Reliable Electronics through Artificial Evolution

Miguel Garvie

Submitted for the degree of D.Phil.
University of Sussex
January, 2005
Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signature:
Acknowledgements

This work would not have been possible without the help of Adrian Thompson, Phil Husbands and the COGS bursary supporting my research.

I would also like to specially thank the following people: Matthew Quinn, Paul Layzell, Pablo Vazquez, Robert Vickerstaff, Andrew Balaam, Jöerg Langeheine and the Electronic Visions team at Heidelberg University, Daniel Lund and all the contributors to the Distributed Hardware Evolution Project, Lincoln Smith, Andrew Phillipides, Regis Leveugle and Inman Harvey.
Reliable Electronics through Artificial Evolution

Miguel Garvie

Summary

This thesis will demonstrate the use of evolutionary algorithms applied to electronic circuit designs (i.e., Evolutionary Electronics) to improve the reliability of hardware in several ways. It will introduce Jiggling, an architecture for the self-repair of space-deployed reconfigurable hardware. Jiggling uses spare resources efficiently at a fine-grained level and if extended to sequential circuits would not depend on any unprotected repair mechanism. Reliability analysis of systems protected by Jiggling show they can survive with above 0.99 probability for 48 times longer than with the standard TMR/Simplex method. An architecture to distribute any evolutionary process across workstations and public contributors through the Internet is introduced and is shown to enable circuits of industrial interest to be evolved. A method to synthesise digital combinational and sequential circuits with totally self-checking (TSC) concurrent error detection (CED) is shown to generate designs using less than 30% of duplication overhead, a figure previously unheard of in the literature. These designs do not adopt the traditional functional logic-checker structural constraint and exhibit multiple self-checking strategies well suited to each circuit’s particularities. The absolutely fault secure class of circuits is introduced as the most stringent one detecting multiple faults with no assumptions on their arrival time and it is shown that evolution can synthesise such circuits. The first method for the automatic synthesis of generic analog circuits with CED is also introduced and it is shown how Evolutionary Electronics is probably the best suited tool for the task. Finally the first example of transfer of a design evolved on real hardware to a different hardware medium is demonstrated. This is paramount because industry cannot be expected to produce evolved circuits in the same medium they were evolved in.

Submitted for the degree of D. Phil.

University of Sussex

January, 2005
# Contents

1 Introduction .................................................. 8
   1.1 Motivation ................................................. 8
   1.2 State of the Art .......................................... 8
   1.3 Background .............................................. 9
      1.3.1 Evolutionary Algorithms .......................... 9
      1.3.2 Hardware Failure Modes ......................... 9
   1.4 Summary .................................................. 10
   1.5 The Thesis .............................................. 11

2 Context ....................................................... 13
   2.1 Evolutionary Electronics – The Tool .................. 13
      2.1.1 Evolutionary Electronics in action – Step by step 13
      2.1.2 Example of intrinsic evolution – Evolution moves through mysterious design search spaces .......... 15
      2.1.3 Genetic Algorithm model – A continuing SAGA .......... 16
   2.2 Reliable Electronics – The Problem ................... 18
   2.3 Evolutionary Electronics and Reliability .......... 18
   2.4 Position of the thesis within the fields .......... 19
   2.5 Thesis Structure ........................................ 19

3 Jiggling: Long survival of FPGA systems through evolutionary self-repair ..... 20
   3.1 Introduction ............................................. 20
   3.2 TMR+Jiggling ............................................. 22
      3.2.1 Transient SEU Mitigation .......................... 22
      3.2.2 Jiggling Architecture Overview .................. 23
      3.2.3 Specification ......................................... 24
      3.2.4 Hardware Implementation .......................... 25
      3.2.5 Simulated Jiggling .................................. 27
      3.2.6 Probability Model ................................... 28
      3.2.7 Benchmark Circuits .................................. 30
   3.3 Results ................................................... 30
      3.3.1 Discussion ........................................... 33
   3.4 Predicting fitness function effectiveness .......... 34
      3.4.1 Random Data Tests .................................. 35
      3.4.2 Structural Tests ..................................... 35
      3.4.3 Fitness function tests vs. real runs: do they match? 38
      3.4.4 Conclusions ......................................... 39
   3.5 Jiggler Variations ........................................ 40
      3.5.1 Results .............................................. 41
   3.6 Trading overhead for time .............................. 42
      3.6.1 Architecture variations implementing different trade-offs .......... 42
      3.6.2 Analysis ............................................. 43
      3.6.3 A minimal Jiggler .................................... 44
      3.6.4 Non-concurrent repair .............................. 44
6.4.8  Combination of Multiple Continuous Fitness Functions  ........................................ 92
6.5  Results .................................................................................................................. 93
  6.5.1  Filters .................................................................................................................. 93
  6.5.2  Digital to Analog Converter ................................................................................. 94
  6.5.3  Op-Amp ............................................................................................................... 95
6.6  Conclusion ............................................................................................................. 99

7  From an Evolvable Motherboard to a Breadboard .................................................. 101
  7.1  Introduction .......................................................................................................... 101
  7.2  Setup .................................................................................................................... 101
    7.2.1  Genetic Algorithm ........................................................................................... 102
    7.2.2  Evolving tone discrimination .......................................................................... 102
  7.3  Results ................................................................................................................. 103
    7.3.1  Early configurations ....................................................................................... 103
    7.3.2  Functional configurations .............................................................................. 103
    7.3.3  Analysis of 8465 ............................................................................................ 104
    7.3.4  Transferring 8465 to a breadboard ................................................................. 106
  7.4  Conclusion ........................................................................................................... 107
  7.5  Future Transfer .................................................................................................... 107

8  Conclusions .............................................................................................................. 108

A  Appendices ............................................................................................................... 110
  A.1  Proof of expressions in Table 3.5 ...................................................................... 110
    A.1.1  A - All Once .................................................................................................... 110
    A.1.2  B - In Order .................................................................................................... 111
    A.1.3  C - Count & Table ........................................................................................ 111
    A.1.4  D - Keen Counters ....................................................................................... 112
  A.2  Distributing a GA Step by Step ........................................................................... 112
    A.2.1  Interfacing any GA package to IslandEv ..................................................... 112
    A.2.2  Setting up your distributed islands evolutionary process ............................. 114
  A.3  Output function generator synthesis using Sis .................................................. 115
    A.3.1  Combinational ............................................................................................... 115
    A.3.2  Sequential ..................................................................................................... 116
Chapter 1

Introduction

1.1 Motivation

Evolutionary Electronics (EE) is the application of evolutionary algorithms (EAs) to electronic circuit design/optimisation. EE has been shown to creatively solve complex circuit design problems (Thompson, 1998b; Tyrrell et al., 2003; Layzell, 2001; Koza et al., 2004; Lohn et al., 2003b) by arriving at unusual designs which efficiently exploit available resources. Unlike a human designer, or the designer of a computer aided design tool, EE is not bound to understand the design problem through a divide and conquer approach and can address all dependencies simultaneously. By operating at a holistic level, EE can search a design space containing radically new structures with complex and subtle interdependencies. EE is an automated hardware design process capable of producing any kind of digital or analog circuit.

The design of reliable hardware involves producing reliable behaviour from unreliable components (Pradhan, 1986; Wakerly, 1978; Sellers et al., 1968; Neumann, 1956). This is a complex task in which hardware resources must be used carefully since each one added to a design may itself fail. It is possible that many efficient solutions to this problem, such as those found in nature, are not to be found through a divide and conquer design approach. The importance of reliable hardware is stressed today as shrinking feature sizes drive increasing fault rates and as the rising ubiquity of hardware risks lives and capital loss upon its failure. The need for autonomous hardware repair is also stressed as more complex probes explore deeper into space. Thus EE seems to be a well-matched tool to improve hardware reliability.

1.2 State of the Art

EE has been applied to the design of digital and analog circuits. Efforts focusing on solving practical problems found in industry are few and far between, while almost all evolved designs remain of small size. Many evolved designs did not transfer from the medium in which they were evolved to other technologies, specially when they used parasitic properties of the source medium.

