution landscapes was introduced. The model is employed to investigate the structure of circuit evolution landscapes in terms of the interplay between smoothness, ruggedness and neutrality. The smoothness and ruggedness are related to the fitness differences between neighbouring points whereas the neutrality refers to the flat landscape areas (Stadler, 1996; Reidys and Stadler, 1998). The study of the characteristics of these landscapes is an important concern in digital circuit Evolution both for their scalability and in the importance of choosing appropriate sets of logic functions used in the assembly of the digital circuits. The research (Vassilev *et al.*, 1999a and 1999b, 2000) concentrates on landscapes associated with five digital circuits, a two-bit multiplier (Figure 15 a), two three-bit multipliers (Figures 16 and 17), and two four-bit parity functions which are evolved by evolutionary algorithms. The interplay of the landscape smoothness, ruggedness and neutrality is studied by an information analysis based on that given by Vassilev (1997b). It is shown that the digital circuit Evolution landscapes are characterised by vast and sharply differentiated landscape plateaux. It is also shown that the continuity of these landscapes depends on the scale and the set of logic functions used in the assembly of digital circuits.

It is beyond the scope of this thesis to give a complete description of this examination of fitness landscapes. Further details can be found in Vassilev *et al.* (2000); Miller *et al.* (2000b).

5.6.2. A Problem of Scale and a possible Solution

In the design process it has long been accepted that the best way to solve a problem is to decompose the problem into several simpler sub-problems and solve these sub-problems. One difficulty with evolving digital circuit programs is that it is computationally expensive, particularly for larger programs. Since more complex functions and larger numbers of inputs require exponentially larger circuits to produce a solution there is a limit to the size and complexity of a circuit program that can be evolved. This is referred to as the scaling problem.

Chapters 6 and 7 describe efforts to overcome the scaling problem. The approach attempts to decompose the solution programs produced by the evolutionary algorithm. This involves extracting meaningful sub-programs, or design principles, from the evolved solutions, and using them to try to solve the scaling problem and also to help in understanding the way the evolved solutions work.

Principle extraction and reuse is achieved by integration of Evolutionary Computation and CBR techniques. This section discusses the features of evolved programs that will facilitate creation of a Case-Base that will allow for adaptation and reuse of evolved Binary Cartesian Genetic programs, and the sub-programs within these programs, to create larger programs at a reasonable computational expense.

It has been shown in Section 4.2.2 that arithmetic adder and multiplier circuits are modular in construction and so are useful functions to study and refine techniques of principle extraction. Modularity by definition allows very large systems to be constructed by connecting modules together. It is clear that as multiplication is a process of repeated addition, multiplication circuits can be built by using AND gates to perform elementary one-bit multiplication and then binary full-adders connected in an arrangement called a cellular array. When biologically inspired algorithms such as evolutionary algorithms are allowed to design the building blocks and assemble the parts an amazing number of potentially new designs may be created. The fundamental question (TFQ) stated in Section 2.3, in one instance was positively answered by Miller *et al.* (1997) where it was shown that the principle of the ripple-carry adder could be inferred by studying evolved designs for one-bit and two-bit adders. This process of data-mining from evolved solutions potentially allows a complete cycle of principle extraction (Figure 2). The extracted principles by making recommendations as to useful components and sub-structures may feed back into the evolutionary algorithm.

These essential sub-structures when collected together and subjected to analysis might lead to the discovery of a new principle. Chapter 7 discusses a "finger printing" technique applicable to the genotype discussed previously that reveals the type and frequency of embedded sub-structures. Initial examination of the principle extraction problem showed that by using this finger printing technique it was possible to find known human principles, and additionally find hitherto unknown principles.

5.7. Summary

It was proposed in this chapter that through a combination of EC and CBR that it may be possible to extract generalisable principles of design from evolved solutions and use them to overcome the limitations of EC. The problems involved in repairing failed evolved solutions were discussed, showing why repair may not be a practical solution to the scaling problem. An analysis of the effects of function choice made available to the EC was given, showing that the expert recommended set of allowed functions favours the study of digital arithmetic circuits.

Three methods, seeding, seeding with the conventional solution and lock-down were seen to reduce the scaling problem shown in Figures 25 and 26. A potential solution to the scaling problem through the combination of EC and CBR was then discussed.

Chapter 6 covers a potential solution to the problems in the reuse of evolved circuit programs. The methods that are used to process the evolved programs to create a Case-Base are described in Section 6.2. Chapter 7 describes how CBR is a suitable technique for the automatic identification of principles. In chapter 7 results of the experiments and their analyses are presented.

6. CBR as a potential solution to the reuse problem

This chapter examines the initial problems involved in reusing FPGA programs that have been evolved using CGP. Large collections of FPGA programs have been evolved, each collection being a collection of FPGA programs that solve a specific problem e.g. the 3×3-bit multiplication problem, the 3-bit carry adder problem. The reuse of these collections of solutions to solve new and larger problems is not a trivial task. A significant problem here is the refinement and understanding of the unrefined and undocumented CGP generated FPGA programs. This chapter shows how the unrefined data can be refined and documented in an automated way, to build a useful Case-Base.

A Case-Base is built to enable the implementation of a CBR system. This chapter shows that CBR can be used as a 'principle' identification technology. The notion of 'a principle' is defined in this research as being any knowledge that is generally applicable to at least one design problem, such as the 2-bit carry adder problem. Two examples of principles would be: the 1-bit carry adder that can be ripple-chained to produce a 2-bit carry adder (Figure 3); and the '2-into-1' example (Figure 28), a common building module seen in evolved multiplier circuits, discussed in the next section.

6.1. CBR as a Principle Identification Technology

One potentially suitable solution to the scaling problem is to find a way to reuse evolved Cartesian Genetic Programming programs using CBR. CBR is an artificial intelligence technique that is designed to reuse past experiences to solve new problems. It can provide answers to problems in poorly understood complex domains; it does not require a domain model or rules; and it can provide an explanation of its own reasoning.

CBR can provide selection, retrieval and adaptation of old software solutions to solve new problems and it has been successfully used as a reuse system for retrieving and adapting software artefacts (Maguire *et al.*, 1995; Smyth, 1996). Case-Based reasoning has already been successfully applied to the understanding of evolutionary produced designs (Hunt, 1995; Maher *et al.*, 1996). This suggests that CBR could be used for understanding, retrieving and adapting evolutionary designs, to solve new problems. CBR also provides a scalable approach, and can be used to create designs larger than the designs that make up its source material (its Case-Base). CBR provides data mining, indexing, matching, retrieval and adaptation, and these techniques should assist the process of principle identification and application.

CBR can partly address the problem of scaling up evolved digital circuit programs. The scaling problem might be overcome by effective reuse of principles contained in the evolved programs. Identifying these principles is however a very complex task. CBR relies upon Cases that have known structure, e.g. attribute value pairs. Since evolved programs lack any “understanding" incorporated in their structure, all knowledge beyond their functionality must be identified before a useful Case-Base can be built. These principles might be able to be recombined and adapted to create new designs for new scaled-up problems.

Evolutionary algorithms have been successfully used as a “knowledge lean" method to generate knowledge for a Case-Base in earlier research (Hunt, 1995). This thesis differs significantly from previous work as the phenotypes (programs in this work), used in previous evolutionary algorithms, have had clearly defined components that make generation of a Case Base simple. It was shown that an evolutionary algorithm is the only general method for producing efficient solutions.

This evolutionary design approach raises several questions:

1. What knowledge exists within the evolved programs that may be of use?
2. How can this knowledge be automatically identified and utilised?
3. How can this knowledge be reused?

To answer these questions the following approach was taken. In human designs small programs are designed, and then linked together to make larger programs. For this reason collections of evolved programs were examined to see if such principle sub-programs could be identified (small reusable program blocks) with methods for assembling them into larger programs. This facilitates understanding of how the evolved programs work. In general these principles may consist of small sub-programs that have been extensively used throughout a large number of programs of different functionality, and the methods for assembling them into larger programs. An example of an identified sub-program that is used in a larger program is shown in Figure 27.

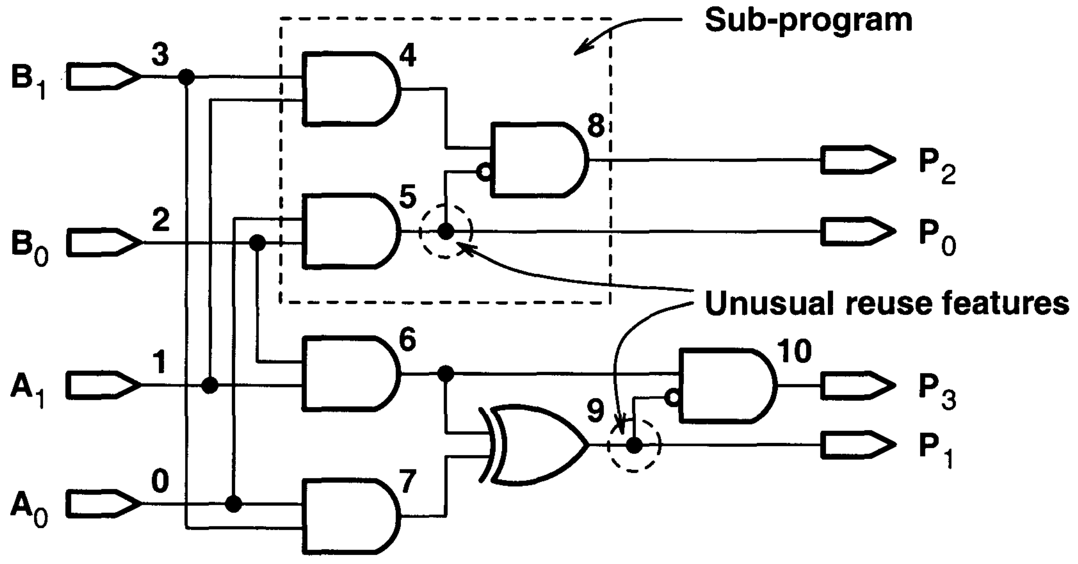


Figure 27. A novel sub-program and two unusual features of reuse in the evolved two-bit multiplier. The labels from 0 to 10 refer to the connection points in the corresponding CGP program.

Each principle contains knowledge pertaining to a particular sub-program. Collections of principles form the Cases in the Case-Base. Case-Based retrieval is then used to retrieve appropriate principles based on specified requirements. Suitable adaptation techniques such as those developed by Hanney (1996), Giraud-Carrier (1996) could be applied to build larger and more complex programs that are too computationally expensive to be evolved. Since the required functionality of programs can be specified as a truth table, the sub-programs that make up the complete design obtained by Case-Based Reasoning can be tested automatically.

An example of an identified sub-program that can be reused to create larger programs is shown in Figure 28. It can be seen in Figure 27 that gates 6, 7 and 9 are the same sub-program as the "2 into 1 example" shown in Figure 28, that has been used to form part of the larger program.

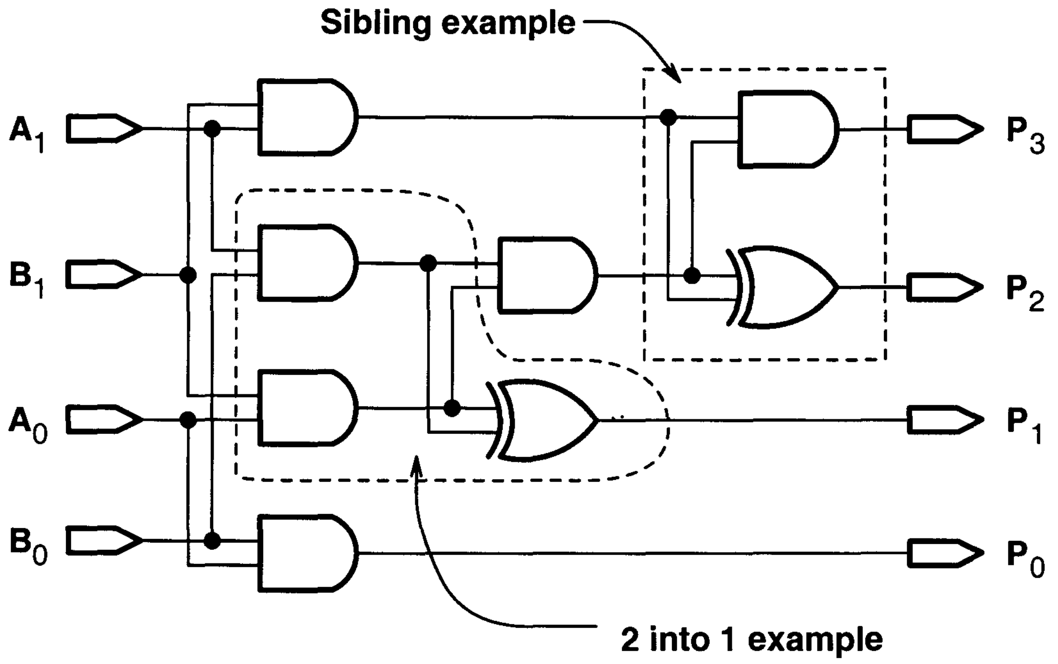


Figure 28. Two examples of sub-program format shown here in the conventional 2-bit multiplier. The pla file for the 2-bit multiplier is given in the appendix.

Repair of faulty solutions can be achieved by Case-Based substitution. The parts of the solution program where error(s) have occurred are identified and replaced with error free substitutes from other Cases that do not display the errors. In a similar manner CBR may be used to optimise the evolved programs produced for specific purposes, e.g. routing, speed, size.

6.2. Automatic Creation of a Case-Base for Reuse

In this research, BCG programs were evolved from randomised starting populations and then processed to create a Case-Base. The best solution from each run of the CGP is added to the raw Case-Base. These raw programs are then processed to make a useful Case-Base. The following experiments do not examine the evolutionary technique itself, but investigate the solutions produced by the technique. The pre-processing is achieved in several stages.

The preliminary stages involve refinement of the data to make the applications of CBR functions like matching and retrieval viable. These stages are: evaluation; removal of imperfect solutions; removal of duplicates; removal of redundant information; compression; normalisation; reduction and refinement rules; test evaluation; calculation of behaviour; removal of behavioural duplicates; statistical measurements to aid user understanding. A detailed description of each stage follows.

6.2.1. Evaluation

The first stage, evaluation is the same as the fitness function used by the CGP used to evolve the circuits (Section 4.1.2.).

6.2.2. Remove imperfect solutions

Chromosomes (Cases) with less than 100% fitness are deleted for several reasons. The main reason is to speed up the subsequent processing. It is also possible that they are of no value to the problem solving process as they always contain less useful information than a perfect solution. This is made apparent when examining repair of these imperfect solutions as shown in Section 5.3. Imperfect solutions could be useful in Cases where no perfect solutions have been found for a given problem, as they may represent partial solutions, and could be combined with each other or with seeding and lockdown techniques to provide a perfect solution.

In the case of multiplier programs it can be seen on analysis of the ‘output chains’, Section 6.6, that one output is significantly more difficult to synthesise than all of the other outputs. This makes it probable that most, if not all, flawed solutions contain a flaw in the same output chain, in similar areas. This would mean that all flawed solutions have the same or similar flaw and could not therefore be used in combination to create a flawless program. This may not be the case for other problem classes, e.g. the Sbox problem (see Appendix 2).

Imperfect solutions may contain interesting information about the evolution of digital circuits but a complete analysis of them would require an extensive amount of research too great to cover in this thesis.

6.2.3. Removal of duplicate programs

The CGP runs can produce structurally identical programs. These duplicates are removed to speed up the pre-processing and the CBR cycle. Although duplicates are of no use to the CBR cycle, it may be useful to know the frequency of occurrence of each structure found.

This information can be useful when examining the search space. The frequency data could also be useful to guide the CBR system into 'easier' territory, the hypothesis being that structures that are common are easier for the CGP to find and therefore may be easier to adapt, as the required adaptation may be easier to find. This notion of 'Fishing in a well-stocked pond', is a subject for future work. In order to compare the structures of the EC produced programs they must be processed to ensure that structurally identical programs have identical chromosomes (genotypes). To ensure this conformance several processes are required. Redundant cells are emptied, then the remaining cells are compressed and normalised. The empty cells left at the end of the chromosomes after these processes are not removed. This facilitates matching of chromosomes, as they are the same size, even when the programs they describe are different sizes. These processes are shown in Figure 29 and explained in greater detail next. These steps simplify comparison of programs.

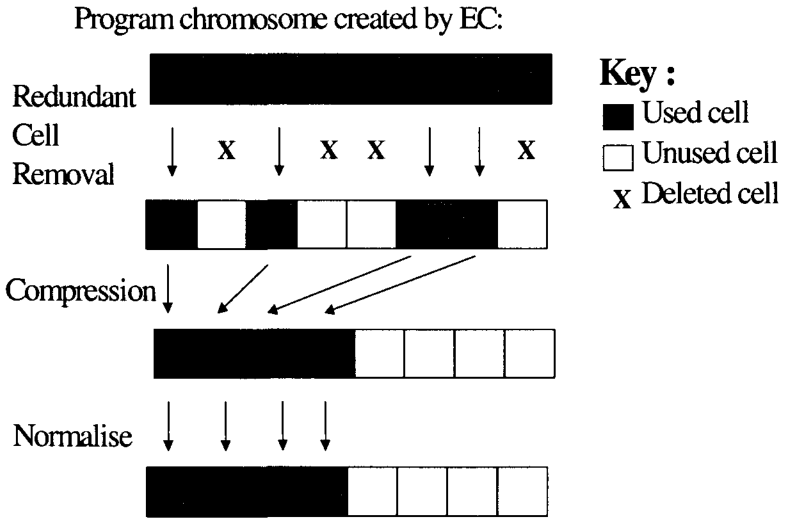


Figure 29. Refining the chromosomes.

6.2.4. Removal of redundant information

This stage involves the removal of redundant information left over from the evolutionary process. Here redundant cells that are not connected to the rest of the program are removed. This is followed by compression and normalisation of the remaining programs to facilitate the CBR functions of matching, retrieval and adaptation.

6.2.5. Compression

Compression involves removal of the spaces left in the genotype after redundant information has been removed, see Figure 29. This is done to increase the likelihood that chromosomes describing identical phenotypes will have identical chromosomes. This facilitates the removal of duplicates and the matching processes. For this same reason the chromosomes are also normalised, described next.

6.2.6. Normalisation

Normalisation reorders differently ordered cells within a genotype with the same function into a standard form. Example:

Given the cell:

9	8	6

As 6 represents the AND function, it is obvious that the two inputs 9 and 8 can be reordered as:

8	9	6

CGP could produce either of these cells, but they are identical in their function.

Some functions e.g. 'a AND NOT b' (function 7) cannot have their inputs rearranged in this manner unless the cells referred to by the indices, 8 and 9 in this case, are themselves swapped over in their index positions.

6.2.7. Reduce and refine (remove inverter pairs)

It is possible for further reductions to be made. Firstly a cell may have identical inputs e.g. the cell:

8	8	6

In this case the cell can be completely removed from the program and the connections of the relevant cells can be reconnected directly to cell 8.

Next, inverter pairs may occur. This could happen, for example, when a NAND- (NOT a AND NOT B) gate is the second input to a type 7 gate (a AND NOT b). In this instance the NOTs cancel out and the NAND gate can be replaced with an AND gate, and the type 7 gate replaced with a type 6 gate (a AND b). This can only be done if no other cells reference the NAND gate as an input, unless that cell itself has an inversion on the relevant input, and then that inverter can also be removed. A full list of rules governing the removal of inverter pairs is given in Table 16.

If the function is	Then
11 and input A is from type 15	change 15 to 6 and 11 to 10
11 and input A is from type 9	change 9 to 12 and 11 to 10
7 and input B is from type 11	change 11 to 10 and 7 to 6
7 and input B is from type 9	change to 12 and 7 to 6
7 and input B is from type 15	change 15 to 6 and 7 to 6
14 and input B is from type 11	change 14 to 12 and 11 to 10
14 and input B is from type 9	change 14 to 12 and 9 to 12
14 and input B is from type 15	change 14 to 12 and 15 to 6

Table 16. The table of rules used to remove inverter pairs. These changes are only applied if the fitness of the subject program is not affected. The numbers refer to the function types given in Table 10.