Circuits with concurrent error detection (CED) improve system reliability by signalling erroneous behaviour. They are traditionally designed by producing encoded output and then checking if the code is broken. EE has not previously been applied to their design and there are no automatic synthesis tools for producing analog circuits with CED. Autonomous self-repair schemes for space-deployed reconfigurable hardware have been proposed which use precomputed alternative blocks working around faults to replace the area around the faulty configuration. Biologically inspired and genetic algorithm (GA) driven self-repair schemes have also been proposed. All self-repair schemes proposed suffer from the problem that they introduce a large unprotected repair mechanism.
The state of the art of each field this thesis touches upon will be discussed in further detail at the start of each experimental chapter.

1.3 Background

1.3.1 Evolutionary Algorithms

Genetic Algorithms (GAs) (Holland, 1975; Goldberg, 1989), Genetic Programming (GP) (Koza, 1992) and Evolutionary Strategies (ES) (Schwefel and Rudolph, 1995) all work by repeated application of blind variation and selection. The basic algorithm works by evaluating individuals each of which encodes a potential solution to the problem at hand, selecting the fittest and subjecting these to genetic operators such as mutation and crossover in order to arrive at new genotypes. It is usually the case that as this process is repeated the probability of finding fitter individuals in the population increases. If a generational GA is used, all individuals are evaluated together and the next generation is produced from their offspring. If elitism is used, the fittest individuals in the previous generation are copied unmodified into the next generation. In the case of GAs and ES a potential solution to a problem is typically encoded into a linear string of bits or numbers called a genotype. In the case of GP the algorithm works on a tree structure. Genetic operators for linear genotypes include mutation which changes a random section of a genotype, and single-point crossover which interchanges genetic material from two genotypes on the right hand side of a randomly chosen position. When evolving hardware the genotype encodes a circuit structure.

There are two extreme approaches to the use of GAs. The first uses large populations with a lot of crossover and few generations. The argument is that a large population seeds evolution with lots of genetic diversity and that the crossover operator combines fit building blocks (Holland, 1975) which are iteratively selected until there is no genetic diversity left in the population. The second uses a small population with many generations so that mutation does most of the design work. The argument (Harvey, 1992b) is that evolution is a tinkering process in which mutation iteratively improves the design represented by a semi-converged population which explores design space. This is especially the case when there are many genotypes which encode individuals with equal fitness, and these genotypes are one mutation away from each other. These genotypes define neutral networks along which the semi-converged population can drift with a chance of finding a fitter neutral network to jump onto. The second approach will be adopted during this work.

1.3.2 Hardware Failure Modes

A component is said to fail when it does not meet its functional specification (Russell and Sayers, 1989). A fault may be caused by the manufacturing process or during the component’s lifetime. One common cause of hardware failure is mishandling. Inappropriate handling of electronic devices could lead to static discharge or physical damage. Designer error could also cause hardware failure for example by feeding too much current into a sensitive component. There are several causes for hardware failure in the absence of human error.

Transient

The most common cause of transient faults are single event upsets (SEUs). These are caused by charged particles such as alpha, beta, gamma particles or heavy ions which are present in outer space and near radioactive sources. When a charged particle crosses a transistor (Fig.1.1) it creates positive and negative charges in its wake which can generate enough current to activate the transistor. This could introduce an error into a circuit’s output and could also flip a memory cell’s value. Many field programmable gate arrays (FPGAs) store their configuration in SRAM which means cosmic rays can modify the circuit implemented by an FPGA. Cosmic particles, a major concern in outer space, are not so common at sea-level except during the maximum activity of the solar cycle and close to radioactive sources. Cosmic particles do reach sea-level and as die sizes shrink transistors become more sensitive to them.
Chapter 1. Introduction

1. Intermittent

Certain permanent faults may manifest themselves intermittently, e.g., only when two shorted digital lines should have different values. Intermittent faults may also be due to bad design, e.g., race conditions or components which are coupled or too sensitive to noise.

1.4 Permanent

Single event latch-up (SEL) occurs when a current induced by a charged particle activates parasitic transistor structures between the desired transistors (Fig. 1.2). These parasitic transistors may be activated into a circuit with a large positive feedback which may draw too much current and cause permanent damage.

Electromigration occurs when high current densities in thin metallic conductors cause the conductor atoms to be displaced by the force of the travelling electrons. If enough of these are displaced they may cause open circuit failure (Fig. 1.3(a)) or short circuit failure (Fig. 1.3(b)). Electromigration is accelerated by the existence of large currents but is also increasingly likely as a device with normal use ages.

Another cause of permanent or intermittent hardware failure may be an undetected manufacturing defect. This could be a short circuit which passed manufacturing testing because it initially had a high resistance. This could then manifest itself in the field. Such manufacturing defects are associated with the burn-in period found on the left hand side of the ‘bathtub’ failure rate vs. device life-time curve. This is followed by a period with low failure rate which then increases again on the right of the bathtub curve as aging faults manifest themselves.

1.4 Summary

This thesis will demonstrate the use of evolutionary algorithms applied to electronic circuit designs (i.e., Evolutionary Electronics) to improve the reliability of hardware in several ways. It will introduce Jiggling, an architecture for the self-repair of space-deployed reconfigurable hardware. Jiggling uses spare resources efficiently at a fine-grained level and if extended to sequential circuits would not depend on any unprotected repair mechanism. Reliability analysis of systems protected by Jiggling show they can survive with above 0.99 probability for 48 times longer than with the standard TMR/Simplex method. An architecture to distribute any evolutionary process
across workstations and public contributors through the Internet is introduced and is shown to enable circuits of industrial interest to be evolved. A method to synthesise digital combinational and sequential circuits with totally self-checking (TSC) concurrent error detection (CED) is shown to generate designs using less than 30% of duplication overhead, a figure previously unheard of in the literature. These designs do not adopt the traditional functional logic-checker structural constraint and exhibit multiple self-checking strategies well suited to each circuit's particularities. The absolutely fault secure class of circuits is introduced as the most stringent one detecting multiple faults with no assumptions on their arrival time and it is shown that evolution can synthesise such circuits. The first method for the automatic synthesis of generic analog circuits with CED is also introduced and it is shown how Evolutionary Electronics is probably the best suited tool for the task. Finally the first example of transfer of a design evolved on real hardware to a different hardware medium is demonstrated. This is paramount because industry cannot be expected to produce evolved circuits in the same medium they were evolved in.

### 1.5 The Thesis

The main thesis is that EE can improve the reliability of hardware. A list of specific objectives this thesis aims to demonstrate will now be listed:

1. The self-repair of FPGA configurations using a GA can be made practical to protect 100 year space missions.

2. Circuits of hundreds of gates can be evolved by distributing evolution across the Internet generating a free supercomputer using the spare processing power of university workstations and computers belonging to public contributors.

3. There exist efficient low overhead circuits with CED having no structural decomposition between functional logic and checker, and evolutionary search is capable of finding them.

4. It is possible to add logic around a fixed irredundant circuit to give it full CED with less than duplication overhead without imposing any structural constraints on the original circuit, and evolution can do this automatically.

5. Evolution can synthesise circuits with CED using a self-checking strategy well adapted to each circuits particularities and can also create novel strategies.

6. Evolution can synthesise circuits with CED using multiple well-suited strategies in each design.
7. Evolution can synthesise circuits with CED using on average less than 30% of duplication overhead, a lower overhead ratio than any previous technique.

8. As engineers have learnt from nature’s evolved designs, so can useful design principles be extracted from artificially evolved designs.

9. Circuits with full detection of any number of faults do exist and evolution is capable of finding them.

10. An automatic synthesis method for generic analog circuits with CED using evolution is possible.

11. A hardware design evolved in reconfigurable hardware can be transferred to a different hardware medium by simulating the source medium’s parasitics.
Chapter 2

Context

This chapter will present a broad overview of the fields touched by this thesis and its position within them. The fields will be presented in greater depth at the start of each experimental chapter.

2.1 Evolutionary Electronics – The Tool

Evolutionary algorithms (Holland, 1975; Goldberg, 1989; Schwefel and Rudolph, 1995; Koza, 1992) operating by repeated application of blind variation and selection have been used in numerous fields. Research into their application to electronic circuit design began a decade ago and Evolutionary Electronics (EE) has now grown into a large field with three conferences every two years (Tyrrell et al., 2003; Lohn et al., 2003b; Zebulum et al., 2004) and numerous research groups worldwide. Landmarks in the field include the circuits evolved in simulation by Koza et al. (1996), the first circuits evolved on real hardware by Thompson et al. (1996) and the first VLSI chip produced exclusively for evolution of electronic circuits at JPL, NASA (Stoica et al., 2000).

Most research has focused on demonstrating how EE could be used to solve a certain class of problems, ie. developing EE as a tool. However there are not many examples of the application of EE as a tool to specific practical problems. Antenna design (Lohn et al., 2001), adaptive pattern recognition (Yasunaga et al., 2000) and image data compression (T. Higuchi et al., 1999) are some scattered examples. One barrier to the evolution of industry-competitive circuits is the poor scalability of the approach. Koza and Andre (1995); Koza et al. (2004) used a grid of computers to evolve circuits competitive with previous patents and novel patentable circuits. Another barrier to the adoption of evolved designs in industry is that they often do not transfer reliably (Thompson, 1998c; Trefzer et al., 2004) out of the medium they were evolved into a target technology.

2.1.1 Evolutionary Electronics in action – Step by step

This section will describe the simple GA used throughout most of this work: a generational GA with a binary genotype, linear rank selection, single point cross-over, mutation and elitism. Let population size be 30, cross-over probability be 0.5 and per bit mutation rate 0.01. These parameters will seem unconventional to most readers acquainted with GAs and are justified in §2.1.3.

The GA starts with a population of 30 random genotypes. Each genotype is a fixed length binary string and every bit is random.

Each genotype encodes a circuit structure as defined by the genotype to phenotype mapping. This mapping is analogous to how an FPGA configuration stream defines a circuit implementation on the chip itself. Indeed the GA in Thompson et al. (1996) operates on a population of Xilinx 6200 configuration streams which are then mapped into circuits by programming the FPGA directly. Alternately, a custom mapping could be defined as the one defined in Vassilev and Miller (2000). In this the genotype defines a feed-forward digital circuit. This circuit can be imagined as a row
of gates such that gates only take inputs from gates on their left. Now the genotype encodes the function of each of these gates in a block of bits called a gene. Each gene dedicates a sub-block of bits to defining the gate function (e.g., and, or, not, etc...) and other blocks to defining where the gate inputs come from. The gate function is often, as in most of this thesis, represented as the output column of the truth table. For example a two-input and gate would be represented as 0001, an nxor gate as 1001 and an inverter of the second input as 1010. The mapping defines the latter as a binary encoded integer specifying how many gates left the source gate is, such that overshooting past the first gate specifies primary inputs. Finally, a few blocks at the start of the genotype define which gates the primary outputs are taken from.

Once a mapping from binary string to circuit is defined the GA is ready to instantiate each genotype as a circuit. This can be done on real hardware such as an FPGA or done in a simulator. These are called intrinsic and extrinsic evolution respectively. Once a circuit is instantiated the GA will evaluate its fitness based on how close its behaviour is to the target. For example to evaluate a combinational circuit it is sufficient to collect the output response to every input. Then, a fitness function calculates a number from the output response data. For example it could add up the number of correct bits in the output response. This fitness evaluation process is then carried out for every genotype, or individual, in the population.

Once all individuals in a population have an associated fitness value the creation of the next generation begins. Since there is elitism, the individual with the best fitness score gets copied unmodified into the new generation. After this, 29 individuals are selected using linear rank selection. This means that the population is sorted and it is twice as likely to choose the best than to choose the median, and all other probabilities are linearly interpolated.

Each of these 29 selected genotypes then undergoes “blind variation” as follows. With 50% chance it will undergo sexual recombination through single point cross-over. If this happens then a partner is selected from the population using the same scheme. The offspring of these parent genotypes is formed by the genetic material of the mother up to point \( p \) and that of the father from \( p \) to the end, where \( p \) is a random position in the genotype. This offspring, or the original selected individual if cross-over did not take place, is then mutated. This is done by iterating over each bit and flipping it with a probability of 0.01.

Once all 29 individuals are thus created the new generation has 30 individuals and all their fitnesses are evaluated as above. After this they are selected and varied to create the next generation and so on as shown in Fig. 2.1... This process ends either when a target fitness has been attained or
2.1.2 Example of intrinsic evolution – Evolution moves through mysterious design search spaces

One of the finest examples of EE at work is also one of the earliest. Thompson et al. (1996) sets out to evolve a tone discriminator on a FPGA. The goal was to evolve a circuit that distinguished between a 1Khz and a 10Khz input without using a clock.

Population size was 50, cross-over probability 0.7 and per bit mutation rate was such that 2.7 mutations were expected per genotype. Genotype length was 1800 bits and this configured a 10x10 section of a Xilinx 6200 FPGA directly. Circuits were evaluated by integrating the analog response to 5 1Khz and 5 10Khz tones applied in random order for 500ms each. Let the integrator reading at the end of test tone $t$ be denoted $i_t$ ($t=1,2,\ldots,10$). Let $S_1$ be the set of five 1Khz test tones, and $S_{10}$ the set of five 10Khz test tones. Then the individual’s fitness was calculated as:

$$f = \frac{1}{5} \left| \left( k_1 \sum_{t \in S_1} i_t \right) - \left( k_2 \sum_{t \in S_{10}} i_t \right) \right|$$

where $k_1 = 1/30730.746$ and $k_2 = 1/30527.973$.

When the GA was run the population was already semi-converged after roughly 50 generations. This was measured by calculating the average Hamming distance between all pairs of genotypes. From then on a semi-converged population explores search space mostly through mutation. Figure 2.2 shows output response at different generations throughout evolution. The circuit in generation 3500 provides clean low output when stimulated at 1Khz and clean high output when stimulated at 10Khz.
Analysis of the functional configuration showed that all but 32 of the 100 cells could be clamped to 0 or 1 without affecting behaviour. From Thompson (1998b) *The circuit discriminated between inputs of period 1ms and 0.1ms using only 32 cells, each with a propagating delay of less than 5ns, and with no off-chip components whatsoever: a surprising feat.* What was unusual was that some of the cells which affected circuit behaviour when clamped to 0 had no connection path by which they could affect the output. Evolution had arrived at a design meeting the specification using resources extremely efficiently and not obeying conventional design rules. Cells were influencing neighbouring ones in order to generate complex dynamics within the circuit which resulted in the desired behaviour at the output. Evolution blindly varied the design selecting fitter behaviour, no matter if this came about because of normal cell operation or because of subtle interactions at the physical level. The subtle interactions used were also particular to that chip since the same configuration on another chip did not produce tone discrimination.

### 2.1.3 Genetic Algorithm model – A continuing SAGA

Traditional evolutionary algorithm (EA) (Holland, 1975; Koza, 1992) implementations start with a large initial random population and evolve it until it genetically converges. The reasoning is that building blocks of high fitness present in the initial random population are combined to form increasingly fit genotypes through sexual cross-over. Most of the genetic information present in the final solution evolved in this manner was present in the initial population. Even though there is no reason why EAs should faithfully imitate nature, it is interesting to point out that sexual cross-over usually only works between individuals of the same species, and that natural evolution is more like an open ended process in which genetically quasi-converged species drift and “explore” the “search space”.