6.2.8. Test evaluation

The primary evaluation procedure used by the CBR is a modified extension to the evaluation procedure used by the CGP. The modification is that this evaluation uses 1-bit arithmetic, not the 32-bit approach explained in Section 4.1.2. The extensions are the calculation of behaviour and the recording of errors found. The error recording procedure uses the 1-bit instead of 32-bit approach to simplify error recording. When using the 32-bit approach the information regarding the position of the errors within the 32-bit word is not immediately available.

6.2.9. Calculate Behaviour

The behaviour of a BCG program is represented by the binary outputs of every cell for the given function. In Table 17 the structure (genotype) can be seen for an example of a 3x2-bit multiplier, and the function and behaviour of this circuit can be seen in Table 18.

0 4 1 6	2 3 2 6	2 4 1 6	1 3 6 6	0 3 6 6	1 4 5 6
8 7 6 7	5 11 0 10	11 5 6 7	5 8 0 6	6 10 0 10	8 13 13 10
9 16 2 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	14	17	12	15
7					

Table 17. Structural knowledge in a program Case for a 13 gate 3x2-bit multiplier, evolved using gates 6, 7 and 10 (see Table 10.). This representation follows that shown in Figure 7, but also shows the additional 3rd input used in Miller's representation. Note that even though the 3rd inputs are shown, none of the gate types used in this example use the third input, as only the MUX cell type (see Table 10) uses three inputs. As there are three inputs for each of these cells, the fourth number in each cell represents the gate type.

Inputs a2 a1 a0 b1 b0	Behaviour of genotype from cell index 5 to cell index 24	Outputs p4 p3 p2 p1 p0
00000	00000000000000000000	00000
00001	00000000000000000000	00000
00010	00000000000000000000	00000
00011	00000000000000000000	00000
00100	00000000000000000000	00000
00101	00100000000000000000	00001
00110	01000000001000000000	00010
00111	01100000001000000000	00011
01000	00000000000000000000	00000
01001	00000100001000000000	00010
01010	00010011100000000000	00100
01011	00010111101000000000	00110
01100	00000000000000000000	00000
01101	00100100001000000000	00011
01110	01010011101000000000	00110
01111	01110100000110000000	01001
10000	00000000000000000000	00000
10001	10000001000000000000	00100
10010	00001000000010000000	01000
10011	10001001000010000000	01100
10100	00000000000000000000	00000
10101	10100001000000000000	00101
10110	01001000001010000000	01010
10111	11101001001010000000	01111
11000	00000000000000000000	00000
11001	10000101001000000000	00110
11010	00011011100010000000	01100
11011	10011110011100000000	10010
11100	00000000000000000000	00000
11101	10100101001000000000	00111
11110	01011011101010000000	01110
11111	11111101010100000000	10101

Table 18. Binary representation of inputs outputs and behaviour for the 3x2-bit multiplier program shown in Table 17. The behaviour column beneath cell index 5 gives the behaviour for the first cell in the genotype in Table 17.

Due to the difficulty in interpreting large binary tables like that shown in Table 18, the columns are compressed into 8-bit base 10 integers, see Table 19. Each *row* in the 8-bit compressed representation represents a *column* of 4x8 bits from the binary representation. The first six rows (in the 8-bit compressed representation) represent the inputs to the program, and the last six rows are the outputs of the program. The rows in-between represent the output or Behaviour of each cell in the program.

32-bit or 16-bit compression could be used instead of 8-bit compression, but these representations are not as easy to interpret as base ten 8-bit integers. This facilitates human understanding of the behaviour, and gives a speed up in processing over the binary case, as explained in Section 4.12.

Inputs				
a2	0	0	255	255
a1	0	255	0	255
a0	240	240	240	240
b1	204	204	204	204
b0	170	170	170	170

Cell No.	Behaviour			
5	0	0	170	170
6	192	192	192	192
7	160	160	160	160
8	0	204	0	204
9	0	0	204	204
0	0	170	0	170
11	0	76	0	76
12	0	76	170	230
13	0	76	0	68
14	0	0	0	136
15	192	106	192	106
16	0	128	0	136
17	0	128	204	68
18	0	0	0	0
19	0	0	0	0
20	0	0	0	0
21	0	0	0	0
22	0	0	0	0
23	0	0	0	0
24	0	0	0	0

Outputs				
p4	0	0	0	136
p3	0	128	204	68
p2	0	76	170	230
p1	192	106	192	106
p0	160	160	160	160

Table 19. 8-bit compressed base 10 integer representation of behaviour shown in binary in Table 18. Each column in the binary representation is compressed into 8 bit integers shown in the rows above.

6.2.10. Remove Behavioural Duplicates

This stage of processing removes programs that have identical behaviour. If two programs are said to be behavioural duplicates it means that there is a one to one correspondence between behaviour rows (in the 8-bit view), although the rows may be in a different order.

The behavioural match is a more powerful matching function than the structural match as it can find identical behaviours where structures are different. This because behaviours capture the context in which structures are used i.e. the binary states of a structure for a given set of inputs.

6.2.11. Behavioural reduction

This stage examines the behaviour of the circuit to see if any two gates in the same program have the same behaviour, and then deletes the surplus one. This is made unlikely with the additional fitness function given in Section 4.1.2 that optimises the size of the circuit. The additional fitness function makes duplicate behaviour contribute to a poorer fitness value.

6.2.12. Statistics

The statistical measures (see Figure 30) can be invoked at any point during the pre-processing and CBR cycle to show the changes in the information in the Case-Base made by the different processing algorithms. The basic measures are: the number of perfect solutions in a test sample; the average number of generations required to produce a) all perfect solutions b) all solutions, and their standard deviations; the average fitness and standard deviation.

500 programs (structural) reduced to 500 programs
There were 0 structural duplicates
There are 435 100% fit programs
The average fitness is: 99.915% standard deviation: 0.0632455
The average number of all solution generations is 102973 with a standard deviation of 1713
The average number of 100% solution generations is 113912 with a standard deviation of 2071
500 programs (behavioural) reduced to 152 programs
There were 348 behavioural duplicates
There are 129 100% fit programs
The average fitness is: 99.8931% standard deviation: 0.114708
The average number of all solution generations is 106920 with a standard deviation of 4009
The average number of 100% solution generations is 120910 with a standard deviation of 5057
The smallest number of gates used was 13 by program #0

Figure 30. An example of basic measurements made on a Case-Base of 500 3x2-bit multipliers.

6.3. Smallest number of gates used

This stage simply counts the number of gates used by each circuit and can deliver a top-ten of smallest circuits in a Case-Base. This is done on a first found basis, e.g. if there are more than 10 equivalent sizes the first found are reported, see Figure 30 above. This is useful for locating individual programs for inspection.

6.4. Two into One Principles

The frequencies of specific instances of the two into one principle, counted over collections of solutions to problems, show modularity and common structures. These have been divided into two main types, input 2-into-1 and internal 2-into-1. Input 2-into-1 are those 2-into-1 principles that consist of two inputs to the program and one gate. Internal 2-into-1 principles are those that consist of two gates feeding into 1 gate, as shown in Figures 31 & 28.

6.5. Sibling principles

'Sibling principle' is the name given to two gates that share the same inputs, (see example in Figure 28).

6.6. Extract chains

Chains are the sub-programs that show the parts of the program that are responsible for an individual output only. These are useful for showing which outputs are most difficult to synthesise. This information could lead to a more effective evolutionary search being defined for a specific problem or problem class.

6.7. The Case and indexing

A program Case consists of function; structure; behaviour; fitness; frequency of occurrence (of the individual circuit), the number of gates used, and the number of generations taken when the last improvement occurred. A full example of a Case is given in Appendix 6.

The programs in the Case-Base can be indexed by any of their attributes. This facilitates experimentation. The primary indices are the program function and fitness. There are additional indices that could be defined e.g. length (max. number of gates in series), depth (max. number of gates in parallel), These additional indices would be useful for optimising various costs e.g. speed, surface area used etc, see Section 4.2.1.5. An example of a Case is given in Appendix 6 - an example Case.

The indexing mechanism used to index cases in the case-base is a case-based index (Kolodner, 1993). In this thesis each program is part of a Case in the Case-Base. Each Case stores its own information pertaining to its similarity to all other Cases. This information relates to the following four indexes. The functional index uses the function type that each program was designed for e.g. 3x2-bit multiplier. The structural index uses the structure of the programs themselves e.g. see Table 17. The behavioural index uses the behaviour of each program e.g. see example Table 19. Matching of Cases is achieved using the Nearest Neighbour Matching function that gives the ranking of Cases (Kolodner, 1993). In this way the index needs only be calculated once, and additional Cases can be indexed in linear time proportional to the number of Cases in the base. Adaptation Guided Retrieval (AGR) (Smyth, B., 1996) is used when retrieving candidate Cases for adaptation. Using AGR means that the Case that can best be adapted to produce a solution is retrieved. In conventional retrieval the Case that most closely matches the problem definition is retrieved, regardless of what adaptation knowledge is available. In the approach presented in this thesis AGR is used to retrieve a Case that can be adapted by existing adaptation knowledge, to produce a solution.

6.8. Summary

In this chapter CBR was discussed as a principle identification technology. CBR can partly address the problems of scale through identification, reuse and adaptation of existing evolved solutions to solve new problems. The notion of the 'principle' or small reusable program block, and some of unusual features of evolved solutions were discussed.

Next the automatic creation of a case-base was shown. Through the processes of evaluation, deletion of imperfect solutions and duplicate programs, removal of redundant information, compression, normalisation and refinement the evolved solutions are prepared for the case-base. Next, the behaviour of the programs is calculated and statistics of the case-base are derived. The 'two-into-one' and 'sibling' principles are then counted. The case structure and indexing mechanism were discussed.

Chapter 7 discusses experiments that were done to identify reusable principles, showing that the evolved solutions are modular in nature and that it is possible to extract and reuse information contained in the evolved solutions.

7. Experiments to Identify Reusable Sub-programs

This Chapter examines methods for identifying sub-programs that can be reused to build larger programs that may be more efficient than any conventionally designed alternative.

In these experiments two types of arithmetic multipliers were examined: the 3x2-bit multiplier (3 bits by 2 bits) and the three-bit multiplier. Trying to extract principles by studying two and three-bit multipliers is a difficult problem because the 3x3-bit multiplier is considerably more complicated. The 3x2-bit multiplier provides a useful bridge between these two circuits and thus may assist attempts to find scalable principles of construction. Experiments were carried out to produce 50 perfect solutions for both the 3x2 and three-bit multipliers. The maximum number of cells allowed was equal to the number required in the most efficient conventional designs. In the case of the 3x2-bit multiplier 15 two-input gates are needed, while the three-bit multiplier requires 30. Two different sets of gates were used. The experimental set-up was as follows:

Three-bit multiplier population size 5, mutation 3 genes on average, gates 6; 7; 10, geometry 1 × 30, levels-back 30.

Three-bit multiplier population size 5, mutation 3 genes on average, gates 6 - 15, geometry 1 × 30, levels-back 30.

3x2-bit multiplier population size 5, mutation 3 genes on average, gates 6 - 15, geometry 1 × 15, levels-back 15.

MUX gates were not allowed in these experiments as they do not generally occur in the conventional circuits and also they make the comparison to the conventional circuits much more difficult.

All of the programs examined were processed to make a basic Case-Base as outlined in Section 6.2. These experiments firstly involved an examination of sub-programs that are directly connected to inputs and secondly the examination of those that are not. Figure 31 shows the connections between the gates in the 2-in to-1 sub-programs. When the frequencies of the 2-into-1 sub-programs in a program are calculated a "fingerprint" for that program is defined. Fingerprints of evolved programs differ from those of human designed programs. These differences are shown in Figures 32, 33 and 34. One identified sub-program is shown in Figure 28 (2-in to-1 example). Sub-programs like the "sibling example", also shown in Figure 28, are the subject of future work.

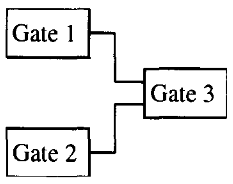


Figure 31. The 2-into-1 sub-program layout.

The results of the experiment showed that the input sub-programs (the inputs are connected to the primary program inputs) closely followed the human design. In the programs studied the products of pairs of inputs are calculated. These products (the AND-gates on the left-hand side of the Figures that are directly connected to the inputs) can be seen in the examples of the two-bit multiplier for the evolutionary design and the human design shown in Figures 15 a and 15 b, respectively.

The evolutionary designs are markedly different from the conventional designs in the way that they reuse parts of the circuit, for example, the unusual reuse of a product of two low significant bits directly in the output of a high significance product. These unusual features are shown in Figure 27. It can be seen that the output P 0 is reused to generate P 2, whereas in Figure 28 (the conventional human design) P 0 is not used in any other part of the circuit.

Figure 32 shows the frequencies of 2-in to-1 sub-programs, that are not connected directly to inputs, for 50 three-bit multiplier circuits, with expert recommendations for available gate functions 6, 7, and 10 that were intended to promote elegant solutions. The 'Sub-program type' numbers each of the 2-into-1 sub-program instances in order with the highest frequency first. The first six bars represent sub-programs: 6-6-10, 6-10-10, 6-10-6, 6-6-6, 6-6-7, and 6-7-10. The first bar in Figure 32 is the frequency of sub-program "2-in to-1 example" in Figure 28 and it is used significantly more than the other sub-programs. The fifth bar in Figure 32 is the frequency of the sub-program shown in Figure 27. The sub-programs represented by the first four bars in Figure 32 are common in conventional designs. The sub-program shown in Figure 27 is novel and is not used in the conventional human design. It is clear that much of the evolutionary two-bit multiplier can be reused to build a three-bit multiplier. This implies that the larger solutions tend to contain the same sub-programs as the smaller solutions.

Figure 33 shows the frequencies of the 2-in to-1 sub-programs, that are not connected directly to inputs, for 50 three-bit multiplier circuits using gates 6 to 15. In this experiment no assumptions were made about suitable gate functions. It was hoped that the experiment would reveal the fundamental building blocks of the multiplier circuits. There were 309 different 2-in to-1 sub-program types. The six most frequent sub-programs were 6-6-10, 6-15-11, 15-15-10, 6-6-7, 15-6-11, and 15-6-13. It can be seen that the dominant sub-program is once again 6-6-10. This confirms that the conventional structures are most common. It is noteworthy that the gate function 15 occurs very often. This was unexpected as the use of a NAND gate is not familiar in conventional multiplier design.

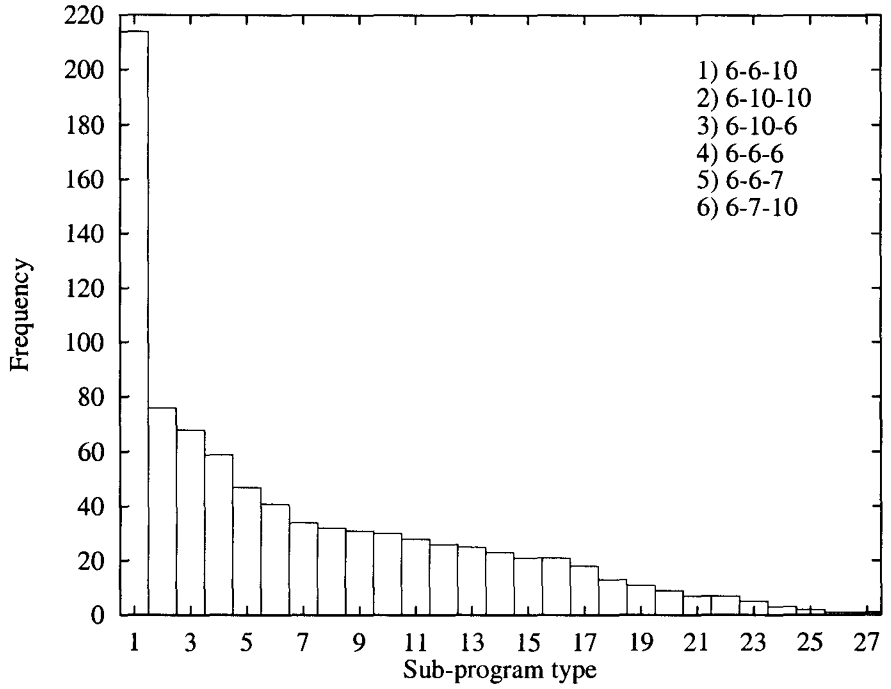


Figure 32. The frequencies of 2-in to-1 sub-programs, that are not connected directly to inputs, counted for fifty three-bit multiplier circuits with expert recommendations for available gate functions 6, 7, and 10. The six most frequent sub-programs are listed.

The graphs in Figures 32, 33 and 34 show how common or rare the different sub-programs are. The 'sub-program type' in each of the figures is simply an identification number.

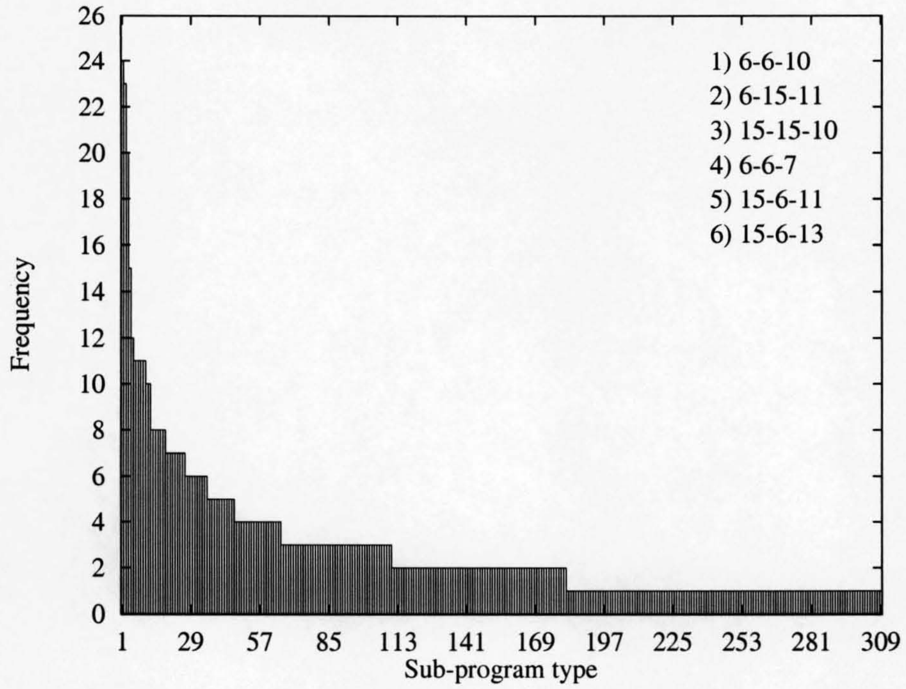


Figure 33. The frequencies of 2-in to-1 sub-programs, that are not connected directly to inputs, for 50 three-bit multiplier circuits without expert recommendations for gate functions, using gate types 6 to 15. The six most frequent sub-programs are listed.