Species Adaptation Genetic Algorithms (SAGA) (Harvey, 1992b) theory was presented as a GA model in which a genetically semi-converged species adapts in an open ended evolutionary process. During this process mutation is the main source of genetic diversity and selection acts as a force of genetic convergence keeping the population semi-converged. Cross-over is still productive in combining the useful genetic material of two similar individuals but is not as important as in the classic model described above. Most genetic information present in fit individuals evolved using SAGA theory will have been “discovered” by mutation, which acts as a tinkerer incrementally improving solutions. SAGA theory also defines the optimal mutation rate as one affecting behaviour per genotype. SAGA theory is suitable for experiments using incremental evolution, in which complex behaviour is evolved by imposing incrementally complex selection pressure so as
to “guide” evolution through a viable path in the landscape. There are situations in which it would be almost impossible to define a complex goal using a single fitness metric for evolution to optimise. Traditional GAs do not cope well with incremental evolution as by the time solutions for the first fitness function have been found genetic diversity is lost and no more evolution is possible. A species in a SAGA model can be taken as it is into the next stage of evolution and will keep adapting to the new fitness function selection pressure just like it did with the old one. A visual representation of a semi-converged species exploring the search space can be found in Fig.2.3.

The search space and the fitness function define the fitness landscape, a conceptual metaphor in which each point in the landscape corresponds to a genotype in the search space and its height is defined by the fitness of the genotype. Two points are said to be neighbours if there is a genetic operation (eg. mutation) converting one to the other. Some fitness landscapes contain many individuals with the same fitness which when connected define ridges. These ridges are also called neutral networks. A semi-converged species can drift along such a network and then find a higher fitness network on which it will “jump on” and continue drifting. The presence of neutrality helps the SAGA model and could mean that no local optima are defined in the landscape so there would always be a neutral path to a fitter individual. A (1+1) ES is an extreme implementation of this process. A (1+1) ES is like a GA with no cross-over, a population size of two and one elite such that the individual selected for reproduction is the fittest, or the non-elite if they are equally fit. It can also be thought of as one individual, the elite, from which a mutant is generated, and the mutant replaces the elite if it has better or equal fitness. This algorithm is a stochastic hill-climber on the fitness landscape which drifts along neutral networks. It is rather like a blind man climbing a mountain by probing the ground with his stick at random directions and walking if its higher or level. Barnett (2001) presents the Netcrawler, an algorithm based on a (1+1) ES for which an optimal mutation rate is calculated, and shows that it is an optimal strategy for searching certain type of landscapes with high neutrality.

Spatially distributed GAs (Muhlenbein et al., 1991; Husbands, 1994) limit genetic convergence by establishing a spatial topology so that fit individuals can only replace less fit ones which are a short distance away. They have experimentally been shown (Husbands, 1994) to produce more consistent results, most likely due to the increased genetic diversity helping to avoid stasis at locally optimum regions of the fitness landscape. They also allow for simpler parallelisation across several computers as interaction between spatially distant individuals is reduced.

There are numerous examples (Thompson et al., 1996; Harvey and Thompson, 1997; Miller et al., 2000; Vassilev et al., 2000; Layzell, 2001; Torresen, 2003; Zebulum et al., 2003) of GAs based on ideas from SAGA theory being used successfully for hardware evolution. It must be noted that “GA based on ideas from SAGA” is here meant as “a GA with a small population size which is allowed to continue evolving after genetic convergence and for which mutation rate is roughly one affecting behaviour per genotype”. There are also examples (Thompson and Layzell, 2000; Layzell, 2001) of a (1+1) ES being used effectively in EE.

A (1+1) ES will be used in chapter 3 due to its small hardware implementation and will be further justified in that chapter. For the rest of the work in this thesis a GA will be used with an automatically shifting of fitness function pressure in order to produce incrementally complex behaviour. The fact that SAGA theory is suitable for such incremental evolution processes and that many EE experiments have used it successfully are the reasons why it is here adopted to the extent that a small population will be allowed to continue evolving after it is genetically converged. Discussion will now turn to the GA used in chapter 5 to evolve digital circuits. The fitness landscape defined by digital circuit evolution is discretised as a Manhattan skyline and will contain many genotypes with equal fitness, many of these defining neutral networks. A GA will tend to drift along these networks as mutants with equal fitness get selected and produce new neutral mutants. The GA used encourages drift by ensuring that an individual copied from a previous generation (such as an elite) always gets a lower rank position than a new individual (freshly mutated or crossed-over) with equal fitness. This will make selection of old individuals less likely and make
Chapter 2. Context

The GA drift faster raising the probability of finding a higher fitness neutral network. The GA will also be distributed across multiple computers (as in §4) using a spatially distributed GA. The island model (Muhlenbein et al., 1991) is used so that each island on a computer contains one semi-converged population. Migration of individuals between islands is limited so that the distributed evolutionary process maintains several different fit species at any time. This is done to avoid convergence of the distributed evolutionary process to one species because the fitness function may not actually be a perfect measure of distance to a solution, i.e. one of the less fit species may be closer to a solution that the fitter species are, and because the presence of several species helps to avoid stasis of the whole cluster at local optima and produces results consistently. Thus it is ensured there are several semi-converged species exploring different areas of the search space of high fitness. These species may split after migration and also become extinct (Fig.2.3).

2.2 Reliable Electronics – The Problem

Hardware fails due to environmental hazards, misuse and aging. The problem of designing highly-dependable hardware systems is a large research area (Pradhan, 1986; Wakerly, 1978; Sellers et al., 1968) containing several disciplines which tackle the problem at different scales and from different angles. Hardware failure rates are increasing as feature sizes decrease making electronics more sensitive to ions and radiation. Hardware is increasingly integrated into our society to the point that its failure could cost human lives and great capital loss, e.g. transport, medicine, communication, military, aerospace. Both higher fault rates and higher failure cost stress the need for reliable electronics more than ever.

The design of circuits which operate reliably in outer space is specially hard due to the large amount of charged particles and radiation present outside the protection of the Earth’s atmosphere. The classical approach uses modular redundancy (Neumann, 1956; Avizienis and Kelly, 1984). As FPGAs are increasingly used in space applications several researchers have turned their attention to fault mitigation on FPGAs deployed in space. Most of this work deals with transient fault mitigation (Carmichael et al., 2001; Sturesson and Mattsson, 2001; C.Yui et al., 2003) and some at permanent fault mitigation (Lach et al., 1998; Yu and McCluskey, 2001b; Abramovici et al., 2001).

Self-checking (SC) circuits increase hardware dependability by guaranteeing that erroneous behaviour will be accompanied by an error signal. Digital combinational and sequential SC circuits are usually synthesised by generating an encoded output function module such that faults break the code, and then checking that output belongs to the code using a checker module. Several automatic synthesis methods of SC circuits have been proposed (Jha and Wang, 1993; De et al., 1994; Touba and McCluskey, 1997; Zeng et al., 1999) and often produce fully checking circuits with less overhead than that required for duplication and comparison. Other methods such as (Mohanram and Touba, 2003) have focused on reducing the overhead cost required for SC – a large barrier to commercial adoption – by providing only partial probabilistic error detection.

Analog SC circuit design methods have not reached the same level of maturity as in the digital domain. Methods do exist (Vinnakota and Harjani, 1994; Chatterjee et al., 1996; Arabi and Kaminska, 1996; Lubaszewski et al., 2000) but are only applicable to a certain class of circuits or require significant effort from a human designer.

2.3 Evolutionary Electronics and Reliability

Research has been done in the EE field to provide reconfigurable devices with self-repairing capabilities when subjected to permanent faults. Embryonics (Mange et al., 1996) is a biologically inspired architecture in which a section of a design subjected to a fault can grow itself into another area of the chip. Macias and Durbeck (2002) developed a similar version using Supercells to autonomously build and self-repair circuits. Vigander (2001); Lohn et al. (2003a) have used a GA to search for an FPGA reconfiguration restoring functional behaviour after a fault. Keymeulen
et al. (2000a) did the same with a field programmable transistor array (FPTA). All these techniques suffer from the impracticality that the repair mechanism is several orders of magnitude larger than the circuit under repair.

Some EE research has aimed at improving error detection techniques. Immuno-tronics (Bradley and Tyrrell, 2001) characterises normal FSM behaviour and then signals when behaviour deviates from the norm. Shneier and Yao (2003) have studied how to reduce common mode failures in module redundant systems by using a Genetic Algorithm. Thompson (1995) studied the fault tolerant properties of evolved robot controllers.