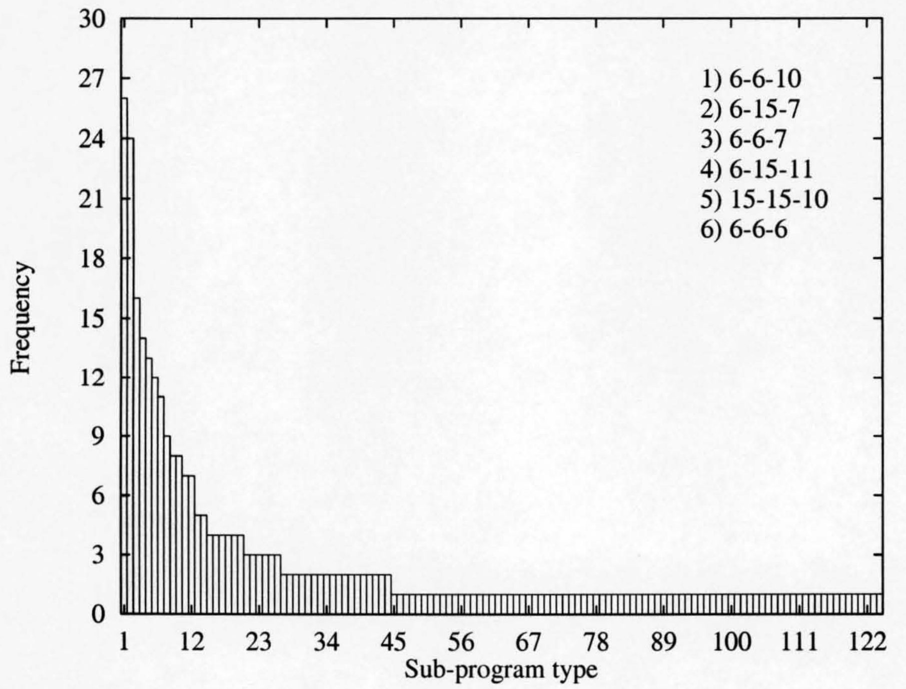


Figure 34. The frequencies of 2-in to-1 sub-programs, that are not connected directly to inputs, for 50 3×2-bit multiplier circuits without expert recommendations for gate functions 6 to 15. The six most frequent sub-programs are listed.

In Figure 34 a total of 124 sub-programs were identified. The first six were 6-6-10, 6-15-7, 6-6-7, 6-15-11, 15-15-10, and 6-6-6. It should be noted that taken in isolation some of the sub-programs are equivalent. For instance, 6-6-10 and 15-15-10 could be considered to be logically identical however it may be that some of the internal connections are reused in another part of the circuit. Once again, the figure shows that 6-6-10 is dominant. At this stage it is not known which sub-programs are responsible for the efficiency of these designs. For instance, the sub-program 6-6-7 is here the third most frequently occurring and the fifth and the fourth in the previous two experiments, respectively.

It has been seen that all multipliers largely use the same type of sub-programs. This shows that they could potentially be reused by a CBR system to design multiplier programs with more than three-bit multiplication.

Preliminary experiments showed that the results were dependent on decisions made in evolving the programs using the EC. These decisions include the geometry of the FPGA program, the gates made available to the EC and the human knowledge used in specifying the requirements. Initial experiments allowed a wide variety of gates to be used by the EC. The CBR system showed that the EC only required a limited set of these gates for optimum performance.

The aim of the experiments in the second investigation was to examine the evolved programs and show that these share some common sub-programs that can be used to build larger programs. Two different sub-programs were identified: those that are directly connected to inputs, and those for which this is not the case. The frequencies of occurrences of these two types can differ significantly. It is necessary to examine very specific types of sub-programs to avoid the combinatorial explosion of enumerating all possible sub-programs.

7.1. Comparison of circuits of different sizes

This experiment involves the scaled matching of programs of different sizes, from the same class e.g. multiplication. When a match has been found between two programs of different sizes the difference between the programs is identified.

This identification of difference could be repeated for subsequently increasing sizes of program pairs. Then these differences would be examined to see if a general principle could be discerned in the increasingly large adaptations.

To compare programs of different sizes the behaviour of a larger program is examined to see if the smaller program's behaviour exists as part of the larger program i.e. the behaviour of the larger program includes all of the behaviour of the smaller program. When two such programs are found, the identified difference between the two programs is the adaptation that needs to be applied to a program of the smaller size to make it equivalent to the larger one.

The smaller program is easier to evolve and after adaptation it is equivalent to the larger program, therefore greatly reducing the time required to produce one of the larger programs. If the extensions themselves could be extended then automatic scaling could be achieved. If automatic scaling could be achieved a general principle for scaling that class could possibly be derived.

Two experiments were conducted:

- The first experiment derives an adaptation (extension) from a 3x2-bit multiplier and a 3x3-bit multiplier (two solutions of the same class and a contiguous scale increase). This adaptation is referred to as adaptation 1. Two solutions were retrieved from the case-base that met the requirement that all of the behaviour of the smaller program (3x2) was part of the behaviour of the larger program (3x3). This is referred to as a 'scaled behavioural match'. An adaptation is then defined as being the sub-programs that must be added to the 3x2-bit multiplier solution to make it behaviourally equivalent to the 3x3-bit multiplier program. Then adaptation 1 (Appendix 3) was applied to the same 3x2-bit multiplier to show that the adaptation contains all of the necessary information to adapt a 3x2-bit multiplier to a 3x3-bit multiplier.

- The aim of the second experiment was to show that adaptation 1 could be applied to a 3x2-bit multiplier that was not a scaled behavioural match with the 3x3-bit multiplier from which adaptation 1 was derived, to obtain a new 3x3-bit multiplier.

Example: How to compare two circuits of different sizes:

The smaller circuit's truth table (See the 3x2-bit multiplier in Table 20.) is half the size (in terms of the length of the rows) of the next larger circuit's truth table (See the 3x3-bit multiplier in Table 21).

On examination of the behaviours of these two sizes of programs it can be seen in Tables 20 and 21 (shaded areas) that the 3x2, if it were a scaled behavioural match to a 3x3, would have a match for each of its behaviour rows in the first half of the behaviour rows of the 3x3. The second halves of the 3x3 behaviour rows may or may not match. The rows that do match are referred to as symmetric and asymmetric. A 'symmetric' behaviour row is a row in the larger of the two programs that is identical about the center point of the behaviour column (see Table 21 index 11). An 'asymmetric' behaviour row is one where the second half of the row is not identical to the first half, but the first half does match with a behaviour row in the smaller program (see Table 21 index 17). Those that don't match are referred to as 'new' (see Table 21 index 10). The term 'new' means that this behaviour did not exist or partially exist (asymmetric) in the original 3x2 program. For example a 3x2 matches the following 3x3. The letters on the left-hand side in Table 21 are annotations made by the matching process (s = symmetric, n = new, a = asymmetric). The 3x3 has one more input than the 3x2, shown in Table 21 as '3x3 only'.

This behavioural matching process also shows how to adapt the smaller program. The symmetrical matched rows should be preserved in the subject of the adaptation (a new 3x2). The rows in the smaller program that match an asymmetric row in the larger one can be deleted, in the subject of adaptation, as they will be replaced by new cells that give the new asymmetric behaviour. Symmetric and asymmetric are identified by and necessary for the matching process. Rows that exist only in the 3x3 behaviour are new and have to be inserted by the adaptation.

If a Case-Base of 3x2-bit multipliers is matched against a Case-Base of 3x3-bit multipliers several scaled behavioural matches are found (see Appendix 4). For example a 3x2 matches the following 3x3. The letters on the left-hand side are annotations made by the matching process (s = symmetric, n = new, a = asymmetric).

Index	Behaviour (converted from binary into 8 cells of 8-bit integers)			
0	0	0	255	255
1	0	255	0	255
2	240	240	240	240
3	204	204	204	204
4	170	170	170	170

5 (o 0)	s	160	160	160	160
6	s	0	0	170	170
7	s	0	204	0	204
8	s	0	0	204	204
9	s	0	76	0	76
10	a	0	76	0	68
11	a	0	128	0	136
12 (o 2)	a	0	76	170	230
13	s	0	170	0	170
14	s	192	192	192	192
15 (o 4)	a	0	0	0	136
16 (o 1)	s	192	106	192	106
17 (o 3)	a	0	128	204	68

Table 20. An annotated behaviour table for a 3×2-bit multiplier. The FPGA program that produced the behaviour in this table can be seen in Table 17.

In Tables 20 and 21, each behaviour box gives an 8-bit integer in decimal, each row in Table 20 contains 4 such boxes and each row in Table 21 contains 8. Each row represents the behaviour of one cell or gate in the program. 8-bit base ten integers are used instead of binary as a visual aid. The indices in brackets e.g. (o 0) to (o 4) or (o 5) show the cells that are used for outputs from the program. The shaded areas in each table represent the behaviour of the 3×2-bit multiplier. The first parts of each table (20 and 21) show the inputs to the multiplier programs and are considered to be a special part of the behaviour. The inputs have been separated from the behaviour of the program itself for clarity.

Index	Behaviour (converted from binary into 8 cells of 8-bit integers)							
0	0	0	255	255	0	0	255	255
1	0	255	0	255	0	255	0	255
2	240	240	240	240	240	240	240	240
3 (3×3 only)	0	0	0	0	255	255	255	255
4	204	204	204	204	204	204	204	204
5	170	170	170	170	170	170	170	170

6 (o 0)	s	160	160	160	160	160	160	160	160
7	n	0	0	0	0	0	255	0	255
8	n	0	0	0	0	0	0	255	255
9	n	0	0	0	0	0	0	160	160
10	n	0	0	0	0	0	0	128	128
11	s	0	0	170	170	0	0	170	170
12	n	0	0	0	0	240	240	240	240
13	s	0	204	0	204	0	204	0	204
14	a	0	204	0	204	0	204	160	108
15	s	0	0	204	204	0	0	204	204
16	s	0	76	0	76	0	76	0	76
17	a	0	0	170	170	240	240	90	90
18	n	0	0	0	0	0	204	0	108
19	a	0	76	0	68	0	12	0	4
20	a	0	128	0	136	0	192	160	104
21	n	0	0	0	0	0	0	0	108
22 (o 2)	a	0	76	170	230	240	188	90	22
23	s	0	170	0	170	0	170	0	170
24	s	192	192	192	192	192	192	192	192
25	a	0	0	204	204	0	255	204	51
26	a	0	0	0	136	0	192	128	32
27 (o 5)	n	0	0	0	0	0	0	128	236
28	a	0	0	0	136	0	192	127	223
29 (o 4)	a	0	0	0	136	0	192	127	147
30 (o 1)	s	192	106	192	106	192	106	192	106
31 (o 3)	a	0	128	204	68	0	63	108	91

Table 21. An annotated behaviour table for a 3×3-bit multiplier.

It can be seen that the shaded behaviour rows of the 3×2-bit multiplier shown in Table 20 have matching behaviour rows in the 3×3-bit multiplier (shown in Table 21). The 3×3-bit multiplier (Table 21) has some behaviour rows, e.g. rows 13 and 14 that both behave in the same way as row 7 in the 3×2-bit multiplier (Table 20), but it row 14 is different to row 13 in the second half of its behaviour row. Row 14 is asymmetric where as row 13 is symmetric around the center for the row.

7.2. Extracting differences and expansions

By extracting the differences of two different sized circuits of the same class the knowledge of how to adapt a smaller circuit into a larger is derived.

To do this the annotated Table 21 is used. The transformation is built by identifying each gate type and its behavioural context. This is referred to as a Behavioural Context Triple (BCT), an example follows (see Figure 35). This is in keeping with the concept of the 2-

into-1 principle, presented earlier. This BCT is an 'in context' 2 into 1 principle (see Figure 31).

Starting at the top of the behaviour table (after the input lines) in Table 21, the first line of program behaviour is at index 6. The annotation here is 's' for symmetric, so this line already exists in the 3x2 circuit and is not a difference between the two programs. The corresponding line in the 3x2 that is to be adapted, is preserved. Next at index 7, the annotation is 'n' for new, so this line is a difference between the two circuits and must be added to the adaptation. Looking at the genotype containing this cell, the second cell in Figure A3.1, it can be seen that the two inputs to this cell are inputs of indices 1 and 3. To create a Behaviour Context Triple (BCT) the behaviour of each of these inputs is taken, then the function type of the cell and the behaviour of the output (New output) of the cell is recorded. The BCTs are the components of an adaptation.

Input	Behaviour							
1	0	255	0	255	0	255	0	255
3	0	0	0	0	255	255	255	255

Input 1 AND Input 3 produces:

New output 1

n	0	0	0	0	0	255	0	255

Figure 35. A Behavioural Context Triple (BCT), showing two behavioural inputs, a function and a Behavioural output.

This recording process is repeated for all asymmetric and new cells. This process gives a collection of BCTs that constitute an adaptation that can be used to adapt a 3x2-bit multiplier to a 3x3-bit multiplier.

In order to adapt a 3x2-bit multiplier to a 3x3-bit multiplier it is necessary to renumber the cells of the program to take the extra input into account. This is a simple task as this program is the same as the original program and it does not use the additional input. For this reason it should be noted that the 'unused input' 3x2 is symmetrical in its behavioural rows when expanded to use the 3x3-bit multiplier truth table. It is necessary to expand the 3x2 to the 3x3 truth table in order to apply the adaptation to it.

These BCTs, together forming an adaptation, can now be applied to another 3x2-bit multiplier program. This is done by matching the required input behaviours of each BCT in the adaptation, that are not available as outputs from other BCTs in the adaptation, to existing behaviours in the new 3x2-bit multiplier to be adapted to a 3x3-bit multiplier. This is how Adaptation Guided Retrieval is realised. Each BCT is applied to the 3x2 under adaptation in the order that they were derived (starting at the first cell in the genotype). The outputs of each BCT become part of the behaviour of the partially adapted 3x2-bit multiplier as each new cell is added, thereby producing a new program adapted from a 3x2-bit multiplier using the adaptation.

The adaptation could also be applied to any program containing the necessary behaviour of the 3x2, e.g. a signal processing program where an old program needs to be scaled-up or re-engineered to meet a new specification.

A full example is given in Appendix 3. This example shows the extracted adaptation applied to the 3x2 that the transformation was partly derived from. This was done to show that all of the necessary information was contained in the transformation.

The next example in Section 7.4 shows the adaptation applied to a different 3x2 program that was not a behavioural match for the 3x3 or the 3x2 from which the adaptation was derived. The Adaptation shown in Table 24, was successful showing that the adaptation can be applied to other programs.

Note: The observation that some cells of the 3x2 have to be replaced in the 3x3 shows that it is not appropriate to 'lockdown' (Section 5.5.2) an entire 3x2 to speed up the evolution of a 3x3. This would result in an inefficient solution.

7.3. Identifying a principle

By identifying the differences of extracted differences from circuits of increasing size a principle for scaling up circuits in a given class could be derived. An alternative to this could be to generate a schema hierarchy as given by Louis (1993) where the top schema on each tree is the most general description of a given cluster of differences. Schemas become increasingly specialised the lower in the hierarchy they appear until each schema represents a specific Case in the Case-Base. An investigation of the potential of this schema approach to principle extraction and scaling is the subject of future work.

7.4. Applying an extracted principle.

This stage uses the extracted principle to expand an existing evolved solution to a circuit with a greater number of inputs.

0 4 1 6	2 3 2 6	2 4 1 6	1 3 6 6	0 3 6 6	1 4 5 6
8 7 6 7	5 11 0 10	11 5 6 7	5 8 0 6	6 10 0 10	8 13 13 10
9 16 2 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	14	17	12	15
7					

A reproduction of Table 17, shown for clarity.

Take the 3x2-bit multiplier from the example in Table 17. and renumber the cells to take the extra input into account:

0 $\underline{5}$ 0 6	2 $\underline{4}$ 0 6	2 $\underline{5}$ 0 6	1 $\underline{4}$ 0 6	0 $\underline{4}$ 0 6	1 $\underline{5}$ 0 6
$\underline{9}$ $\underline{8}$ 0 7	$\underline{6}$ $\underline{12}$ 0 10	$\underline{12}$ $\underline{6}$ 0 7	$\underline{6}$ $\underline{9}$ 0 6	$\underline{7}$ $\underline{11}$ 0 10	$\underline{9}$ $\underline{14}$ 0 10
$\underline{10}$ $\underline{17}$ 0 10	0 0 0 –1	0 0 0 –1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	0	$\underline{15}$	$\underline{18}$	$\underline{13}$
$\underline{16}$	$\underline{8}$				

Table 22. The 3x2-bit multiplier from the behaviour example Table 19. with indices renumbered to take the extra input of the target 3x3-bit multiplier into account. The third inputs in each cell have been set to zero, as all of the gates in this example are two input gates. A new output is added and set to zero, to take the extra output of the target 3x3-bit multiplier into account.

The numbers with underscores are the input connections of each cell that have been increased by 1 to make room for the new input. As the 3x3 has one more output than the 3x2, an additional output is also added (p5) in Table 23 with a default index setting of 0. The rows marked p0 to p4 are the outputs of the original 3x2 and correspond directly to the output indices (15, 18, 13, 16, 8) given in the above structure. Then using the 3x3-bit multiplier PLA instead of the 3x2-bit multiplier PLA the following behaviour is produced:

Index		Behaviour							
0	(p5 -default)	0	0	255	255	0	0	255	255
1		0	255	0	255	0	255	0	255
2		240	240	240	240	240	240	240	240
3	3x3only	0	0	0	0	255	255	255	255
4		204	204	204	204	204	204	204	204
5		170	170	170	170	170	170	170	170
6	s	0	0	170	170	0	0	170	170
7	s	192	192	192	192	192	192	192	192
8	s (p0)	160	160	160	160	160	160	160	160
9	s	0	204	0	204	0	204	0	204
10	s	0	0	204	204	0	0	204	204
11	s	0	170	0	170	0	170	0	170
12	s	0	76	0	76	0	76	0	76
13	s (p2)	0	76	170	230	0	76	170	230
14	s	0	76	0	68	0	76	0	68
15	s (p4)	0	0	0	136	0	0	0	136
16	s (p1)	192	106	192	106	192	106	192	106
17	s	0	128	0	136	0	128	0	136
18	s (p3)	0	128	204	68	0	128	204	68
19	Redundant	0	0	0	0	0	0	0	0

20	Redundant	0	0	0	0	0	0	0	0
21	Redundant	0	0	0	0	0	0	0	0
22	Redundant	0	0	0	0	0	0	0	0
23	Redundant	0	0	0	0	0	0	0	0
24	Redundant	0	0	0	0	0	0	0	0
25	Redundant	0	0	0	0	0	0	0	0

Table 23. The behaviour for a 3×2-bit multiplier with an extra input required by a 3×3-bit multiplier. This prepares the 3×2 for expansion to the 3×3. The rows marked as 'Redundant' are not used in the program, Table 22.

It can be seen that the 3×2 is symmetrical about the mid-points of its behavioural rows when expanded to use the 3×3 PLA file. This is expected as this program is the same as the original 3×2 program and it does not use the additional input. It is necessary to expand the 3×2 to the 3×3 PLA file in order to apply an expansion to it.

Next the adaptation is applied to this prepared 3×2. The relevant behaviour required by each BCT in the adaptation is found in turn. As the inputs to each BCT are found the gate or cell for that BCT is inserted into the expanding 3×2 program. This produces the 3×3-bit multiplier program (genotype) shown in Table 24. The circuit (phenotype) is shown in Figure 36.

0 5 0 6	2 4 0 6	2 5 0 6	1 4 0 6	0 4 0 6	1 5 0 6
9 8 0 7	1 3 0 6	0 3 0 6	8 14 0 6	7 11 0 10	4 15 0 6
2 3 0 6	15 9 0 10	6 18 0 10	13 19 0 6	12 20 0 7	19 22 0 10
14 21 0 6	12 20 0 10	13 10 0 10	23 26 0 6	17 24 0 10	14 27 0 10
29 24 0 7	23 26 0 10	28	30	31	25
16	8				

Table 24. The new 3×3–bit multiplier after adaptation from the 3×2-bit multiplier from the behaviour example Table 19. The white cells have not been changed during adaptation. The light shaded cells represent new cells inserted by the adaptation. The dark shaded cells represent cells that have been replaced by the adaptation.

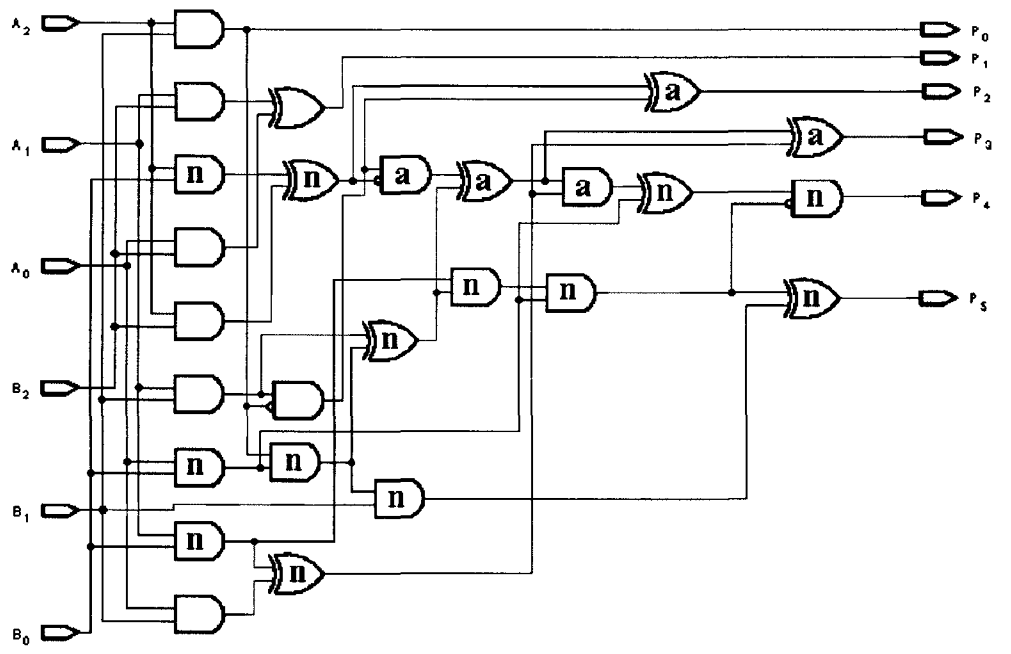


Figure 36. An example of a 3x2-bit multiplier that has been adapted (scaled up) to a 3x3-bit multiplier program.

Figure 36 shows an example of an adapted program. The gates containing no letter are unchanged from the original 3x2-bit multiplier. The gates containing an 'a' are the asymmetric gates that have the behaviour of the 3x2-bit multiplier, but had to be replaced to give the full functionality required for the 3x3-bit multiplier. The gates containing 'n' are gates that are new in this adapted program.

7.5. The Wee Ken evolved FPGA program reuse system

The CGP-CBR software system that supports this research, known as 'Wee Ken', has been implemented. Wee Ken was implemented in C++ using the Borland 4.5 C++ compiler in Windows 95. The CBR components of Wee Ken are original code by the author. The CGP software used in this thesis to evolve the FPGA programs was adapted from the original C code developed by J. F. Miller. Wee Ken must be supplied with truth tables and parameters defining the target architectures of the digital circuit programs. In Wee Ken, all other functionality, including adaptation, is automated.

7.6. Summary

This work has shown that it is possible to automatically extract and apply principles of design in the complex domain of Binary Cartesian Genetic programming. These principles contain knowledge pertaining to structured sub-programs, and give an understanding of the evolved programs. Previous work in this area has been in domains where the solutions produced by evolutionary algorithms had definite components that can be easily made into a Case-Base (Maher *et al.*, 1996).

The work presented here addresses the difficult task of identifying components for Case building in a domain where these components are not obvious. The techniques developed are seen to produce digital circuits in the form of gate array programs that may be more efficient than their equivalent human design. The techniques developed also show how to adapt programs produced by EC, and how to learn the principles involved in EC generated digital circuit program designs, and apply them to new problems.

Further analysis will involve examining different types of sub-program format from that shown in Figure 31, for example the "sibling example" in Figure 28. Interpretation becomes more difficult using a larger number of function choices, and it has been shown that the CBR system can automatically identify sub-programs that were known to human designers, and also identify novel sub-programs. This allows the CBR system to automatically suggest limited function types for the algorithms used in evolving new programs to improve performance. It may also be possible to use the CBR system to automatically seed the evolutionary algorithm, a technique proved by Maher *et al.* (1996). Their approach uses evolution to adapt an existing solution to a new problem. This approach could be used to re-engineer small changes to existing programs, possibly using a technique similar to Lockdown (See Section 5.5.2).

In the experiments reported here the maximum number of cells available to the CGP with which to evolve a program was equal to the number required to build the conventional circuit. It is already known that the larger the maximum number of gates allowed to construct the circuit the easier it becomes to evolve (Miller *et al.*, 1998a). This could imply that in these experiments the more conventional sub-programs are likely to dominate. Further experiments where the maximum number of cells available is less than the conventional should reveal whether or not conventional sub-programs also appear in more efficient evolved programs.

From the point of software reuse, this work shows that it is possible to achieve a high level of automation of software reuse in Binary Cartesian Genetic programming where precise requirements can be specified, and behaviour can be completely analysed. Future work could also examine the portability of the proposed approach to other software engineering problems.

Experiments of a similar nature to those described in this section have been carried out on two-bit adders (See Appendix 5). The analysis showed once again that there were some sub-programs that were much more frequent than the majority of the sub-programs. In these experiments recommendations for atomic components were made to promote elegant solutions. These recommendations were compared to those suggested by the frequency analysis of the sub-programs and to those suggested by the frequencies of the atomic gates. The experiments showed that the expert recommendations gave rise to high numbers of duplicate solutions in the sets. Further to this the sub-program recommended atomic gate selections gave rise to fewer duplicate solutions and a slightly higher average fitness in each set. The average number of generations required to produce a solution remained close to that of the expert recommendation based set. The expert based results and the Case-Base based results were compared to results based on the frequencies of the atomic gates. The recommendations suggested by the frequencies of the atomic gates gave lower numbers of perfect solutions, larger numbers of duplicate solutions and a lower average fitness. This suggests that the solutions with the highest fitness tend to be modular in nature. It is expected that larger problems in the same class will show greater differences between the techniques. It is also possible that these results may not hold for all problem classes, as some problem classes are not currently known to have modular solutions.

8. Conclusions

It has been shown that there are problems in software reuse. These problems are the creation and documentation of software libraries, and the identification and adaptation of software artefacts in these libraries to solve new problems.

It was seen that EC techniques could be used to create software for a library in an automated fashion. By developing a Case-Base it was shown how these libraries could be automatically documented. Through the use of CBR techniques it was shown that artefacts in these automatically generated software libraries could be automatically identified and adapted to solve new problems.

The modularity of the evolved programs was shown, and the mechanism by which these modules could be assembled to create larger programs was illustrated with the conventional techniques of logic synthesis.

It was proposed that by examining examples of increasing scale from a given problem class that principles of scale for that problem class could be identified. Methods for identifying suitable material for deriving such principles have been shown, with additional methods for extracting and applying this knowledge.

The main research problem was how to apply automatic identification, refinement and application of substitution rules to Cases with non-flat, structured problems and solutions. This involved research into developing the latest methods in CBR adaptation techniques, to apply these techniques to Cases with unstructured solutions, which was a significant advance over the application of techniques like these to Cases with simple numerical atomic solutions.

The main contribution of this thesis was in the area of automatic extraction and application of principles in the complex domain of software reuse in FPGA programming. This means applying and enhancing the ideas given by Hanney (1996), and by Smyth (1996) to structured CBR Cases with complex interacting components with no obvious modules.

The EC techniques shown were able to produce digital circuits in the form of FPGA programs that were more efficient than their equivalent human design, in a design area where there is very limited design knowledge.

The techniques developed show how to refine programs produced by EC, and how to learn principles involved in digital circuit design, and apply them to new problems.

It was argued here in this thesis that a much larger space of possible designs can be explored by employing an evolutionary algorithm together with a process of assembling and test. This has been demonstrated in the case of digital circuit program design, in particular, arithmetic circuits.

This thesis has examined some fundamental questions concerning the role of evolutionary algorithms as a novel methodology for design. It has tried to indicate a possible answer to the question: Can new principles of design be discovered by using a simulation of some of the processes of evolution? This leads on to the further question: In which type of design problems is it most likely that new principles might be discovered? Clearly since the search space of all possible designs is enormously enlarged compared with traditional rule-based methods, an extremely fast fitness function and a large amount of computation effort are required. It should be anticipated that tens of millions, even billions, of genotypes would have to be examined. Digital circuit design is an ideal candidate for novel principle extraction. The fitness function simply uses the bit wise operations that CPUs were designed for. For example, on a 450MHz PC one can evaluate 50,000 designs for a three-bit multiplier per second (in a 1×30 geometry). In spite of the extraordinary speed of fitness evaluation it is time consuming to evolve correct three-bit multiplier circuits. About 50 million genotypes need to be examined to achieve a high probability of success. Thus it becomes essential to understand more about the nature of the fitness landscapes. This work has been undertaken in Miller *et al.* (2000b) and Vassilev *et al.* (2000). Even with a computer that could deliver large numbers of correct designs the problem of data mining the evolved circuits to extract principles still exists. It is not feasible for an expert to study and compare hundreds of unconventional designs. An automated approach to this problem, using techniques of CBR, was shown.

In Section 2.4 it was shown how an evolutionary algorithm together with a process of assembling and testing could be used to produce novel and efficient designs for digital arithmetic circuits. One of the central ideas was to look at the possibilities of identifying new principles which would allow the construction of efficient multiplier circuits of arbitrary size by studying evolved examples of two and three-bit multipliers. In one sense this problem is ideal for artificial evolution. Firstly, this is because the evaluation process for a genotype representing a circuit is extremely fast as it relies on precisely those simple bit-level operations that modern CPUs were designed for. Secondly, the binary nature of the evolved circuits makes them relatively simple to understand. There are two ways in which it might be possible through artificial Evolution to try to build efficient large systems. One is to try to discover a general scalable principle of design. The second is to try to produce as efficient and large building blocks as possible. The work in this thesis develops an automated way of extracting sub-principles in evolved circuits. This was achieved through examination of the *products* of CGP. The second aspect concerns the *process* of CGP and the nature of the fitness landscapes associated with these digital circuits, examined by Miller *et al.* (2000b), and by Vassilev *et al.* (2000). An examination of the relationship between these two aspects shown in Figure 37, *process* and *products*, is the subject of future work. It is hoped that this will show when evolutionary techniques should be used and when CBR techniques could be applied.

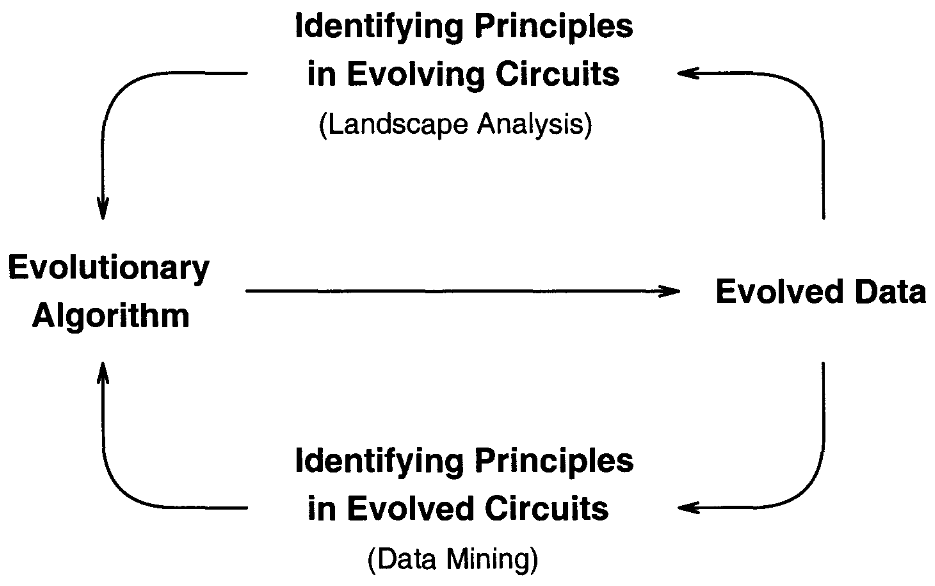


Figure 37. The principle extraction loop.

The structure of fitness landscapes has been studied in terms of their smoothness, ruggedness and neutrality by Vassilev (1997b), and by Vassilev *et al.* (2000). This has been done using an information analysis on a time series that is obtained by sampling the fitness values on a random walk. A major impediment in studying the structure of circuit Evolution landscapes is that they originate from two completely different alphabets responsible for the gate functionality and the connectivity of the evolved digital circuits. It was shown that it might be better to consider these landscapes as a product of three subspaces associated with the gate functionality, internal and output connectivity of the gate array. Hence, the genotypes sub-divide into three chromosomes with different characteristics. It has been shown that the landscapes have vast neutral areas with sharply differentiated plateaux and these in turn are related to the scale the objective function. The landscapes were found to become more continuous with increasing scale (Vassilev, 1997b; Vassilev *et al.*, 2000).

Even with a computer that could deliver large numbers of correct designs an expert would have the problem of examining all of the evolved circuits to extract principles. It has been shown that it is possible to automatically identify and apply the evolutionary design principles contained within the phenotypes. This greatly reduces the knowledge acquisition bottleneck, a primary factor in the creation of a Case-Base in any CBR system (Hanney, 1996). In previous work where evolutionary algorithms were used in conjunction with CBR the genotypes have had clearly identified modules (Maher *et al.*, 1996), making the construction of the Case-Base simple. It was shown here that by examining the frequency of occurrence of small sub-circuits (2-in to-1) that a sort of program "fingerprint" could be constructed. This not only confirms the familiar conventional principles but also reveals novel sub-circuits that are good building blocks in the evolved circuits. It was shown that the principles identified in small-scale multipliers (two-bit) match those identified in two larger scale multipliers, the two-and-a-half-bit multiplier and the three-bit multiplier. This suggests that there are principles in these multiplier circuits that may hold true for all sizes of multipliers and as such may be used to create larger scale multipliers that are beyond the reach of current evolutionary processing power.

Another important factor is that modular construction bypasses the necessity for testing of the truth table for the whole circuit, only the modules need be tested. For example the ripple carry adder principle shows that any number of one bit carry adder units may be chained together to produce a perfectly functional larger carry adder, like the conventional sixteen-bit carry adder. This reduces the problem of exhaustive testing of very large circuits. This identification of principles facilitates experts in understanding the nature of the evolutionary designed solutions. It is possible that a new carry adder principle may be discovered that produces more efficient carry adders. These processes enable the creation of a Case-Base, the foundation for a reasoning system that could be used to solve the scaling problem. This leads the way to the automation of the reasoning techniques of CBR.

8.1. Future Work

There is still a long way to go in this field. It is of interest to investigate why particular principal modules are evolved. These modules could be easier to evolve as they are more flexible e.g. more easily modified for multiple uses (more general).

Current results obtained were based on simple gate array circuit programs. Future work entails using the above techniques in the creation of programs that are too large to evolve. It is also of interest to examine the potentials for applying these techniques to programs with higher level functions, e.g. assembly language with states and loops.

Future work could examine the frequencies of the BCTs themselves, giving the modularity of the programs from a behavioural (in context) point of view. New test problems could be defined which will potentially allow simpler generalisable principles to be identified. These problems could be for example, a 4x2-bit multiplier that could then be compared to a 4x3-bit multiplier and 3x3, 3x2-bit multiplier.

Future work of interest could be to induce rules as given by Hanney (1996) by finding closely matching programs, of the same class, with small differences in fitness, and defining a rule of the form 'IF this change is made THEN this increase in fitness is achieved'. These rules could then be applied to programs of larger sizes to examine the possibility that some of these rules might be scale independent principles.

Future work will attempt to generalise identified differences from circuits of increasing size in order that a principle for scaling up circuits in a given class could be derived. Further analysis will involve examining adaptations generated by pairs of programs from different classes that have similar components, e.g. multipliers, signal processors and carry adders. It may also be possible to use EC to adapt large program solutions created by CBR, i.e. using the CBR system to seed the EC, a technique demonstrated by Hunt (1995) and demonstrated in the 'Lockdown' process in Section 5.5.2. This is using EC for the CBR sub-task of adaptation, supported by Maher *et al.* (1996).

References

Akers, S. B. (1978). Binary decision diagrams. IEEE Transactions on Computers C-27, 509-516.

Bäck, T., Hoffmeister, F. and Schwefel, H. P. (1991). A survey of evolutionary strategies. In: Belew, R. and Booker, L. (eds.), Proceedings of the 4th International Conference on Genetic Algorithms, pp. 2-9. San Francisco, CA: Morgan Kaufmann.

Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D. (1998). Genetic Programming: An Introduction. San Francisco, CA: Morgan Kaufmann.

Barletta, R. (1991). An Introduction to case-based reasoning. AI Expert.

Bergmann, R. and Wolfgang, W., (1996). PARIS: Flexible Plan Adaptation by Abstraction and Refinement. Centre for Learning Systems and Applications, University of Kaiserslautern, P.O. Box 3049, D-67653 Kaiserslautern, Germany.

Biggerstaff, T.J. (1992). An Assessment and Analysis of Software Reuse. Advances in Computers, Vol. 34.

Brace, K. S., Rudell, R. L. and Bryant, R. E. (1990). Efficient implementation of a BDD package. In: Proceedings of 27th A CM/IEEE Design Automation Conference, pp. 40-45.

Bramer, M., Liu W., Thompson, S., Dattani, I., (1996). Knowledge Discovery and Datamining. Tutorial at Expert Systems 1996. 16th Annual Conference of the British Computer Society Specialist Group on Expert Systems.

Brayton, R. K., Hachtel, G. D., McMullen, C. T. and Sangiovanni-Vincentelli, A. L. (1984). Logic Minimization Algorithms for VLSI Synthesis. MA: Kluwer Academic Publishers.

Bryant, R. (1991). On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. IEEE Transactions on Computers 40, 205-213.

Cendrowska, J. (1987). PRISM: An algorithm for Inducing modular rules. International Journal of Man-Machine Studies, 27, 349-370.

Chen, X. and Hurst, S. L. (1982). A comparison of universal-logic-module realisations and their application in the synthesis of combinatorial and sequential logic networks. IEEE Transactions on Computers C-31, 140-147.

Davio, M., Deschamps J.P. and Thayse A. (1983). Digital systems with algorithm implementation. Wiley.

Devadas, S. (1993). Comparing two-level and ordered binary decision diagram representations of logic functions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12, 722-723.

Devadas, S., Ghosh, A. and Keutzer, K. (1994). Logic Synthesis. New York: McGraw-Hill Inc.

Drechsler, R., Gockel, N. and Becker, B. (1996). Learning heuristics for OBDD minimisation by evolutionary algorithms. In: Parallel Problem Solving from Nature IV, vol. 1141 of Lecture Notes in Computer Science, pp. 730-739. Heidelberg: Springer.

Drechsler, R., Sarabi, A., Theobald, M., Becker, B. and Perkowski, M. A. (1994a). efficient representation and manipulation of switching functions based on Ordered Kronecker Functional Decision Diagrams. In: Proceedings of the Design Automation Conference, pp. 415-419.

Drechsler, R., Theobald, M. and Becker, B. (1994b). Fast OFDD based minimisation of fixed polarity reed-muller expressions. In: Proceedings of the European Design Automation Conference, pp. 2-7.

Flockton, S. and Sheehan, K. (1999). A system for intrinsic Evolution of linear and non-linear filters. In: Stoica, A., Keymeulen, D. and Lohn, J. (eds.), Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware, pp. 93-100. Los Alamitos, CA: IEEE Computer Society.

Flockton, S. J. and Sheehan, K. (1998). Intrinsic circuit Evolution using programmable analogue arrays. In: Sipper, M., Mange, D. and Perez-Uribe, A. (eds.), Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware, vol. 1478 of Lecture Notes in Computer Science, pp. 144-153. Heidelberg: Springer-Verlag.