### 2.4 Position of the thesis within the fields

Within EE, it firstly touches the subject of autonomous self-repair through reconfiguration by suggesting a practical implementation of achieving this with a GA. Secondly it presents a practical application for EE: the design of SC circuits. Next, it presents methods such as fitness functions, genetic operators and ways of transferring an evolved design to a different technology, which belong to the EE field as a whole as they are applicable to the evolution of circuits in general. The thesis also presents a system for distributing any evolutionary process across the Internet which is also useful to anyone within the evolutionary computation field.

Within the field of autonomous fault mitigation in space deployed FPGAs, this thesis proposes a new method for repairing FPGA configurations subjected to permanent local damage which fits in with current techniques for mitigation of transient errors. The presented method is evaluated more thoroughly than previously proposed methods.

Within the digital SC circuits field, the thesis puts forward a new automatic synthesis method for combinational and sequential SC circuits. This method differs from others because it does not limit the structure to the traditional function-checker decomposition. The method also provides the best overhead figures for all benchmarks attempted. Within the field of analog self-checking the thesis proposes the first (untested) automatic synthesis method for analog self-checking circuits.

### 2.5 Thesis Structure

Chapter 3 will present and evaluate an architecture to improve the reliability of space-deployed FPGA-based systems in which an evolutionary algorithm is used to search for healthy configurations in the field. The rest of the thesis will apply evolution to the design of electronic circuits with CED. The processing power required for such an evolutionary design process grows roughly exponentially with problem size and chapter 4 presents the architecture used to distribute the evolutionary design process across the Internet and how such a free grid computer could be created. Chapter 5 presents and evaluates a method to evolve digital circuits with CED. These increase reliability by signalling erroneous operation. Chapter 6 presents a method to evolve analog circuits with CED. If evolved analog circuits are ever adopted by industry, it is crucial for them to transfer from the medium they were evolved in to a different medium. Chapter 7 describes an experiment in which a circuit evolved on one medium is analysed and transferred to a breadboard. Finally, chapter 8 presents the conclusion.
Chapter 3

Jiggling: Long survival of FPGA systems through evolutionary self-repair

This chapter will propose and evaluate an architecture to increase the reliability of space-deployed FPGA-based systems using an evolutionary algorithm to repair configurations of modules subjected to faults.

Motivation

Field Programmable Gate Arrays (FPGAs) are gaining common use in space missions because of their low cost and flexibility. FPGAs are susceptible to transient errors caused by Single Event Upsets (SEUs) and techniques have been developed to mitigate them. During deep space exploration missions lasting decades the possibility of permanent local damage in electronic systems due to aging effects should not be discarded. FPGAs offer the capacity to be repaired by reconfiguring logic around permanent faults. Techniques for doing this have been proposed but introduce a prohibitively large single point of failure in the repair mechanism. This paper explores the possibility of trading repair mechanism overhead for repair time. Jiggling is an extremely low overhead stochastic repair mechanism small enough that a pair could mutually repair each other. Jiggling is suitable to be used to protect the most mission critical modules from permanent local damage providing 0.99 reliability for a period 48 times longer than TMR/Simplex does while incurring no performance penalty nor interruption of normal operation. Availability analysis of several benchmark circuits protected with Jiggling is provided together with analysis of the comparative effectiveness of several architectural variations.

3.1 Introduction

Reconfigurable hardware devices such as Field Programmable Gate Arrays (FPGA) are being increasingly used in space applications because they allow cheap and fast production of prototypes and final designs in low volume. Buggy designs can be fixed post-deployment, and the same hardware can be used to perform various tasks – some possibly unforeseen – over the duration of a mission. The National Aeronautics and Space Administration (NASA) have decided on a paradigm change toward smaller and cheaper spacecraft that use cost-effective Commercial Off-The-Shelf (COTS) components (Brownsworth and Morris, 2003) such as FPGAs. In their own words (Sampson, 2002) the overall objective is to enable insertion of COTS into NASA and other governmental agency space systems at minimal risk. The use of COTS components such as FPGAs is becoming common-place in space applications and other industries (Albert and Brownsworth, 2002).
SRAM based FPGAs deployed in space are susceptible to radiation hazards, most commonly (Wang et al., 1999; MacQueen et al., 1999; Fuller et al., 2000; Sturesson and Mattsson, 2001; C.Yui et al., 2003) Single Event Upsets (SEU) not causing permanent damage and mitigated through techniques such as Configuration Readback and Scrubbing (Carmichael et al., 2001; Sturesson and Mattsson, 2001; C.Yui et al., 2003). Total dose exposure, or a Single Event Functional Interrupt (SEFI), may cause catastrophic damage to a chip, unless it is subjected to an annealing process (MacQueen et al., 1999; Sturesson and Mattsson, 2001). The research described above addresses transient and global faults and does not focus on permanent local damage.

Even though permanent local damage has not been commonly observed in radiation tested FPGAs there are several reasons why it should not be ignored. Cases of Single Event Latch-up (SEL), which may cause local permanent damage by inducing high current density, have been reported (Straulino, 2000). Moreover, some SEUs in configuration data cannot be mitigated without a full reset of the chip which may not be possible for a mission-critical module, thus manifesting themselves as permanent local faults. Also, radiation testing in the laboratory is not 100% faithful to space conditions and does not last as long as a mission. In fact, no FPGA has ever been tested for more than 15 years, while NASA is planning 100 year missions for deep space exploration. Long usage of a device could lead to permanent local damage through electromigration or other aging effects. Failures such as a short with initially very high resistance or a tunnelling open effect may also only manifest themselves a considerable time after deployment. It would be unwise engineering to assume that permanent local damage to FPGA cells would not occur during long space missions exposed to extreme environmental conditions and radiation. Space missions are not the only deployments that can benefit from strategies dealing with permanent local faults, although they are particularly needy of autonomous onboard repair since communication with earth is low bandwidth and high latency. Radiation and aging effects are also encountered at sea-level (Evans-Pughe, 2003) and may be problematic for inaccessible systems where component replacement is not feasible.

N Module Redundancy (NMR) uses $N$ copies of a module, where $N$ is usually odd, with a voting system to tolerate faults at up to $\frac{N-1}{2}$ modules. Triple Module Redundancy (TMR) or TMR/Simplex are currently widely used to mitigate faults and are considered to have “saved” several space missions. A TMR (Neumann, 1956) system has three copies of a module and uses a voting system on their outputs so that the final output is an agreement between at least two modules. A TMR/Simplex defaults to a single module once one module fails, thereby increasing reliability. Even given a meagre fault rate of 1 per year this mitigation technique would not be effective for long missions. TMR+Scrubbing (Carmichael et al., 2001; Sturesson and Mattsson, 2001) provides fault tolerance as above and wipes out SEUs in FPGA configuration data by regular reprogramming. Configuration readback (Fuller et al., 2000) is able to locate configuration errors and fix them by partial reconfiguration. These schemes are only as good as a TMR system in the presence of permanent faults and rely on a golden (unbreakable) memory to contain configuration data. Latched user data in sequential designs can be protected with state recovery schemes (Yu and McCluskey, 2001a).

Lach et al. (Lach et al., 1998) proposed a tile based approach for reconfiguring an FPGA design to avoid permanent faults. This approach only tolerates limited faults per tile, assumes the existence of a golden memory holding precompiled configurations and a golden fault location mechanism. The repair mechanism is not transparent since the tile is off-line during the reconfiguration process, which may rule out repair of a mission-critical module. A similar approach by Yu et al. (Yu and McCluskey, 2001b) likewise requires a set of golden pre-compiled configurations and a golden fault diagnosis system hosted on an extra FPGA. The Roving STARS (Abramovici et al., 2001) approach to permanent fault mitigation in FPGAs proposes a self-testing column and row to shift itself across the FPGA. Its fault detection latency is of around 4000 cycles, and requires constant reconfiguration of the FPGA and is therefore a constant power drain. It relies on a golden micro-processor (the ‘TREC’) large enough to perform timing analysis, contain a database...
with various designs and faults, a fault location algorithm and a place and route system. It requires more than 420K of golden memory for storage. A final cost is that the system clock is stopped regularly.