Friedman, S. J. and Supowit, K. J. (1990). Finding the optimal variable ordering for binary decision diagrams. IEEE Transactions on Computers C-39, 710-713.

Fuchs, B., Mille, A., Chiron, B., (1996), Using Explanations to guide Adaptation. ECAI 96.

Fujita, M. and Matsunaga, Y. (1993). Variable ordering of binary decision diagrams for multilevel logic minimization. Fujitsu Scientific and Technical Journal 29, 137-145. 23

Gibbs S., Tsichritzis D., Casais E., (1990). Class Management For Software Communities. Communications of the ACM, Vol.33, No. 9.

Giraud-Carrier, C. (1996). Flare: Induction with prior knowledge. In: Proceedings of Expert Systems 1996, vol. XIII of Research and Development in Expert Systems, pp. 173-181. SGES Publications.

Grundy, D. L. (1994). A computational approach to VLSI analog design. Journal of VLSI Signal Processing 8 (1), 53-60.

Hanney, K. (1996). Learning Adaptation Rules From Cases. Technical report, Department of Computer Science, Trinity College, University of Dublin, Ireland. MSc thesis.

Hunt, J. (1995). Evolutionary Case Base Design. Progress in Case-Based Reasoning First UK workshop. Berlin Springer Verlag.

Iba, H., Iwata, M. and Higuchi, T. (1997). Machine learning approach to gate-level Evolvable Hardware. In: Higuchi, T. and Iwata, M. (eds.), Proceedings of the 1st International Conference on Evolvable Systems: From Biology to Hardware, vol. 1259 of Lecture Notes in Computer Science, pp. 327-343. Heidelberg: Springer-Verlag.

Job, D. and Shankararaman V. (2000). Evolutionary Computation with Case-Based Reasoning. In Lees B. and Shankararaman V. (eds.), Proceedings of the 5th UK Workshop on Case-Based Reasoning. To appear in Expert.

Job, D., Shankararaman V. and Miller, J.F. (1999a). Hybrid AI Techniques for Software Design. The 11th International Conference on Software Engineering and Knowledge

Engineering. pp. 315-319. Printed by Knowledge Systems Institute Graduate School, Skokie Illinois.

Job, D., Shankararaman V. and Miller, J.F. (1999b). Hybrid AI Techniques for Automated Software Reuse. International Conference on Case-Based Reasoning. Technical Report of the Centre for Learning Systems and Applications (LSA) of the University of Kaiserslautern.

Job, D., Shankararaman V. and Miller, J.F. (1999c). Combining CBR and GA for Designing FPGAs. Proceedings of the 3rd International Conference on Computational Intelligence and Multimedia Applications.

Job, D., Shankararaman V. and Cordingley, B. (1996) Using Natural Language Processing in Case-Based Reasoning Systems. Research and Development in Expert Systems XIII.P 173-181. Proceedings of Expert Systems 1996, the Sixteenth Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems, Cambridge, December 1996. SGES Publications 1996. ISBN 1 899621 13 X.

Kajitani, I., Hushino, T., Nishikawa, D., Yokoi, H., Nakaya, S., Yamauchi, T., Inuo, T., Kajihara, N., Iwata, M., Keymeulen, D. and Higuchi, T. (1998). A gate-level EHW chip: Implementing GA operations and reconfigurable hardware on a single LSI. In: Sipper, M., Mange, D. and Perez-Uribe, A. (eds.), Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware, vol. 1478 of Lecture Notes in Computer Science, pp. 1-12. Heidelberg: Springer-Verlag.

Kolodner, J.L., & Leake, D.B. (1996). A Tutorial Introduction to Case-Based Reasoning. In, Case-Based Reasoning: Experiences, Lessons, & Future Directions. AAAI Press/The MIT Press, Menlo Park, California, US.

Kolodner, J. (1993). Case-Based Reasoning. San Mateo, CA: Morgan Kaufmann.

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

Koza, J. R. (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. Cambridge, MA: MIT Press.

Krueger C.W., 1992. Software Reuse. ACM Computing Surveys, Vol. 24, No.2, June 1992.

Lala, P. K. (1996). Practical Digital Logic Design and Testing. NJ: Prentice Hall. Lee, C. Y. (1959). Representations of switching circuits by binary decision programs. Bell Systems Technical Journal 38, 985-999.

Leake, B., (1993) Learning Adaptation Strategies by Introspective Reasoning about Memory Search. Computer Science Department, Indiana University, Bloomington, IN 47405, U.S.A.

Leake, D., ed., (1996), Case-Based Reasoning: Experiences, Lessons and Future Directions. Menlo Park: AAAI Press.

Lee, C. Y., (1959) Representations of switching circuits by binary descision diagrams. Bell Systems Technical Journal vol. 38 pp. 985-999.

Lenart, M. and Pasztor, A. (1994). How much knowledge is needed? Co-Evolutionary Design. 7th International Conference on Industrial Engineering Applications of Artificial Intelligence. Gordon and Breach, Switzerland.

Louis S. J. (1993). Genetic algorithms as a computational tool for design. Doctor of Philosophy, Department of Computer Science, Indiana University.

Louis, S. McGraw, G. and Wyckoff, R. (1992). Automating Explanation of Genetic Algorithm Results (two paradigms collide). 5th Florida Intelligence Symposium. USA Florida AI Research Society. P201-5.

McEvoy, J.P. and Zarate, O. (1996). Quantum theory for Beginners. Icon books Ltd.

Maguire, P., Shankararaman, V., Szegfue, R. and Morss, L. (1995). Application of case-based reasoning to software reuse. In: Watson, I. (ed.), Progress in Case-Based Reasoning, Lecture Notes in Artificial Intelligence, pp. 165-174. Berlin: Springer- Verlag.

Maher, M. L. and de Silva Garza, A. G. (1996). The adaptation of structural systems designs using genetic algorithms. In: Information Processing in Civil and Structural Engineering Design, pp. 189-196. CIVIL-COMP Press.

McCluskey, E. (1956). Minimization of boolean functions. Bell System Technical Journal 35, 1417-1444.

Mili, H. (1995). Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering, Vol. 21, No. 6, JUNE 1995

Miller, J. F., Job, D. and Vassilev, V. K. (2000a). Principles in the evolutionary design of digital circuits, Part I. Journal of Genetic Programming and Evolvable Machines 1 (1).

Miller, J. F., Job, D. and Vassilev, V. K. (2000b). Principles in the evolutionary design of digital circuits, part II. J. Genetic Programming and Evolvable Machines 1 (2).

Miller, J. F. (1999a). An empirical study of the efficiency of learning boolean functions using a Cartesian genetic programming approach. In: Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M., Honavar, V., Jakiela, M. and Smith, R. E. (eds.), Proceedings of the 1st Genetic and evolutionary Computation Conference, vol. 2, pp. 927-936. San Francisco, CA: Morgan Kaufmann.

Miller J., T. Kalganova, N. Lipnitskaya and D. Job. (1999b). The Genetic Algorithm as a Discovery Engine: Strange Circuits and New Principles. Proc. of the AISB Symposium on Creative Evolutionary Systems (CES'99). Edinburgh, UK.

Miller, J. F. and Thomson, P. (1998a). Aspects of digital evolution: Evolvability and architecture. In: Eiben, A. E., Bäck, T., Schoenauer, M. and Schwefel, H.-P. (eds.), Parallel Problem Solving from Nature V, vol. 1498 of Lecture Notes in Computer Science, pp. 927-936. Berlin: Springer.

Miller, J. F. and Thomson, P. (1998b). Aspects of digital evolution: Geometry and learning. In: Sipper, M., Mange, D. and Perez-Uribe, A. (eds.), Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware, vol. 1478 of Lecture Notes in Computer Science, pp. 25-35. Heidelberg: Springer-Verlag.

Miller, J. F., Thomson, P. and Fogarty, T. (1997). Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In: Quagliarella, D., Periaux, J., Poloni, C. and Winter, G. (eds.), Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, pp. 105-131. Chechester, UK: Wiley.

Miller, J. F., Luchian, H., Bradbeer, P. V. G. and Barclay, P. J. (1994). Using a genetic algorithm for optimising fixed polarity reed-muller expansions of boolean functions. International Journal of Electronics 76, 601-609. 24

Motorola (1997). Motorola Semiconductor Technical Data: Advance Information Field Programmable Analog Array 20-cell Version MP AA020. Motorola Inc.

Muhlenbein, H. and Schlierkamp-Voosen, D. (1993). The science of breeding and its application to the Breeder Genetic Algorithm (BGA). evolutionary Computation 1 (4), 335-360.

Murakawa, M., Yoshizawa, S., Adachi, T., Suzuki, S., Takasuka, K., Iwata, M. and Higuchi, T. (1998). Analogue EHW chip for intermediate frequency filters. In: Sipper, M., Mange, D. and Perez-Uribe, A. (eds.), Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware, vol. 1478 of Lecture Notes in Computer Science, pp. 134-143. Heidelberg: Springer-Verlag.

Poli, R. (1997). Evolution of graph-like programs with parallel distributed genetic programming. In: Bäck, T. (ed.), Proceedings of the 7th International Conference on Genetic Algorithms, pp. 346-353. San Francisco, CA: Morgan Kaufmann.

Poli, R. (1999). Sub-machine-code GP: New results and extensions. In: Poli, R., Nordin, P., Langdon, W. B. and Fogarty, T. (eds.), Proceedings of the 2nd European Workshop on Genetic Programming, vol. 1598 of Lecture Notes in Computer Science, pp. 65-82. Heidelberg: Springer-Verlag.

Poli, R., Page, J. and Langdon, W. B. (1999). Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In: Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E. (eds.), Proceedings of the 1st Genetic and evolutionary Computation Conference, vol. 2, pp. 1162-1169. San Francisco, CA: Morgan Kaufmann.

Reidys, C. M. and Stadler, P. F. (1998). Neutrality in Fitness Landscapes. Tech. Rep. 98-10-089, Santa Fe Institute. Submitted to Appl. Math. & Comput.

Richter, M. (1998). Introduction - the basic concepts of CBR. Case-Based Reasoning Technology: from foundations to applications.

Quine, W. (1952). The problem of simplifying truth functions. American Mathematical Monthly 59, 521-531.

Sasao, T. (1993). Logic Synthesis and Optimization. MA: Kluwer Academic Publishers.

Scherr, W. (2000). Personal communication from Wolfgang Scherr. Senior Engineer, Mixed Signal Design Support. Infineon Technologies AG. Microelectronic Design Centers, Austria GmbH. P.O. Box 173, A-9500 Villach.

Schoenauer, M. (1998). Applications of Evolutionary Algorithms. Lecture notes of the Evonet Summer school on Evolutionary Computation. (ESSEC '98). Edited by Venturini, G. and Eiben, A.E. E3I, University of Tours, France.

Schwefel, H.-P. (1981). Numerical Optimization of Computer Models. Chichester, UK: John Wiley & Sons.

SEKE, (1999). The 11th International Conference on Software Engineering & Knowledge Engineering. Knowledge Systems Institute Graduate School (KSI), Skokie Illinois.

Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Perez-Uribe, A. and Stauffer, A. (1997). A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. IEEE Transactions on evolutionary Computation 1 (1), 83-97.

Smyth, B. (1996). Case Based Design. Ph.D. thesis, Department of Computer Science, Trinity College, University of Dublin, Ireland.

Stadler, P. F. (1996). Landscapes and their correlation functions. Journal of Mathematic al Chemistry 20, 1-45. Stadler, P. F. and Grunter, W. (1993). Anisotropy in fitness landscapes. Journal of Theoretical Biology 165, 373-388. 28

Stoica, A., Fukunaga, A., Hayworth, K. and Salazar-Lazaro, C. (1998). Evolvable Hardware for space applications. In: Sipper, M., Mange, D. and Perez-Uribe, A. (eds.), Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware, vol. 1478 of Lecture Notes in Computer Science, pp. 166-173. Heidelberg: Springer-Verlag.

Stoica, A., Keymeulen, D., Tawel, R., Salazar-Lazaro, C. and Li, W. (1999). Evolutionary experiments with a fine-grained reconfigurable architecture for analog and digital cmos circuits. In: Stoica, A., Keymeulen, D. and Lohn, J. (eds.), Proceedings of the 1st Nasa/DoD Workshop on Evolvable Hardware, pp. 76-84. Los Alamitos, CA: IEEE Computer Society.

Tanaka M. and Hira T. (1994). Genetic Case-Base for conceptual Structural Design. Japan-U.S.A. symposium on flexible automation. SCI.

Teller, A. and Veloso, M. (1995). PADO: Learning tree structured algorithms for orchestration into an object recognition system. Tech. Rep. CMU-CS-95-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, P A.

Thompson, A. (1996). Silicon Evolution. In Koza (Eds.), Genetic Programming 1996: Proceedings of the 1st Annual Conference on Genetic Programming (GP96), pp. 444-452. MIT Press.

Thompson, A. (1997). An evolved circuit, intrinsic in silicon, entwined with physics. In: Higuchi, T. and Iwata, M. (eds.), Proceedings of the 1st International Conference on Evolvable Systems: From Biology to Hardware, vol. 1259 of Lecture Notes in Computer Science, pp. 390-405. Heidelberg: Springer-Verlag.

Thompson, A., Layzell, P. and Zebulum, R. S. (1999). Explorations in design space: Unconventional electronics design through artificial evolution. IEEE Transactions on evolutionary Computation 3 (3), 167-196.

Thomson, P. and Miller, J. F. (1996). Symbolic method for simplifying and-exor representations of boolean functions using a binary decision technique and a genetic algorithm. IEE Proceedings in Computers and Digital Techniques 143, 151-155.

Vassilev, V.K., Job, D. and Miller, J.F. (2000). Towards the Automatic Design of more efficient digital circuits. The Second NASA/DoD Workshop on Evolvable Hardware.

Vassilev, V. K. (1997a). Information analysis of fitness landscapes. In: Husbands, P. and Harvey, I. (eds.), Proceedings of the 4th European Conference on Artificial Life, pp. 116-124. Cambridge, MA: MIT Press.

Vassilev, V. K. (1997b). An information measure of landscapes. In: Bäck, T. (ed.), Proceedings of the 7th International Conference on Genetic Algorithms, pp. 49-56. San Francisco, CA: Morgan Kaufmann.

Vassilev, V. K., Fogarty, T. C. and Miller, J. F. (2000). Information characteristics and the structure of landscapes. Evolutionary Computation 8 (1), 31-60.

Vassilev, V. K., Miller, J. F. and Fogarty, T. C. (1999a). Digital circuit Evolution and fitness landscapes. In: Proceedings of the Congress on evolutionary Computation, vol. 2, pp. 1299-1306. Piscata way, NJ: IEEE Press.

Vassilev, V. K., Miller, J. F. and Fogarty, T. C. (1999b). On the nature of two-bit multiplier landscapes. In: Stoica, A., Keymeulen, D. and Lohn, J. (eds.), Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware, pp. 36-45. Los Alamitos, CA: IEEE Computer Society.

Wilke, W., Bergmann, R., (1996). INRECA: Induction and Reasoning from Cases. Centre for Learning Systems and Applications, University of Kaiserslautern, P.O. Box 3049, D-67653 Kaiserslautern, Germany.

Wright, S. (1932). The roles of mutation, in breeding, crossbreeding and selection in evolution. In: Jones, D. F. (ed.), Proceedings of the 6th International Conference on Genetics, vol. 1, pp. 356-366. 29

Xilinx (1997). AppLINX CD-ROM - Xilinx application notes, data sheets, and other product information. Xilinx Ltd, Benchmark House, 203 Brooklands Rd., Weybridge, Surrey KT130RH. UK.

Xilinx (1996). The Programmable Logic Data Book. Xilinx Ltd, Benchmark House, 203 Brooklands Rd., Weybridge, Surrey KT130RH. UK.

Zebulum, R. S., Pacheco, M. A. and Vellasco, M. (1998). Analog circuits Evolution in extrinsic and intrinsic modes. In: Sipper, M., Mange, D. and Perez-Uribe, A. (eds.), Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware, vol. 1478 of Lecture Notes in Computer Science, pp. 154-165. Heidelberg: Springer-Verlag.

Zebulum, R. S., Pacheco, M. A. and Vellasco, M. (1999). Artificial Evolution on active filters: A case study. In: Stoica, A., Keymeulen, D. and Lohn, J. (eds.), Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware, pp. 66-75. Los Alamitos, CA: IEEE Computer Society.

Appendix 1 – Software Reuse strategies

High-Level Languages (HLL's) (Krueger, 1992; Biggerstaff, 1992)

In HLL's simple abstractions are used to represent multiple lines of assembly language code (see example below). The simple abstractions are easy to remember and to use them often only a few parameters need be specified. This technique is known to speed up software development approximately five times, and has been adopted in many areas of the software engineering community. The only drawback with this kind of system is that a large amount of analysis and design are required to create a system. This is due to the abstraction being only a level up from assembly language programming.

The high level language statement:

```
IF X == 0 THEN <Statement(s)>
ELSE <Statement(s)>
```

Replaces the following assembly language:

```
10: LOADA 100
11: JMPZRO 20
12: ADDA 110
13: ADDAI 111
:
20: SUBA 110
21: SUBAI 111
```

An example of a High-Level Language statement and equivalent assembly language.

The assembly language statements themselves are a similar level of abstraction above micro-code as HLL statements are above assembly language statements. Each assembly language statement represents a number of micro-code statements.

Design and Code Scavenging (Krueger, 1992)

Design and Code Scavenging involves reusing high-level language code fragments and is therefore a level of abstraction higher than high-level languages themselves. The reuse of the code fragments does require understanding of the code fragments and the code must be adapted to the new requirements manually. This can lead to highly successful reuse if large code fragments are found easily and require little adaptation. This technique is also potentially dangerous as it could take more time to find, understand, adapt and debug a scavenged piece of code than it would have taken to write the code fragment from scratch. (Mili, 1995) makes the important point that studies have shown that many users are able to successfully adapt and reuse components with only a rough understanding of the components workings.

Source Code Components (Krueger, 1992; Biggerstaff 1992; Mili, 1995)

Source Code Components are ready to use building blocks. These blocks are specifically designed for reuse, and vary greatly between domains. This means that no adaptation or debugging is required, the programmer need only locate and integrate the block of code. The programmer still requires an understanding of what the block does and how it works. This method works well in specific areas, but it is considered to be difficult to create a general set of source code components. If any editing of the component is required then adaptation and debugging is required and the benefits are lost. It is noted here that Object Oriented languages, through the inheritance mechanism, afford a direct relationship with the software reuse notions of abstraction (super class) and specialisation (subclass).

Software Schemas (Krueger, 1992; Mili, 1995)

Software Schemas are a level above source code components in that they extend the components to include reuse techniques (specification, parameterisation, classification and verification). This technique has achieved success in domain specific areas, the prominent feature being the controlled parameterisation, so the programmer does not require great understanding of the component. This technique fails when scaling up to cover wider domains. Most domains are difficult to build into this technique and some components are difficult to describe, increasing the need for the programmer to understand the components' code.

Application Generators (AG's) (Krueger, 1992; Biggerstaff 1992; Mili, 1995)

Application Generators are like a programming language compiler but highly automated and highly domain specific. Within their domain Application Generators are extremely effective and are excellent for particular problems. However they are difficult to build and there are too few of them available to give any wider domain coverage.

The abstractions used in Application generators are very high level, come directly from the Applications' domain and can be mapped directly into executable code. This technique requires little programming knowledge as most of the development is automated.

Very High-Level Languages (Krueger, 1992; Biggerstaff 1992; Mili, 1995)

Very High-Level Languages are an extension of High-Level Languages. The extension enables the programmer to generate code from an abstract specification. They are more general in domain than Application Generators but sacrifice the power that they have in order to achieve this. The VHLLs do use higher levels of abstractions than HLLs and a particular VHLL can be best suited to certain applications however, the level of abstraction can be difficult to use and this technique can also produce low performance code.