A biologically inspired approach termed Embryonics (Mange et al., 1996) requires large amounts of overhead due to its architecture and in one version the entire chip design is held in a golden memory inside each cell. Zebulum et al. (Zebulum et al., 2003) have used a Genetic Algorithm (GA) to repair analog designs on a Field Programmable Transistor Array. However they assume a golden ‘SABLES’ system composed of a DSP, memory for a full population, and a fitness evaluation mechanism which in some cases requires a fully functional copy of the circuit being repaired in the first place. They also only attempt the repair of a very limited fault set. Thompson (Thompson, 1995), Vigander (Vigander, 2001) and Lohn et al. (Lohn et al., 2003a) have applied a GA to repair FPGA designs, however still assuming a golden GA module with a full population and fitness evaluation mechanism.

Most FPGA fault mitigation techniques mentioned so far suffer from the Repairing the Repairer dilemma in which a new single point of failure assumed unbreakable is introduced in the mitigation mechanism. This is especially awkward when the mitigation mechanism assumed unbreakable is larger than the breakable system itself. A repair module small and simple enough to be itself repaired, would offer an obvious advantage. This chapter will describe such a low overhead fault mitigation mechanism sufficiently small that a pair could mutually repair each other, relying on no golden single point of failure. Section 2 will introduce the TMR+Lazy Scrubbing+Jiggling architecture for transient and permanent fault mitigation, section 3 will lay out reliability analysis for various circuits and section 4 will study the repair effectiveness of different fitness functions. Section 5 will compare variations on the Jiggling architecture, section 6 will explore several trade-offs between repair mechanism overhead and time, section 7 will study the link between population fault tolerance and Jiggling, and section 8 will provide some conclusions.

3.2 TMR+Jiggling

The proposed mechanism is based upon the TMR approach. The system is constantly kept on-line by the fault tolerance of TMR which also votes out SEUs affecting user data. SEUs to configuration data are either mitigated either through ‘Lazy Scrubbing’ or through ‘Flatrolling’, both of which are novel techniques described below. ‘Jiggling’ repairs permanent faults by using the two healthy modules to repair the faulty one. Once Jiggling repair is complete three healthy modules are again available and the system is capable of repairing the next permanent fault as in Fig.3.4. Self-repair can proceed until spare resources are exhausted.

After transient fault mitigation methods are introduced, the chapter deals entirely with permanent fault mitigation through Jiggling. This first study will deal entirely with combinational circuits, leaving sequential ones for future work (see discussion §3.3.1).

3.2.1 Transient SEU Mitigation

Two alternative architectures for correcting configuration errors caused by transient faults will be introduced, both of which are compatible with Jiggling.

Lazy Scrubbing

Traditional Scrubbing reconfigures an FPGA regularly from an external memory. To reduce power consumption: instead of reconfiguring every module in the TMR system regularly, a module will only be reconfigured when its output is different to the other two. To remove a single point of failure the approach drops the need for a golden memory holding the module configuration. Instead, the configuration is read from all three modules and a majority vote is taken of this data (taking offsets into account) to reconfigure the faulty module. Lazy Scrubbing draws less power, requires less overhead and has a much smaller single point of failure than traditional Scrubbing.
The evolutionary Jiggling repair process will lead to modules having different configurations once recovery from the first permanent local fault is achieved. At this point in the mission, Scrubbing can no longer be used. However with a low permanent fault rate, the low overhead for Lazy Scrubbing would still be worthwhile to ensure tolerance against a high SEU rate during the initial phase of the mission. Once Scrubbing is disabled, the Jiggling mechanism will still recover from SEUs in FPGA configuration, although with a higher latency.

Flatrolling
As Lazy Scrubbing, Flatrolling only begins when the output at one module disagrees with that at the voter. It assumes, and this is thought to be the case, that a SEU causes one bit in the module configuration to be flipped at a time. Flatrolling works by flipping every bit in turn and checking if normal behaviour has been restored. Flatrolling also uses less power than Scrubbing because it performs reconfiguration on-demand instead of regularly. It also reduces the single point of failure since an external healthy configuration is not required. Flatrolling has the advantage over Lazy Scrubbing that it is still effective after the first Jiggling repair has taken place since all modules need not have the same configuration. If Flatrolling fails the fault will be assumed to be permanent and Jiggling will begin. Jiggling is capable of repairing from multiple errors in the configuration.

3.2.2 Jiggling Architecture Overview
Figure 3.1 shows the simplest setup of a Jiggling system with three copies of module $M$ and a repair module containing the voter and a minimal implementation of a GA. TMR can be applied at many levels. Based on experiments to date, $M$ should be under a thousand gate equivalents to make the repair process feasible. One GA circuit could service many TMR systems, all residing on a single FPGA.

A module is considered to have local permanent damage if it fails to repair after Scrubbing. This initiates the Jiggling repair mechanism which uses the remaining functional modules as a template of desired behaviour to guide a (1+1) Evolutionary Strategy (ES) Schwefel and Rudolph (1995) – the minimal expression of a GA – towards a solution which avoids or exploits\(^1\) the faulty element. This repair process does not require the system to go off-line since the two healthy modules are still driving the majority voter. Spare resources are allocated in each module. Mutations inserted by the GA are single bit flips in the FPGA configuration stream of the faulty module.

Given that permanent faults in FPGAs are not very frequent (indeed completely ignored by Xilinx Lesea and Alfke (2003)) we can trade overhead and the knowledge contained therein for time and allow blind variation and selection find the knowledge required to complete the repair.

Section 3.2.3 will describe the evolutionary algorithm used during repair, the fitness evaluation of the faulty module, the hardware nature of mutations and how the repairer is repaired. Section \(^1\)eg. a short circuit at a look-up table could be used as routing
3.2.4 will cover the repair mechanism control loop, its hardware implementation and overhead analysis. Section 3.2.5 will describe the simulator used to collect repair time statistics for various benchmark circuits while section 3.2.6 will present the probability model used for availability analysis.

3.2.3 Specification

Evolutionary Algorithm used during Repair

A (1+1) ES has one elite individual and one mutated version of it. If the mutant is equal or superior it will replace the elite. Otherwise, another mutant is generated. This strategy has been applied successfully to hardware evolution Thompson and Layzell (2000) and has been considered Barnett (2001) to be an effective strategy for exploring fitness landscapes with high neutrality, such as those of digital circuits. It was mainly chosen here for its simplicity and allowance of a small hardware implementation.

Given that several bit flips may be required to restore healthy behaviour to the module with a permanent fault, an exhaustive search of all possible reconfiguration would blindly iterate through a space of size $2^b$ where $b$ is the configuration length. This space is larger than the number of atoms in the universe for the circuits used in this work. A (1+1) ES is a hill-climbing random walk within the configuration space guaranteed to move to a fitter (as defined in Fitness Evaluation below) or equal configuration at each step. The (1+1) ES is not guaranteed to find a solution in time. Yet faced with a stochastic fault source, no system is capable of guaranteeing survival. A study of the probability of survival for a Jiggling system is provided in §3.3.

A (1+1) ES is brittle in the presence of noise because if a bad mutant gets a lucky high fitness it could replace a superior elite which would then be lost. Noise is present in the evaluation of circuits on an FPGA when they are not constrained to be combinational. Sequential circuits behave differently under changes in input pattern ordering and gate delays which may vary with environmental conditions.

To discourage evolution from discarding a good elite, a History Window method is used: the last $H$ accepted mutations are stored so that if at any moment the current elite’s fitness is lower than the previous one’s, all $H$ mutations are reverted. By rolling back after encountering individuals with noisy fitness, evolution is discouraged from exploring areas of the fitness landscape encoding sequential circuits. If $p$ is the probability of a lucky high noisy evaluation, then the probability a sequential circuit has been lucky $H$ times varies exponentially as $p^H$. Thus, the larger $H$ is, the higher the chance that the circuit reverted to is stable.