Transformational Systems (Krueger, 1992; Biggerstaff 1992; Mili, 1995)

Transformational Systems use a two-stage approach to development. Firstly the semantic behaviour of the program is described, and then the developers apply transformations to this specification to produce a program. These systems use rule-based expert knowledge to apply the transformations. These systems are general purpose and the level of abstraction is higher than that of VHLL's as the generated code is more efficient due to human guidance. This human guidance does entail understanding of the system and the program but these systems are expected to improve in time.

There are few transformational systems in use. The most recent research has been into their usefulness for systems maintenance, where they are known to sustain the quality of the software under maintenance.

Software Architectures (Krueger, 1992)

Software Architectures are an attempt to reuse large parts of designs and implementations. This technique looks at the subsystems and their interactions rather than reuse of algorithms and data structures. The architectural abstractions come directly from application domains and can be automatically mapped in to executable implementations. This is similar to having multiple application generators in a schema type reuse system. These systems can be used to generate complete software systems or components of software systems. The major drawback in this type of system is that many architectures are required for general domain coverage, and they are difficult to create.

These systems can be categorised into two main approaches to software reuse. Either they reduce the amount of work required to specify the system from the initial idea, or they reduce the effort required to produce executable code, once the specification is complete.

Appendix 2 – Example phenotypes (circuits)

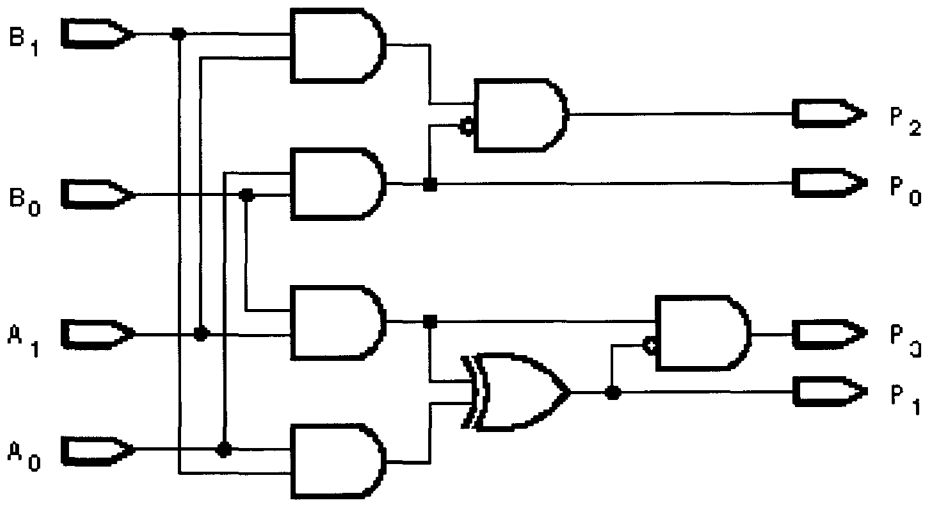


Figure A2.1 An evolved 2x2-bit multiplier

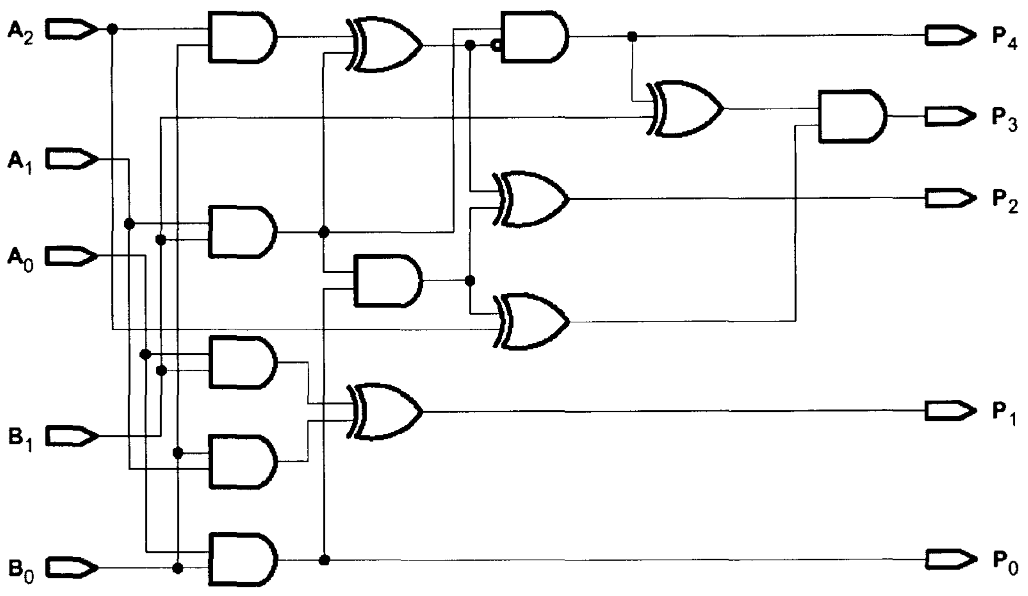


Figure A2.2 An evolved 3x2bit multiplier circuit.

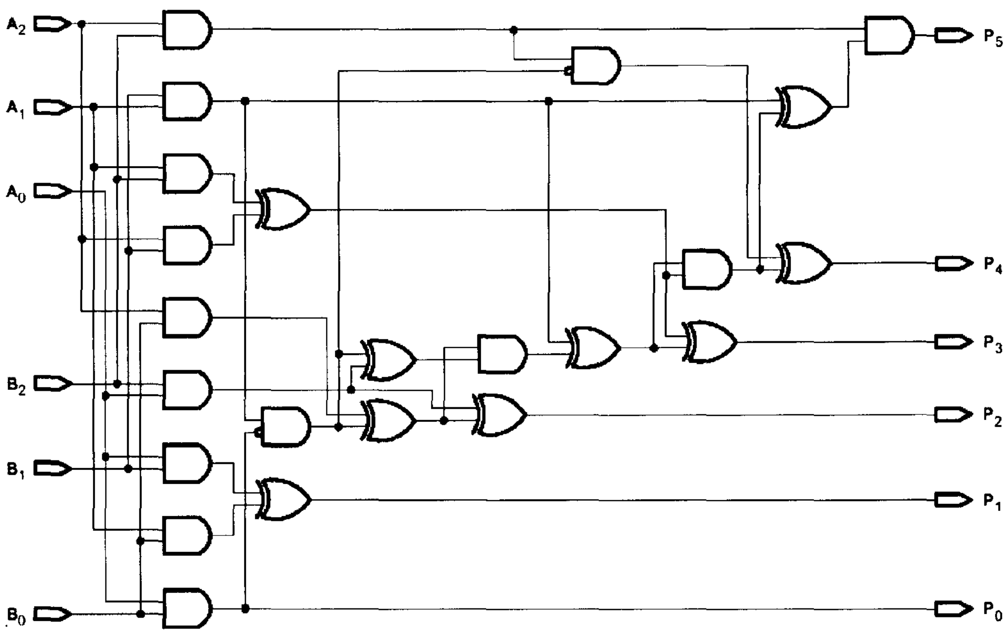


Figure A2.3 An evolved 3x3-bit multiplier.

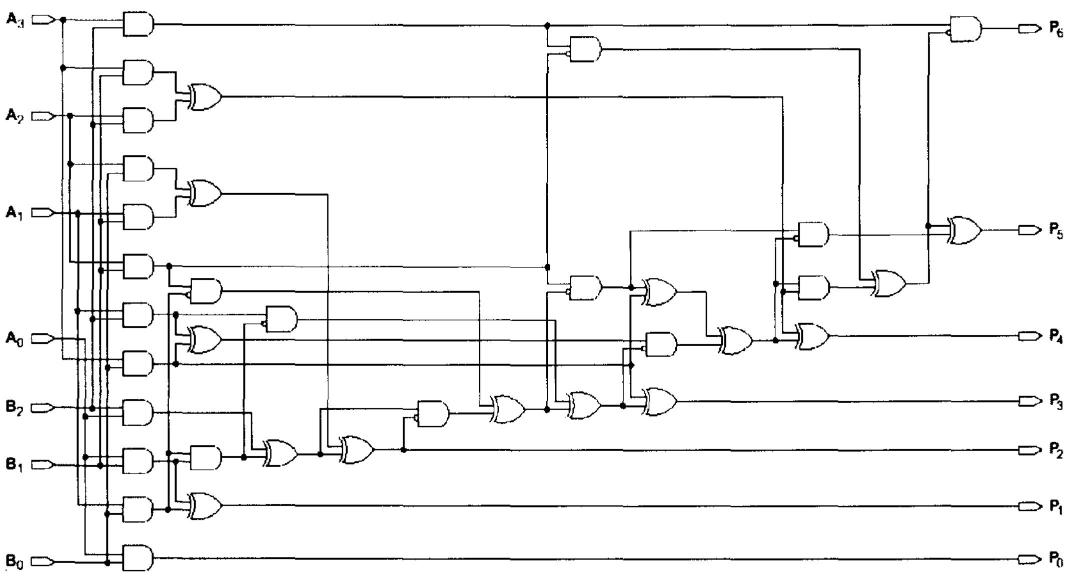


Figure A2.4 An evolved 4x3-bit multiplier.

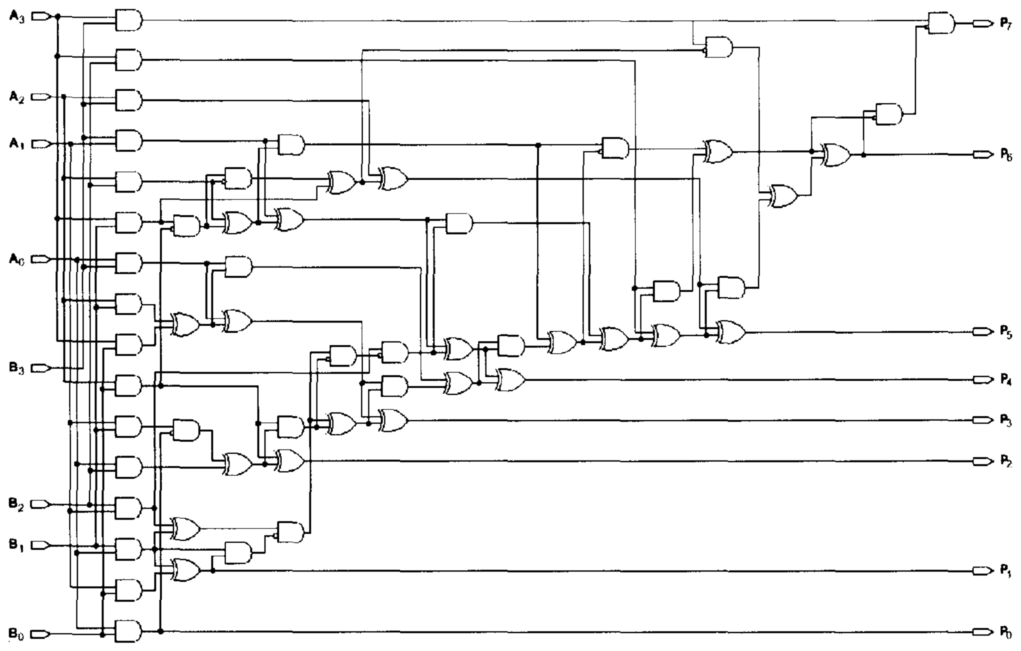


Figure A2.5 An evolved 4x4-bit multiplier.

Appendix 2.1 - Example PLA files.

PLA files:

inputs	6
outputs	1
products	64
000000	1
000001	0
000010	0
000011	1
000100	1
000101	0
000110	0
000111	0
001000	0
001001	1
001010	1
001011	0
001100	1
001101	1
001110	1
001111	0
010000	0
010001	1
010010	1
010011	0
010100	0
010101	1
010110	1
010111	1
011000	0
011001	1
011010	1
011011	0
011100	0
011101	0
011110	0
011111	1
100000	0
100001	1
100010	0
100011	1
100100	1
100101	1
100110	1
100111	0
101000	1
101001	0
101010	0
101011	1
101100	0

101101	0
101110	1
101111	0
110000	1
110001	0
110010	1
110011	1
110100	1
110101	0
110110	0
110111	1
111000	0
111001	1
111010	1
111011	0
111100	0
111101	0
111110	0
111111	1

Appendix 2 – Example phenotypes (circuits)

Table A2.1 SBox 10 PLA

inputs	3
outputs	2
products	8
000	00
001	01
010	01
011	10
100	01
101	10
110	10
111	11

Table A2.2 The one bit carry adder.

inputs	5
outputs	3
products	32
00000	000
00001	001
00010	001
00011	010
00100	010
00101	011
00110	011
00111	100
01000	001
01001	010
01010	010
01011	011
01100	011
01101	100
01110	100
01111	101
10000	010
10001	011
10010	011
10011	100
10100	100
10101	101
10110	101
10111	110
11000	011
11001	100
11010	100
11011	101
11100	101
11101	110
11110	110
11111	111

Table A2.3 The 2-bit carry adder.

inputs	4
outputs	4
products	16
0000	0000
0001	0000
0010	0000
0011	0000
0100	0000
0101	0001
0110	0010
0111	0011
1000	0000
1001	0010
1010	0100
1011	0110
1100	0000
1101	0011
1110	0110
1111	1001

Table A2.4 The 2x2-bit multiplier.

inputs	5
outputs	5
products	32
00000	00000
00001	00000
00010	00000
00011	00000
00100	00000
00101	00001
00110	00010
00111	00011
01000	00000
01001	00010
01010	00100
01011	00110
01100	00000
01101	00011
01110	00110
01111	01001
10000	00000
10001	00100
10010	01000
10011	01100
10100	00000
10101	00101
10110	01010
10111	01111
11000	00000
11001	00110
11010	01100
11011	10010
11100	00000
11101	00111
11110	01110
11111	10101

Table A2.5 The 2x3-bit multiplier

inputs	6
outputs	6
products	64
000000	000000
000001	000000
000010	000000
000011	000000
000100	000000
000101	000000
000110	000000
000111	000000
001000	000000
001001	000001
001010	000010
001011	000011
001100	000100
001101	000101
001110	000110
001111	000111
010000	000000
010001	000010
010010	000100
010011	000110
010100	001000
010101	001010
010110	001100
010111	001110
011000	000000
011001	000011
011010	000110
011011	001001
011100	001100
011101	001111
011110	010010
011111	010101
100000	000000
100001	000100
100010	001000
100011	001100
100100	010000
100101	010100
100110	011000
100111	011100
101000	000000
101001	000101
101010	001010
101011	001111
101100	010100
101101	011001
101110	011110
101111	100011
110000	000000

110001	000110
110010	001100
110011	010010
110100	011000
110101	011110
110110	100100
110111	101010
111000	000000
111001	000111
111010	001110
111011	010101
111100	011100
111101	100011
111110	101010
111111	110001

Table A2.6 The 3x3-bit multiplier.

Appendix 2 – Example phenotypes (circuits)

inputs	7
outputs	7
products	128
0000000	0000000
0000001	0000000
0000010	0000000
0000011	0000000
0000100	0000000
0000101	0000000
0000110	0000000
0000111	0000000
0001000	0000000
0001001	0000001
0001010	0000010
0001011	0000011
0001100	0000100
0001101	0000101
0001110	0000110
0001111	0000111
0010000	0000000
0010001	0000010
0010010	0000100
0010011	0000110
0010100	0001000
0010101	0001010
0010110	0001100
0010111	0001110
0011000	0000000
0011001	0000011
0011010	0000110
0011011	0001001
0011100	0001100
0011101	0001111
0011110	0010010
0011111	0010101
0100000	0000000
0100001	0000100
0100010	0001000
0100011	0001100
0100100	0010000
0100101	0010100
0100110	0011000
0100111	0011100
0101000	0000000
0101001	0000101
0101010	0001010
0101011	0001111
0101100	0010100
0101101	0011001
0101110	0011110
0101111	0100011
0110000	0000000

0110001	0000110
0110010	0001100
0110011	0010010
0110100	0011000
0110101	0011110
0110110	0100100
0110111	0101010
0111000	0000000
0111001	0000111
0111010	0001110
0111011	0010101
0111100	0011100
0111101	0100011
0111110	0101010
0111111	0110001
1000000	0000000
1000001	0001000
1000010	0010000
1000011	0011000
1000100	0100000
1000101	0101000
1000110	0110000
1000111	0111000
1001000	0000000
1001001	0001001
1001010	0010010
1001011	0011011
1001100	0100100
1001101	0101101
1001110	0110110
1001111	0111111
1010000	0000000
1010001	0001010
1010010	0010100
1010011	0011110
1010100	0101000
1010101	0110010
1010110	0111100
1010111	1000110
1011000	0000000
1011001	0001011
1011010	0010110
1011011	0100001
1011100	0101100
1011101	0110111
1011110	1000010
1011111	1001101
1100000	0000000
1100001	0001100
1100010	0011000
1100011	0100100
1100100	0110000
1100101	0111100

1100110	1001000
1100111	1010100
1101000	0000000
1101001	0001101
1101010	0011010
1101011	0100111
1101100	0110100
1101101	1000001
1101110	1001110
1101111	1011011
1110000	0000000
1110001	0001110
1110010	0011100
1110011	0101010
1110100	0111000
1110101	1000110
1110110	1010100
1110111	1100010
1111000	0000000
1111001	0001111
1111010	0011110
1111011	0101101
1111100	0111100
1111101	1001011
1111110	1011010
1111111	1101001

Table A2.7 The 4x3-bit multiplier

inputs	8
outputs	8
products	256
00000000	00000000
00000001	00000000
00000010	00000000
00000011	00000000
00000100	00000000
00000101	00000000
00000110	00000000
00000111	00000000
00001000	00000000
00001001	00000000
00001010	00000000
00001011	00000000
00001100	00000000
00001101	00000000
00001110	00000000
00001111	00000000
00010000	00000000
00010001	00000001
00010010	00000010
00010011	00000011
00010100	00000100
00010101	00000101
00010110	00000110
00010111	00000111
00011000	00001000
00011001	00001001
00011010	00001010
00011011	00001011
00011100	00001100
00011101	00001101
00011110	00001110
00011111	00001111
00100000	00000000
00100001	00000010
00100010	00000100
00100011	00000110
00100100	00001000
00100101	00001010
00100110	00001100
00100111	00001110
00101000	00010000
00101001	00010010
00101010	00010100
00101011	00010110
00101100	00011000
00101101	00011010
00101110	00011100
00101111	00011110
00110000	00000000

00110001	00000011
00110010	00000110
00110011	00001001
00110100	00001100
00110101	00001111
00110110	00010010
00110111	00010101
00111000	00011000
00111001	00011011
00111010	00011110
00111011	00100001
00111100	00100100
00111101	00100111
00111110	00101010
00111111	00101101
01000000	00000000
01000001	00000100
01000010	00001000
01000011	00001100
01000100	00010000
01000101	00010100
01000110	00011000
01000111	00011100
01001000	00100000
01001001	00100100
01001010	00101000
01001011	00101100
01001100	00110000
01001101	00110100
01001110	00111000
01001111	00111100
01010000	00000000
01010001	00000101
01010010	00001010
01010011	00001111
01010100	00010100
01010101	00011001
01010110	00011110
01010111	00100011
01011000	00101000
01011001	00101101
01011010	00110010
01011011	00110111
01011100	00111100
01011101	01000001
01011110	01000110
01011111	01001011
01100000	00000000
01100001	00000110
01100010	00001100
01100011	00010010
01100100	00011000
01100101	00011110

01100110	00100100
01100111	00101010
01101000	00110000
01101001	00110110
01101010	00111100
01101011	01000010
01101100	01001000
01101101	01001110
01101110	01010100
01101111	01011010
01110000	00000000
01110001	00000111
01110010	00001110
01110011	00010101
01110100	00011100
01110101	00100011
01110110	00101010
01110111	00110001
01111000	00111000
01111001	00111111
01111010	01000110
01111011	01001101
01111100	01010100
01111101	01011011
01111110	01100010
01111111	01101001
10000000	00000000
10000001	00001000
10000010	00010000
10000011	00011000
10000100	00100000
10000101	00101000
10000110	00110000
10000111	00111000
10001000	01000000
10001001	01001000
10001010	01010000
10001011	01011000
10001100	01100000
10001101	01101000
10001110	01110000
10001111	01111000
10010000	00000000
10010001	00001001
10010010	00010010
10010011	00011011
10010100	00100100
10010101	00101101
10010110	00110110
10010111	00111111
10011000	01001000
10011001	01010001
10011010	01011010

10011011	01100011
10011100	01101100
10011101	01110101
10011110	01111110
10011111	10000111
10100000	00000000
10100001	00001010
10100010	00010100
10100011	00011110
10100100	00101000
10100101	00110010
10100110	00111100
10100111	01000110
10101000	01010000
10101001	01011010
10101010	01100100
10101011	01101110
10101100	01111000
10101101	10000010
10101110	10001100
10101111	10010110
10110000	00000000
10110001	00001011
10110010	00010110
10110011	00100001
10110100	00101100
10110101	00110111
10110110	01000010
10110111	01001101
10111000	01011000
10111001	01100011
10111010	01101110
10111011	01111001
10111100	10000100
10111101	10001111
10111110	10011010
10111111	10100101
11000000	00000000
11000001	00001100
11000010	00011000
11000011	00100100
11000100	00110000
11000101	00111100
11000110	01001000
11000111	01010100
11001000	01100000
11001001	01101100
11001010	01111000
11001011	10000100
11001100	10010000
11001101	10011100
11001110	10101000
11001111	10110100

11010000	00000000
11010001	00001101
11010010	00011010
11010011	00100111
11010100	00110100
11010101	01000001
11010110	01001110
11010111	01011011
11011000	01101000
11011001	01110101
11011010	10000010
11011011	10001111
11011100	10011100
11011101	10101001
11011110	10110110
11011111	11000011
11100000	00000000
11100001	00001110
11100010	00011100
11100011	00101010
11100100	00111000
11100101	01000110
11100110	01010100
11100111	01100010
11101000	01110000
11101001	01111110
11101010	10001100
11101011	10011010
11101100	10101000
11101101	10110110
11101110	11000100
11101111	11010010
11110000	00000000
11110001	00001111
11110010	00011110
11110011	00101101
11110100	00111100
11110101	01001011
11110110	01011010
11110111	01101001
11111000	01111000
11111001	10000111
11111010	10010110
11111011	10100101
11111100	10110100
11111101	11000011
11111110	11010010
11111111	11100001

Table A2.8 The 4x4-bit multiplier.