Reconfiguration as the Hardware Nature of Mutations

Mutations are single bit flips in the configuration stream of the faulty module $M_f$. Modules are always allocated $2^C$ addresses of configurable logic blocks (CLB) so that the address of a mutated block can be simply a randomly generated $C$ bit number. If a circuit only requires a fraction of these CLBs, the rest are allocated as spare resources for repair. A mutation may affect any part of the CLB such as a Look-up Table (LUT) or the routing. Ease of partial self-reconfiguration and robustness to random configuration bit flips make some FPGA architectures more suitable than others.

Fitness Evaluation

Since normal circuit operation is not interrupted during repair of $M_f$, circuit fitness evaluation is done on the fly using the inputs being applied during normal mission operation.

The simple correlation fitness function described in §3.4 is used to calculate fitness. This requires collecting $m_q$ and $a_q$ for each output wire $q$. Let $I$ ($Q$) be the number of circuit input (output) wires, then there are $2^I$ possible inputs, and $m_q = \sum_{x=0}^{2^I-1} y_q(x)y_q(x)$ is the number of inputs for which both faulty module response $y_q(x)$ and voter output response $y_q'(x)$ are high. $a_q = \sum_{x=0}^{2^I-1} y_q(x)$ is the number of inputs which generate high output at the faulty module. The $m_q$ and $a_q$ are each collected in a separate $I$ bit register per output $q$ with $2Q$ such registers in total.
Chapter 3. Jiggling: Long survival of FPGA systems through evolutionary self-repair

and they are updated – i.e. incremented or not – when an input is encountered for the first time. Evaluation ends when all inputs have been encountered. If an input is applied more than once during evaluation, the \( m_q \) and \( a_q \) are only updated the first time. Fitness evaluation also requires the calculation of constants \( d_q \) at the start of the repair process. \( d_q = \sum_{x=0}^{2^l-1} y'_q(x) \) is the number of inputs producing output \( q \) at the voter to go high. Fitness is then calculated using equation 3.1 which is a version of the statistical correlation metric with a simpler scaling.

\[
f = \sum_{q=0}^{Q-1} \left( \frac{2^l m_q - a_q d_q}{2^l d_q - d_q^2} \right) \quad (3.1)
\]

This fitness evaluation rewards configurations with behaviour similar to that of the healthy modules. By hill-climbing the landscape defined by this fitness function, the (1+1) ES will be attracted to configurations whose behaviour is increasingly similar to the healthy modules’, possibly finding one with fully healthy behaviour.

**Repairing the Repairer**

Since the repair module is small, it would be feasible to be repaired with the same strategy, once this method has been adapted to sequential circuits. Each repair module could be itself tripled as in Fig.3.2 and repaired by another repair module. Both repair modules would then be in charge of repairing multiple systems. This is viable in the case that permanent damage is infrequent enough that systems would not collect too many faults while awaiting their repair module to attend them.

### 3.2.4 Hardware Implementation

**Jiggling Repair Cycle**

The (1+1) ES with History Window control loop for repairing faulty module \( M_f \) is described below in pseudocode. \( F_c \) and \( F_p \) are registers holding the current and previous fitness values. \( S \) is a bitwise storage indicating which vectors have been encountered. \( D_q, M_q \) and \( A_q \) are registers storing \( d_q, m_q \) and \( a_q \).

1. Set register \( F_p = 0 \) and perform dummy evaluation collecting output response to calculate the constants in the \( D_q \).
2. Evaluate Elite: collect output response into the \( M_q \) and \( A_q \) until all inputs have been encountered.
3. Calculate fitness \( f \) as described in §3.2.3 and store in \( F_c \). If \( F_c = F_{\text{MAX}} = Q \) then stop.
4. If \( F_c < F_p \) then revert all mutations in the history shift register \( SR_{\text{H}} \) and go to step 1.
5. Shift the value of \( F_c \) up into \( F_p \).
6. Insert new random mutation \( m \) in \( M_f \).
Figure 3.3: Possible Jiggler minimal GA implementation showing data flow directed by the Control FSM as in the algorithm in §3.2.4.

8. Calculate fitness \( f \) as described in §3.2.3 and store in \( F_c \). If \( F_c = F_{\text{MAX}} = Q \) then stop.

9. If \( F_c < F_p \) revert mutation \( m \).

10. Else shift the value of \( F_c \) up into \( F_p \) and push \( m \) onto \( SR_H \).

11. Go to step 2.

The control logic can be implemented as a Finite State Machine (FSM) with 16 states. In the case that an incorrect configuration gets a lucky perfect score, the repair process will be resumed as soon as its behaviour is different from the voter.

**Structure**

Figure 3.3 shows a possible hardware implementation for the Jiggler. The voter provides system fault tolerant output \( Y \) as the majority of the module outputs \( Y_0, Y_1, Y_2 \). It also provides faulty module index \( f \) and output \( Y_f \) to the minimal GA.

Given that the circuit being repaired has \( I \) inputs and \( Q \) outputs then \( S \) needs \( 2^I \) bits storage and the \( D,M,A \) are \( 3Q \) registers. A shift register chain \( SR_F \) of size two holds \( F_c \) and \( F_p \). A 8 bit floating point multiplier and adder unit is used for fitness calculation. Arithmetic operations must be ordered carefully – eg. by performing subtractions first – to avoid loss of precision. A shift register chain \( SR_H \) of size \( H \) will store the mutation history window with the addresses of the last \( H \) mutated configuration stream bits. A Random Number Generator (RNG) will generate the random mutation address \( m \). A reconfiguration unit will flip the configuration stream at a particular address and initiate the partial reconfiguration procedure. Its operation will depend on the reconfiguration mechanism of the FPGA architecture chosen. The Control FSM will implement the control loop coordinating the activity of all other modules.

**Overhead Analysis**

Given a circuit has \( I \) inputs and \( Q \) outputs, \( 2^I \) bits storage are needed for \( S, 3QI \) for the \( D,M,A \) registers, and \( 3 \times 8 \) for the \( F \) registers. If the the address offset of a configuration bit within a module requires \( A \) bits, \( H \times A \) bits storage are needed for \( SR_H \). The control module can be implemented as a 16 state FSM and might require 4 latches and 20 four-input LUTs. The \( A \) bit RNG could be implemented as a linear feedback shift register with roughly \( A/18 \) LUTs and \( A \) latches. The reconfiguration module’s size would depend on the FPGA architecture and could vary between 3 and 30 LUTs and some latches. For this analysis it is assumed it requires 15 LUTs and 15 latches. The 8 bit floating point adder and multiplier could be implemented with 16 latches and 32 LUTs as speed is not an issue. The voter could be implemented in roughly \( Q \) LUTs. Given a CLB offset address within a module needs \( C \) bits we need \( Q \times C \) bits for the reconfigurable module output addresses.
Chapter 3. Jiggling: Long survival of FPGA systems through evolutionary self-repair

The largest Jiggler required by all benchmarks in Table 3.1 is that required by bw: \( I = 5, Q = 28, H = 32, A = 7 \) so \( 32 + 420 + 24 + 224 = 700 \) memory bits are needed, plus \( 20 + 15 + 32 = 67 \) LUTs and \( 4 + 15 + 16 = 35 \) latches but since each 4 input LUT can hold 16 bits, 141 CLBs are required in total. This Jiggler would be capable of protecting multiple subsystems implementing any of the benchmarks in Table 3.1 thus amortising the overhead per subsystem. It could itself also repair and be repaired by other Jigglers §3.2.3.

3.2.5 Simulated Jiggling

The Jiggling method was evaluated by collecting repair time statistics from a simulated model.