Appendix 3 - Example Adaptation

The 3x3–bit multiplier (ref. #95):

2 5 4 6	1 3 1 6	0 3 0 6	6 8 7 6	4 9 6 6	0 5 2 6
2 3 3 6	1 4 4 6	9 13 2 10	0 4 0 6	14 6 0 7	11 12 14 10
7 14 0 6	16 17 15 7	14 19 3 10	8 18 0 6	16 17 16 10	1 5 9 6
2 4 6 6	7 15 0 10	20 25 19 6	10 21 19 10	8 26 16 10	28 21 17 7
23 24 7 10	20 25 24 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
27	29	31	22	30	6

Figure A3.1

Is found to be a behavioural match with:

The 3x2-bit multiplier (ref. #30):

0 4 1 6	2 3 2 6	2 4 1 6	1 3 6 6	0 3 6 6	1 4 5 6
8 7 6 7	5 11 0 10	11 5 6 7	5 8 0 6	6 10 0 10	8 13 13 10
9 16 2 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	14	17	12	15
7					

Figure A3.2

Then an adaptation is identified:

#1

0 255 0 255 0 255 0 255	AND	0 0 0 0 0 255 0 255
0 0 0 0 255 255 255 255		

#2

0 0 255 255 0 0 255 255	AND	0 0 0 0 0 0 255 255
0 0 0 0 255 255 255 255		

#3

160 160 160 160 160 160 160 160	AND	0 0 0 0 0 0 160 160
#2		

#4

204 204 204 204 204 204 204 204	AND	0 0 0 0 0 0 128 128
#3		

#5

240 240 240 240 240 240 240 240	AND	0 0 0 0 240 240 240 240
0 0 0 0 255 255 255 255		

#6

#3	EOR	0 204 0 204 0 204 160 108
0 204 0 204 0 204 0 204		

#7

0 0 170 170 0 0 170 170	EOR	0 0 170 170 240 240 90 90
#5		

#8

#1	AND	0 0 0 0 0 204 0 108
#6		

#9

0 76 0 76 0 76 0 76	AND NOT	0 76 0 68 0 12 0 4
#7		

#10

#6	EOR	0 128 0 136 0 192 160 104
#9		

#11

#2	AND	0 0 0 0 0 0 0 108
#8		

#12

0 76 0 76 0 76 0 76	EOR	0 76 170 230 240 188 90 22
#7		

#13

#1	EOR	0 0 204 204 0 255 204 51
0 0 204 204 0 0 204 204		

#14

#10	AND	0 0 0 136 0 192 128 32
#13		

#15

#4	EOR	0 0 0 0 0 0 128 236
#11		

#16

#2	EOR	0 0 0 136 0 192 127 223
#14		

#17

#16	AND NOT	0 0 0 136 0 192 127 147
#11		

#18

#10	EOR	0 128 204 68 0 63 108 91
#13		

Outputs : #s:15, 17, 18, 12, original, original

Then #30 (the 3x2-bit multiplier) is renumbered to take an extra input:

(#30) – extra input:

0 5 0 6	2 4 0 6	2 5 0 6	1 4 0 6	0 4 0 6	1 5 0 6
9 8 0 7	6 12 0 10	12 6 0 7	6 9 0 6	7 11 0 10	9 14 0 10
10 17 0 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	0	15	18	13
16	8				

Next cells that are no longer needed (deletions and substitutions) are replaced by the first BCTs:

(#30)- substitutions:

0 5 0 6	2 4 0 6	2 5 0 6	1 4 0 6	0 4 0 6	1 5 0 6

9 8 0 7	6 12 0 10	2 3 0 6	6 14 0 10	12 15 0 7	6 9 0 6
7 11 0 10	9 16 0 10	10 19 0 10	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0	17
20	13	18	8		

Then the geometry of the 3x2–bit multiplier (1x22) is expanded to that of the target 3x3-bit multiplier (1x26) and then rest of the BCTs are added:

#30 – adapted:

0 5 0 6	2 4 0 6	2 5 0 6	1 4 0 6	0 4 0 6	1 5 0 6
9 8 0 7	1 3 0 6	0 3 0 6	8 14 0 6	7 11 0 10	4 15 0 6
2 3 0 6	15 9 0 10	6 18 0 10	13 19 0 6	12 20 0 7	19 22 0 10
14 21 0 6	12 20 0 10	13 10 0 10	23 26 0 6	17 24 0 10	14 27 0 10
29 24 0 7	23 26 0 10	28	30	31	25
16	8				

Adaptation example 2

#56

2 4 3 6	0 3 0 6	0 4 6 6	1 3 5 6	7 8 0 6	8 5 7 7
1 4 6 6	10 9 3 7	2 3 2 6	7 10 2 10	6 8 10 10	11 13 9 10
12 15 9 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	9	17	14	16
5					

#56 – expanded for extra input:

2 5 0 6	0 4 0 6	0 5 0 6	1 4 6 6	8 9 0 6	9 6 0 7
1 5 0 6	11 10 0 7	2 4 0 6	8 11 0 10	7 9 0 10	12 14 0 10
13 16 0 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	0	10	18	15
17	6				

#56 – expanded and adapted using the adaptation given:

2 5 0 6	0 4 0 6	0 5 0 6	1 4 6 6	8 9 0 6	9 6 0 7
1 5 0 6	11 10 0 7	2 4 0 6	8 11 0 10	7 9 0 10	12 14 0 10
13 16 0 10	1 3 0 6	0 3 0 6	2 3 0 6	21 3 0 10	19 7 0 10
20 6 0 6	24 9 0 10	24 3 0 6	19 25 0 6	27 20 0 6	26 28 0 10
11 22 0 7	11 22 0 10	30 25 0 10	32 23 0 6	32 23 0 10	33 25 0 10
35 28 0 7	29	36	34	31	17
6					

Appendix 4 – Scaled behavioural matches

A list of scaled behavioural matches found between a 3x2-bit multiplier Case-Base and a 3x3-bit multiplier Case-Base.

Case-base 1 2x3-bit multipliers	Case-base 2 3x3-bit multipliers
16	136
30	86
30	95
36	18
56	102
56	125

Table A4.1. 3x2-bit multipliers that have a 3x3-bit multiplier scaled behavioural match.

#16
Contains 14 gates

2 3 3 6	1 3 0 6	0 3 0 6	2 4 2 6	1 4 8 6	0 4 7 6
5 9 5 6	6 11 5 10	7 9 3 6	5 9 4 10	11 13 6 7	7 15 14 10
13 16 13 10	10 12 13 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	13	17	18	14
8					

#136
Contains 27 gates

1 3 4 6	0 4 5 6	0 5 4 6	2 3 5 6	8 9 2 6	1 4 8 6
2 5 1 6	1 5 7 6	0 3 9 6	6 7 11 10	11 12 9 6	2 4 5 6
16 15 14 7	13 17 8 10	10 15 9 10	11 16 2 10	8 9 16 10	10 21 8 10
14 23 17 6	18 20 8 10	24 25 16 7	11 22 13 6	14 27 9 10	21 22 5 10
24 20 27 7	28 30 4 7	25 27 4 10	0 0 0 -1	0 0 0 -1	0 0 0 -1
26	31	32	29	19	12

#30
Contains 13 gates

0 4 1 6	2 3 2 6	2 4 1 6	1 3 6 6	0 3 6 6	1 4 5 6
8 7 6 7	5 11 0 10	11 5 6 7	5 8 0 6	6 10 0 10	8 13 13 10
9 16 2 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	14	17	12	15
7					

#86
Contains 26 gates

0 4 0 6	0 3 6 6	0 5 6 6	1 5 5 6	1 4 4 6	1 3 8 6
2 3 11 6	2 5 10 6	10 13 7 7	6 11 1 10	12 14 2 10	8 12 8 10
2 4 12 6	16 17 7 7	3 19 16 6	10 17 20 6	20 15 3 7	7 13 16 7
10 19 7 10	7 24 18 6	8 16 16 10	15 24 3 10	22 23 3 10	9 18 20 10
21 28 26 10	22 25 2 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
31	30	27	26	29	13

#30
Contains 13 gates

0 4 1 6	2 3 2 6	2 4 1 6	1 3 6 6	0 3 6 6	1 4 5 6

8 7 6 7	5 11 0 10	11 5 6 7	5 8 0 6	6 10 0 10	8 13 13 10
9 16 2 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	14	17	12	15
7					

#95
Contains 26 gates

2 5 4 6	1 3 1 6	0 3 0 6	6 8 7 6	4 9 6 6	0 5 2 6
2 3 3 6	1 4 4 6	9 13 2 10	0 4 0 6	14 6 0 7	11 12 14 10
7 14 0 6	16 17 15 7	14 19 3 10	8 18 0 6	16 17 16 10	1 5 9 6
2 4 6 6	7 15 0 10	20 25 19 6	10 21 19 10	8 26 16 10	28 21 17 7
23 24 7 10	20 25 24 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
27	29	31	22	30	6

#36
Contains 14 gates

2 4 0 6	0 3 4 6	0 4 4 6	2 3 7 6	1 4 3 6	8 9 0 10
8 9 4 6	1 3 8 6	6 9 0 7	11 6 11 7	7 12 2 6	7 12 10 10
13 14 15 10	11 16 3 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	15	17	18	10
5					

#18
Contains 27 gates

0 3 5 6	1 4 2 6	1 5 1 6	1 3 0 6	0 5 6 6	0 4 7 6
2 3 6 6	7 12 9 10	10 13 2 6	9 11 13 10	2 5 4 6	12 13 14 7
14 15 7 10	6 7 8 7	2 4 8 6	8 20 19 6	8 20 4 10	14 15 21 6
19 23 20 10	10 13 4 10	6 24 6 7	21 25 2 10	21 25 10 6	18 28 4 10
17 29 19 10	29 30 7 7	24 31 28 10	0 0 0 -1	0 0 0 -1	0 0 0 -1
26	32	30	27	22	16

#56
Contains 13 gates

2 4 3 6	0 3 0 6	0 4 6 6	1 3 5 6	7 8 0 6	8 5 7 7
1 4 6 6	10 9 3 7	2 3 2 6	7 10 2 10	6 8 10 10	11 13 9 10
12 15 9 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	9	17	14	16
5					

#102
Contains 26 gates

2 3 0 6	0 4 4 6	1 5 2 6	2 4 3 6	0 3 7 6	0 5 3 6
6 11 1 10	6 11 4 6	1 4 6 6	8 9 4 10	13 15 15 6	13 14 16 10
2 5 1 6	1 3 5 6	12 14 11 6	10 14 14 6	14 18 11 7	10 16 17 10
7 17 5 10	20 23 22 10	22 12 8 7	12 22 23 10	24 26 5 10	16 21 10 10
25 26 6 7	19 28 20 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
29	30	31	27	15	18

#56
Contains 13 gates

2 4 3 6	0 3 0 6	0 4 6 6	1 3 5 6	7 8 0 6	8 5 7 7
1 4 6 6	10 9 3 7	2 3 2 6	7 10 2 10	6 8 10 10	11 13 9 10
12 15 9 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
0 0 0 -1	0 0 0 -1	9	17	14	16
5					

#125
Contains 26 gates

2 3 0 6	0 5 2 6	6 7 3 10	2 5 8 6	1 3 7 6	2 4 5 6
1 5 8 6	0 3 3 6	9 13 12 6	1 4 3 6	0 4 14 6	8 15 9 6
14 15 16 10	15 9 15 7	13 17 5 10	13 18 18 6	10 16 3 10	8 19 15 10
11 12 3 10	14 22 21 7	21 25 10 7	18 22 1 10	19 23 27 6	21 25 13 10
20 26 2 7	27 28 24 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
29	30	31	23	24	9

Appendix 5 - The 2-bit adder with carry, '2-into-1' frequencies

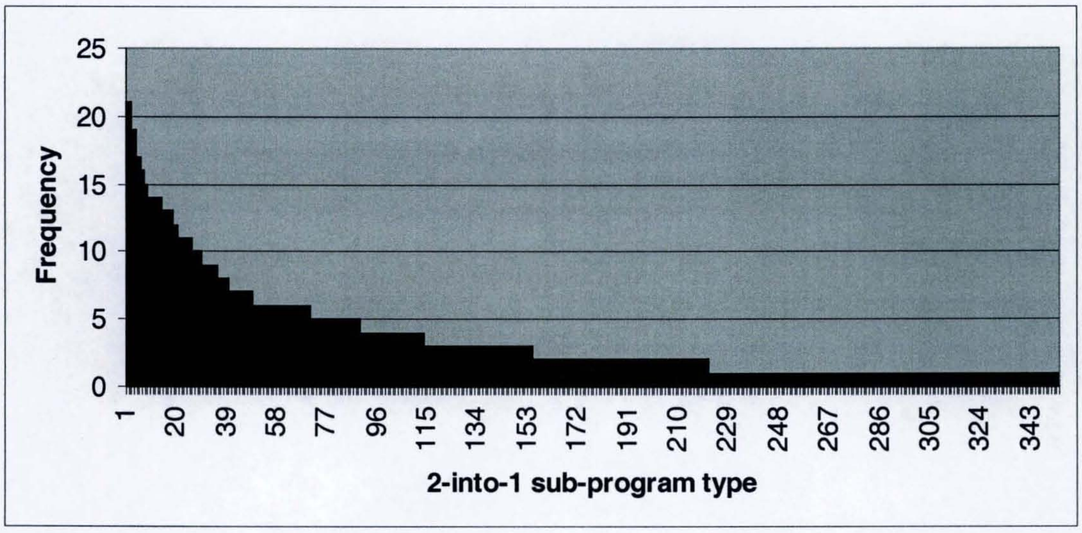


Figure A5.1 Frequencies of '2-into-1' sub-programs for the 2-bit adder with carry, from a set of 500 chromosomes. Geometry: 1 row x 10 columns, gates allowed: 6-15.

#	Gate1	Gate2	Gate3	Frequency
1	11	10	13	21
2	10	10	6	21
3	11	10	11	19
4	11	10	8	19
5	10	11	14	17
6	11	11	9	17
7	10	10	7	16
8	11	11	13	15
9	11	11	12	14
10	12	14	6	14
11	10	11	7	14
12	10	11	11	14
13	11	10	10	14
14	10	11	12	14

Figure A5.2 the top 14 '2-into-1' sub-programs for the 2-bit adder with carry, shown in Figure A5.1.

Appendix 6 – An example Case

Index	Behaviour (converted from binary into 8 cells of 8-bit integers)							
0	0	0	255	255	0	0	255	255
1	0	255	0	255	0	255	0	255
2	240	240	240	240	240	240	240	240
3	0	0	0	0	255	255	255	255
4	204	204	204	204	204	204	204	204
5	170	170	170	170	170	170	170	170

Table A6.1 An annotated example of the Inputs (Behaviour) in a Case.

6 (o 0)	160	160	160	160	160	160	160	160
7	0	0	0	0	0	255	0	255
8	0	0	0	0	0	0	255	255
9	0	0	0	0	0	0	160	160
10	0	0	0	0	0	0	128	128
11	0	0	170	170	0	0	170	170
12	0	0	0	0	240	240	240	240
13	0	204	0	204	0	204	0	204
14	0	204	0	204	0	204	160	108
15	0	0	204	204	0	0	204	204
16	0	76	0	76	0	76	0	76
17	0	0	170	170	240	240	90	90
18	0	0	0	0	0	204	0	108
19	0	76	0	68	0	12	0	4
20	0	128	0	136	0	192	160	104
21	0	0	0	0	0	0	0	108
22 (o 2)	0	76	170	230	240	188	90	22
23	0	170	0	170	0	170	0	170
24	192	192	192	192	192	192	192	192
25	0	0	204	204	0	255	204	51
26	0	0	0	136	0	192	128	32
27 (o 5)	0	0	0	0	0	0	128	236
28	0	0	0	136	0	192	127	223
29 (o 4)	0	0	0	136	0	192	127	147
30 (o 1)	192	106	192	106	192	106	192	106
31 (o 3)	0	128	204	68	0	63	108	91

Table A6.2 An annotated example of the Behaviour in a Case.

2 5 4 6	1 3 1 6	0 3 0 6	6 8 7 6	4 9 6 6	0 5 2 6
2 3 3 6	1 4 4 6	9 13 2 10	0 4 0 6	14 6 0 7	11 12 14 10
7 14 0 6	16 17 15 7	14 19 3 10	8 18 0 6	16 17 16 10	1 5 9 6
2 4 6 6	7 15 0 10	20 25 19 6	10 21 19 10	8 26 16 10	28 21 17 7
23 24 7 10	20 25 24 10	0 0 0 -1	0 0 0 -1	0 0 0 -1	0 0 0 -1
27	29	31	22	30	6

Table A6.3 An example of a Structure in a Case.

Fitness	384 (100% functional) + 4 (Redundant cells e.g. 30-26=4)
Frequency	4 (Number of times this program occurred)
Generation	6434526
Number of rows	1
Number of columns	30
Number of gates	26

Table A6.4 An example of the other attributes and values in a Case.

Gate 1	Gate 2	Gate 3	Frequency
100	105	6	1
102	103	10	1
6	105	10	1
100	104	6	1
103	100	6	1
6	6	7	2
6	6	10	2
...	...	...	...

Table A6.5 An example of the sub-programs in a Case. This is a 'finger-print' for the Case Values of 100 and over represent inputs to the program, values less than 100 represent gate types, of the 2-into-1 sub-programs.

Appendix 7 – The SBox problem

An investigation into evolving the SBox encryption circuits. This investigation by Dominic Job was based on work in Miller, J. F. and Thomson, P. 1998 a and b.

In considering new architectures for digital circuit evolution, there are two key issues (Miller, J. F. and Thomson, P. 1998a and b):

(a) Functionality of cells - where evolution selects the logical functionality of a particular cell, and determines whether or not that cell should possess functionality.
(b) Routing - where the routes that provide the interconnection between functional cells are evolved into non-functional cells (or functional cells which are partially used as routes).

The routeability of circuits is an important practical issue that is aptly demonstrated by the SBOX combinational design problem. This is a circuit used in data encryption. Traditional synthesis techniques are not able to produce a design which will place and route on to the Xilinx 6216 within the geometric bounding-box currently desired by engineers. This shows the importance of designing in such a way as to incorporate both issues of functionality of circuits and also the manner in which they route on target devices (Miller, J. F. and Thomson, P. 1998a and b).

Each Sbox has 6 inputs, 4 outputs and 64 lines in the truth table. In each experiment an Evolutionary Strategy was used with a population size of 5 over 10,000,000 generations, with elitism and a mutation rate of 1.5%. Two multiplexers were the available functions: ((A & ~C) | (B & C)), ((A & ~C) | (~B & C)) as these two functions are known to be capable of producing the required functionality. In the three experiments below for the Sbox1, Sbox2 and Sbox3 problems, the allowed geometry was increased from 8x12 for the first experiment to 8x16 in the second and lastly 8x32 in the third. This was done to attempt to get a 100% perfect solution to one of the problems, as the high fitness values e.g. 99.22% with 50% perfection suggest that 100% perfect solutions are attainable. The structure of the chromosomes given is explained in the reference texts (Miller, J. F. and Thomson, P. 1998a and b).

Experiment 1: Sbox1

The first experiment was to attempt to place and route the Sbox1 problem in an 8x12 horizontal cell area on a Xilinx 6216 FPGA with 'levels back' = 12 (the number of columns across which a connection can be made to an earlier column in the cell area). The 8x12 area requires 100 genes in the chromosome. The results are given in table 1 below, and the best chromosome is then given.

The result was achieved at generation:	Percentage fitness	Percentage perfect*
2520338	96.88	0.00
6481415	98.05	0.00
6411700	96.09	0.00
7839230	97.27	0.00
8126761	94.92	0.00
2618692	98.44	0.00
8223080	96.48	0.00
9785943	98.44	50.00
4303018	97.27	0.00

*Percentage perfect is the percentage of the truth table that is perfectly calculated from top to bottom without breaks, reported at 25% intervals.

Table 1. The Sbox1 problem in an 8x12 horizontal cell area.

Best chromosome from experiment 1:

At generation 9785943 the following result was achieved

5 5 0 17	1 0 3 17	2 3 3 17	5 1 3 17	1 5 4 16	2 1 4 17
3 1 2 17	2 4 0 17	12 13 3 16	9 8 0 16	6 8 12 17	6 9 10 17
6 10 3 16	3 4 5 16	0 0 11 17	13 13 5 17	10 13 19 17	20 15 10 16
4 18 3 16	3 16 11 17	18 11 17 16	17 9 19 17	14 2 4 17	16 2 21 16
15 18 12 17	28 29 15 17	28 28 27 17	26 11 7 17	4 2 22 17	24 18 26 17
21 22 3 16	25 28 23 16	35 26 11 17	30 32 33 16	33 3 20 17	23 4 12 17
13 29 11 16	20 33 35 16	31 12 10 17	2 0 30 17	38 43 27 16	45 29 43 17
10 18 7 17	40 31 1 16	23 33 36 16	9 37 40 17	44 32 3 17	39 34 9 17
6 43 26 16	37 47 42 16	6 26 28 16	9 41 45 16	0 8 33 17	48 22 52 16
32 31 8 16	23 31 14 17	33 53 29 17	43 50 23 16	54 34 3 16	23 28 58 16
35 47 61 16	35 54 41 16	53 49 5 16	56 50 10 17	64 16 33 16	13 47 42 16
12 39 30 16	27 25 25 16	45 16 9 17	62 26 56 17	37 14 31 17	50 67 24 16
36 12 59 16	10 59 27 16	72 70 35 16	70 71 19 17	23 54 68 17	48 56 22 17
75 18 56 16	75 43 28 16	2 84 66 17	28 2 29 16	53 61 20 16	17 53 12 17
68 76 40 17	43 30 28 17	49 39 73 16	5 50 51 17	9 71 62 16	77 71 85 17
1 37 63 17	9 41 29 16	59 51 5 16	42 59 85 16	71 16 45 16	16 83 21 17
77 55 68 99					

Experiment 2: Sbox2

The second experiment was to attempt to place and route the Sbox2 function in an 8x16 horizontal cell area on a Xilinx 6216 FPGA with 'levels back' = 16. The 8x16 area requires 132 genes in the chromosome. The results are given in table 2 below, and the best chromosome is then given.

The result was achieved at generation:	Percentage fitness	Percentage perfect*
3134822	97.27	0.00
8841161	98.05	50.00
6539456	98.44	50.00
1433786	98.44	50.00
8010889	98.44	50.00
6179232	96.88	0.00
4305603	98.05	0.00
7723414	96.09	0.00
1454634	98.44	50.00
3548762	98.44	50.00
8267265	94.92	0.00
7318510	97.66	0.00
2107938	98.05	50.00
3830277	98.44	50.00
2631902	98.44	50.00
3552944	97.66	0.00
8672225	96.09	0.00

3205873	98.44	50.00

*Percentage perfect is the percentage of the truth table that is perfectly calculated from top to bottom without breaks, reported at 25% intervals.

Table 2. The Sbox2 problem in an 8x16 horizontal cell area.

Best chromosome from experiment 2:

At generation 3205873 the following result was achieved

2 0 5 16	2 5 0 17	0 4 5 16	5 5 4 17	1 2 1 17	2 2 3 17
1 0 2 17	0 3 0 16	13 10 9 17	5 8 9 17	1 10 1 17	4 7 7 17
1 5 10 17	3 2 1 17	11 10 8 17	6 11 7 16	7 1 20 17	1 0 20 17
7 14 18 17	19 2 8 16	0 4 11 17	11 12 14 16	1 1 9 17	18 2 5 17
27 13 6 16	1 26 5 16	17 14 12 17	23 26 21 16	25 21 29 16	18 2 15 17
10 25 24 16	20 27 12 16	36 25 35 17	36 36 31 17	35 12 11 16	15 22 34 17
34 22 32 17	0 26 7 16	16 22 37 16	6 19 28 16	37 40 32 17	40 39 29 16
33 3 39 16	9 33 1 16	34 25 20 17	27 32 27 16	42 44 30 16	10 9 34 16
40 45 11 16	9 25 19 17	7 30 18 17	30 30 48 17	52 52 4 17	17 15 4 16
50 27 26 17	47 47 13 17	45 20 13 17	39 19 28 16	2 16 32 17	7 35 13 16
8 10 43 16	51 59 2 16	30 23 31 16	8 20 41 16	32 56 43 16	47 32 15 16
60 61 52 16	2 53 66 17	31 17 64 17	57 27 45 17	38 16 58 17	56 44 5 16
28 28 49 17	75 43 41 16	56 9 64 16	56 45 7 17	69 58 48 16	55 42 34 17
32 29 70 16	24 54 59 17	85 26 14 17	76 42 19 16	55 29 18 16	75 63 15 17
50 23 52 17	74 50 74 17	28 79 54 16	61 61 4 17	33 74 87 17	83 16 64 17
65 7 0 17	51 64 56 17	7 23 32 17	73 21 60 16	88 39 39 17	76 67 19 16
90 90 88 17	43 82 0 17	21 90 75 17	83 29 43 17	4 63 45 16	69 96 94 16
66 25 56 17	66 88 21 16	2 77 3 16	56 64 105 17	50 99 79 17	14 52 3 16
100 49 50 16	10 100 92 17	84 102 45 17	20 72 15 17	58 47 69 17	39 55 25 17
90 108 51 17	73 22 98 16	43 79 20 16	50 50 106 17	19 28 63 16	83 117 54 16
53 115 11 17	7 48 2 17	122 35 60 17	65 76 12 17	41 22 34 17	76 3 20 17
117 56 58 16	86 108 95 16	58 93 125 89			

Experiment 3: Sbox3

The third experiment was to attempt to place and route the Sbox3 function in an 8x32 horizontal cell area on a Xilinx 6216 FPGA with 'levels back' = 32. The 8x32 area requires 260 genes in the chromosome. The results are given in table 3 below, and the best chromosome is then given.

The result was achieved at generation:	Percentage fitness	Percentage perfect*
4107590	99.22	50.00
9147835	98.05	0.00
2597586	96.09	0.00
5974820	99.22	50.00
1732460	99.22	50.00

*Percentage perfect is the percentage of the truth table that is perfectly calculated from top to bottom without breaks, reported at 25% intervals.

Table 3. The Sbox3 problem in an 8x32 horizontal cell area.

Best chromosome from experiment 3:

At generation 4107590 the following result was achieved:

5 3 1 17	3 0 0 17	0 2 5 16	5 4 2 16	5 0 5 17
4 5 2 17	5 0 2 16	2 2 1 17	4 4 5 16	9 8 7 16
1 9 7 16	4 11 11 16	3 10 5 17	9 12 5 17	2 2 5 17
3 3 1 17	14 5 16 17	5 5 13 17	14 18 16 17	10 18 21 16
11 6 8 16	1 17 15 17	2 15 8 17	17 12 1 16	20 11 7 17
6 13 27 17	22 27 29 16	10 17 1 17	28 28 27 17	6 17 14 16
23 4 12 17	26 26 25 17	35 1 31 16	24 23 29 16	12 20 0 16
28 14 0 16	4 6 33 16	19 7 6 17	11 23 22 17	13 4 18 16
10 41 18 16	36 7 8 16	40 24 30 17	30 41 19 16	28 40 6 16
7 45 36 17	31 40 12 16	34 6 17 16	25 18 53 17	13 12 45 17
12 12 11 16	33 48 42 16	48 12 1 16	3 42 49 17	27 29 28 16
3 47 38 16	40 36 12 17	35 8 44 16	28 54 18 17	24 45 44 16
8 0 13 16	27 35 23 16	36 38 21 17	61 44 49 16	36 19 46 17
33 54 19 16	56 64 68 17	24 20 57 17	50 43 25 17	54 3 58 17
42 3 7 17	17 27 37 16	12 21 60 16	54 69 56 17	44 57 59 16
60 38 65 16	37 37 12 17	45 17 12 17	63 10 74 16	17 5 26 16
18 31 38 16	19 53 57 16	36 48 78 16	36 17 30 16	35 73 36 16
81 3 22 16	66 51 79 16	41 66 84 16	54 75 67 17	84 46 11 16
44 42 69 17	66 38 65 16	10 29 66 16	23 76 29 17	39 29 72 17
45 19 41 16	76 13 77 17	74 74 82 16	63 67 23 17	34 94 64 17
48 87 2 17	59 32 1 16	101 88 17 17	19 41 93 16	46 42 19 16
60 76 76 17	37 23 9 17	16 67 10 17	6 58 68 17	33 50 32 17
80 26 66 16	55 40 82 17	54 105 5 17	88 87 46 16	103 74 77 17
49 29 3 17	9 38 72 17	21 109 61 16	34 97 85 16	106 108 15 17
53 32 4 17	111 68 63 17	57 116 24 16	7 96 64 17	54 79 116 16
7 50 111 16	31 103 103 16	92 92 80 17	101 36 24 17	99 125 21 16
30 26 124 16	13 110 91 17	101 14 2 16	55 13 105 16	131 16 86 17
118 2 17 17	113 121 21 16	33 59 3 17	99 131 96 17	112 81 108 16
92 133 121 17	55 37 27 17	54 44 138 17	48 127 136 16	78 9 131 17
85 46 45 16	36 118 98 17	32 37 36 17	95 106 80 16	144 14 112 16
15 24 68 16	143 50 137 17	42 115 48 17	15 18 41 16	90 52 20 17
86 25 16 17	129 99 64 16	11 103 24 17	58 120 60 16	148 64 96 16
21 111 54 16	150 83 22 16	37 156 57 16	105 134 140 17	69 83 25 16
57 25 46 17	31 77 16 17	138 125 153 17	50 156 75 16	8 160 54 16
128 90 122 16	104 116 5 17	143 68 31 16	1 124 42 16	0 25 95 17
1 160 124 16	165 28 153 17	141 161 54 17	164 78 69 16	96 146 139 17
180 117 56 17	151 176 88 16	167 145 11 17	58 149 38 16	111 183 77 16
64 184 64 17	41 130 18 16	180 157 15 16	27 118 9 17	145 112 4 16
127 98 176 17	27 185 187 17	93 183 140 16	0 187 50 17	77 124 191 16
29 80 180 17	180 70 129 16	81 121 10 16	107 173 115 16	58 127 8 17
11 164 187 16	154 182 137 16	61 196 182 17	22 73 127 17	151 62 202 17
47 182 128 17	112 2 102 17	13 118 41 16	12 134 66 16	120 18 1 17
21 140 181 16	159 113 18 17	37 50 149 16	43 156 15 17	211 107 13 16
132 144 44 17	107 120 49 17	207 69 75 17	209 23 167 16	86 13 212 17
96 144 183 16	107 209 23 17	94 106 101 16	195 152 72 17	48 69 146 17
212 23 150 17	39 193 212 17	221 33 75 17	153 82 11 16	7 186 144 16

Appendix 7 – The SBox problem

150 209 5 16	167 180 92 17	207 151 105 17	59 150 175 17	148 213 15 17
102 144 214 17	156 83 124 16	2 106 181 16	92 156 227 17	154 219 121 17
16 24 26 17	20 128 61 16	179 69 216 16	97 136 169 16	110 45 9 16
35 84 40 17	244 169 180 17	55 29 74 16	158 4 57 16	43 28 226 17
39 112 106 17	180 204 155 17	63 140 204 17	5 177 185 17	79 200 209 17
65 214 13 17	252 200 146 240			

Appendix 8 – A list of publications arising out of the research

Evolutionary Computation with Case-Based Reasoning
Dominic Job and Venky Shankararaman. (Selected papers from the UK CBR 2000 Workshop) in Expert Update. Summer 2001, Vol 4, No 2. SGES. ISSN 1465-4091.

Towards the Automatic Design of more efficient digital circuits
Vessiln K. Vassilev, Dominic Job and Julian F. Miller 2000. The Second NASA/DoD Workshop on Evolvable Hardware

Principles in the Evolutionary Design of Digital Circuits -- Part I
J. F. Miller, D. Job, and V. K. Vassilev 2000. Journal of Genetic Programming and Evolvable Machines, Vol. 1, No. 1, (pp. 8-35)

Principles in the Evolutionary Design of Digital Circuits -- Part II
J. F. Miller, D. Job, and V. K. Vassilev 2000. Journal of Genetic Programming and Evolvable Machines, Vol. 1, No. 2, (pp. 259-288)

Hybrid AI Techniques for Software Design
Dominic Job, Venky Shankararaman and Julian Miller 1999. The 11th International Conference on Software Engineering and Knowledge Engineering. Printed by Knowledge Systems Institute Graduate School, Skokie Illinois.

The Genetic Algorithm as a Discovery Engine: Strange Circuits and New Principles.
Miller J., T. Kalganova, N. Lipnitskaya and D. Job (1999) Proc. of the AISB Symposium on Creative Evolutionary Systems (CES'99). Edinburgh, UK.

Hybrid AI Techniques for Automated Software Reuse
D. Job, V. Shankararaman and J. Miller 1999. International Conference on Case-Based Reasoning. Technical Report of the Centre for Learning Systems and Applications (LSA) of the University of Kaiserslautern.

Combining CBR and GA for Designing FPGAs
D. Job, V. Shankararaman and J. Miller 1999. Proceedings of the 3rd International Conference on Computational Intelligence and Multimedia Applications.

Appendix 9 – An explanation of the acronyms used in this thesis

FPGA - Field Programmable Gate Array

FPGA programs are a limited form of program consisting of a feed forward network of primitive logic functions.

This thesis examines a specific area of software reuse in the programming of Field Programmable Gate Arrays (FPGAs). An FPGA is a programmable microchip that takes as a program a representation of a digital logic circuit. The FPGA takes on the digital circuit configuration given to it as a program. FPGAs have the advantage over Gate Arrays in that they do not have to be manufactured for a specific purpose. . FPGAs can also be quickly reprogrammed to fulfil a new specification, whereas non-programmable Gate Arrays cannot (Xilinx, 1996).

Logic programming can be seen as a specific kind of software programming. Gate Arrays are microchips that require a Logic program to perform a function. Field programmable Gate Arrays (FPGAs) are a specific type of gate array that is user programmable, and can be reused by reprogramming it with a new program. Normal Gate arrays are programmed once and then discarded after use.

CBR - Case-Based Reasoning

CBR is a problem solving method that reuses old solutions to solve new problems

CB - Case-Base

A collection of problem-solution pairs, used in CBR

HCBR - Hierarchical Case Base Reasoning

Hierarchical Case Base Reasoning (HCBR) (Smyth, 1996) are designed to facilitate indexing and retrieval by organising the Cases into a hierarchy where specific Cases are indexed under more general Cases.

A form of CBR where the CB can been seen as a hierarchy of cases where the cases are more general higher up the hierarchy. The lowest cases are problem-solution pairs, the higher a case is the more general it is. This facilitates matching and reasoning as the distance between cases can be measured by the number of nodes of the hierarchy tree that have to be ascended to find a common case.

AGR - Adaptation Guided Retrieval

AGR finds existing Cases in the Case-Base that can be best adapted to solve the problem, rather than simply retrieving the closest Case to the problem Case

EC - Evolutionary Computation

Computational techniques based on theories or natural evolution and natural systems, for example Darwinian theory and ant colonies.

CGP - Cartesian Genetic Programming

A digital logic circuit is encoded as a more general graph based computational model called CGP (Miller, 1999a). CGP is a graph-based form of genetic programming. Other graph based genetic programming forms are Parallel Distributed GP (PDGP) proposed by Poli (1997) and Parallel Algorithm Discovery and Orchestration (PADO) (Teller and Veloso, 1995). CGP represents a data-flow graph (Banzhaf *et al.*, 1998).

NLP - Natural Language Processing

Fully automated computing techniques and methods for interpreting and making useful inferences (understanding) about natural language e.g. the interpretation of English, or French by a computer.

Natural Language Processing (NLP) can extract knowledge from Natural Language documents and other data-mining techniques can produce knowledge e.g. by Filtering or mapping legacy databases to new uses

SR - Software Reuse

Software reuse (SR) encompasses many techniques. In general these techniques fall into one of the categories give in Table 2.1. Each technique aims to reuse existing knowledge as much as possible whilst minimising the amount of new work required to produce a satisfactory solution (Krueger, 1992).

PLA - Programmable logic array

PLA files (PLA stands for programmable logic array) commonly specify combinational logic functions. A PLA file is a truth table with additional information about the numbers of inputs, outputs and products of the target program, and has the format shown in Table 2.2. A PLA file differs from a Truth Table in that a PLA file need not have all outputs or inputs specified.

TT - Truth Table

Binary table of inputs and outputs for a digital function.

TFQ - The Fundamental Question

"Can we by evolving a series of sub-systems of increasing size, extract the general principle and hence discover new principles?"

BCT - Behavioural Context Triple

Like a two input logic gate (AND, OR, XOR etc) but including 'context' where the context of a gate is defined as the binary inputs and outputs for the specific gate in a specific circuit with a specific function such as multiplication.