FPGA model

Various FPGA architectures Xilinx (2003); Atmel (2002); Triscend (2002); Lattice (2003); Altera (2003), some of which have been deployed in space missions, can be simplified to a model where each CLB holds one LUT and one D-Latch. Routing between such CLBs is limited yet can be assumed universal Lohn et al. (2003a) for small circuits such as those dealt with in this work. This study of the Jiggling approach tackles combinational circuits only so the FPGA model adopted uses four-input LUTs and no latches. It is assumed it is not complex to turn off all latch functionality for a given area of an FPGA.

Simulator Characteristics

The simulator used is a simple version of an event driven digital logic simulator in which each logic unit is in charge of its own behaviour when given discrete time-slices and the state of its inputs. Routing is considered unlimited so any unit can be connected to any other allowing recurrent connections inducing sequential behaviour, so care must be taken to update all units ‘simultaneously’. This is achieved by sending the time-slices to the logic units in two waves: the first to read their inputs and the second to update their outputs. During each evaluation, circuit inputs were kept stable for 30 time-slices and the outputs were read during the 5 last time slices.

Gate delays are simulated in time-slice units and are randomized at the start of each evaluation with a Gaussian distribution with \( \sigma = 0.1 \) and a \( \mu \) varying between 3 and 6 thus simulating a probe subjected to a changing environment. \( \mu \) increments, decrements or keeps its value within the \([3, 6]\) range every \( t_d \) simulated generations where \( t_d \) is itself taken from a Gaussian distribution \( (\mu = 270000, \sigma = 90000) \). For the scenario studied in this chapter these statistics would translate to the mean of the frequently randomized gate delays changing roughly every few seconds.

The Stuck-At (SA) fault model was chosen as an industry standard providing a fairly accurate model of the effects of radiation hazards at the gate level. SA faults can be introduced at any of the logic units of the simulator simply by setting its output always to 0 or 1.

FPGA configuration stream encoding

As mentioned earlier, mutations are performed on the section of the FPGA configuration stream which encodes the faulty module. During simulated evolutionary repair, the GA deals with linear bit string genotypes which are equivalent to the simulated FPGA configuration stream. As mentioned in §3.2.3, there are \( 2^C \) addresses available. The last \( I \) addresses are assigned to circuit inputs while the remaining \( 2^C - I \) refer to LUT units within the module. \( C \) bits are required per address. The first \( Q \times C \) configuration bits – where \( Q \) is the number of circuit outputs – encode the addresses from which module outputs will be read by the voter. This simulates mutations to the configuration memory controlling routing to the module outputs. The rest of the stream is divided into \( 2^C - I \) sections, one for each LUT. Each of these sections contains 16 bits encoding the LUT and \( 4 \times C \) bits encoding where the inputs of this LUT are routed from. LUT addresses are assigned in the order they appear in the configuration stream.

Evaluation Procedure

In order to mimic normal mission operation during circuit evaluation, the order in which test vectors are applied for the 30 simulated time steps is randomised. All analysis in this chapter
assumes the availability of a fresh input on every clock cycle. However during simulation each vector is applied only once during each evaluation to reduce computational effort. This does not affect the validity of the simulation because subsequent applications of previously seen inputs are ignored during a real Jiggling evaluation. Fitness is calculated using equation 3.10.

**Collecting Repair Time Statistics**

Repair time information is required to perform a reliability analysis of the Jiggling approach. Since all modules are equal repair statistics from a single module convey the information required. The time taken to repair the first fault when all spares are available will clearly be different to that of repairing the \( n^{th} \) when \( n-1 \) successful repairs have already taken place probably allocating at least \( n-1 \) spares.

To collect repair time information, \( W \) random fault sequences of length \( 5s \) – where \( s \) is the initial number of spare LUTs – are generated. \( 5s \) was chosen so that it would be all but impossible for repairs to keep succeeding after this amount of faults had accumulated. For each of these \( W \) sequences, faults are inserted into the simulated module in order and the number of generations of \((1+1)\) evolution required to arrive at a fully functional configuration of the faulty module is recorded. \( H \) consecutive generations with a fully functional elite must elapse before the module is considered repaired to avoid lucky solutions. Each time such a repair process is completed the next fault in the sequence is inserted (the previous ones still present) and the generations till repair are again counted. The repair time of the \( i^{th} \) fault in the \( j^{th} \) sequence will be referred to as \( r_{ij} \). If a fault is inserted in a LUT which is currently a spare then 0 generations are required for repair. If 4.32 million (M) simulated generations elapse during the \( i^{th} \) repair of sequence \( j \) and the module has not been repaired then the fault sequence is aborted and the repair time of all unrepaired faults in the sequence is set to \( \infty \) so that \( \forall a \geq i r_{aj} = \infty \). This is done to limit the amount of CPU time used for simulation and will significantly skew the statistics pessimistically since a long-term mission may have months to repair a permanent fault.

### 3.2.6 Probability Model

Figure 3.4 shows a Markov model with the three states the Jiggling system can be in. It begins its life fault-free in the healthy state. When a fault arrives at module \( M_f \) it moves to the repair state. If during repair another fault arrives in \( M_f \) it will stay in this state. If during repair a fault arrives in one of the other modules reaches the fail state. In this state the voter may encounter three different output vectors for the same input vector and could revert to Simplex mode choosing any \( M_{k \neq f} \) with a 50% chance of choosing the functioning module. However, if two modules are failing they may fail under different input vectors so the voter is always correct and there is still enough information to repair both of them. This is especially relevant given that \( M_f \) has fitness 0.99 during most of the repair process with most outputs correct under most input vectors. For the purposes of this work these eventualities will be ignored making for a pessimistic failure analysis. Hence, the system will only be considered to go back to the healthy state when \( M_f \) is repaired before a fault arrives at \( M_{k \neq f} \).

The probability of moving from the healthy to the repair state is dictated by the permanent fault arrival rate and this may be affected by such factors as usage, age and environmental stress. The probability of moving from the repair back to the healthy state will depend on both the permanent
fault rate and the time to repair which is likely to increase as faults accumulate and spares get used up.

Faults affecting the voter and GA module could be dealt with as mentioned in §3.2.3.

Reliability Analysis

The probability model will be laid out in a top-down fashion. Recall from above that the Jiggling system is considered to fail when, during repair of \( M_f \) a fault arrives at \( M_{k\neq f} \). Consider \( P_i \) the probability of the \( i \)th repair within the whole system (after the \( i \)th fault at any module) succeeding. This is the probability that, given \( i-1 \) system repairs were successful, the latest fault at \( M_f \) will be repaired before a fault arrives at \( M_{k\neq f} \). Consider \( N(t) \) the Poisson distribution of permanent local fault arrival during a time period \( t \). Now the probability that the system has not failed before time \( t \), ie. its reliability is:

\[
R(t) = \sum_{n=0}^{\infty} \left( P(N(t) = n) \prod_{i=1}^{n} P_i \right) (3.2)
\]

The outer sum considers all possible fault counts \( n \) during \( t \). For each \( n \) the probability of \( n \) faults occurring during \( t \) is multiplied by the probability of all \( n \) system repairs succeeding. Given a fault rate \( \lambda \) per unit area, and a total area \( A \) for the three modules then \( N(t) \) is a Poisson distribution with parameter \( \lambda At \) so:

\[
P(N(t) = n) = \frac{(\lambda At)^n e^{-\lambda At}}{n!} (3.3)
\]

If \( p_i \) is the probability of the \( i \)th repair on a single module succeeding, the probability \( P_i \) of the \( i \)th system repair succeeding can be calculated:

\[
P_i = \sum_{i=0}^{t-1} P(F(i-1) = r)p_{r+1} (3.4)
\]

where \( P_0 = 1 \) and \( F(k) \) is the probability distribution of the number of previous faults (and repairs) at the module under repair \( M_f \) given \( k \) previous system faults. The sum considers all possible previous fault counts \( r \) at \( M_f \). For each \( r \) the probability of \( r \) faults having occurred at \( M_f \) out of \( i-1 \) system faults is multiplied by the probability of the \((r+1)\)th single module repair succeeding. The probability of exactly \( r \) failures at \( M_f \) out of \( k \) system failures can be calculated:

\[
P(F(k) = r) = \left( \frac{1}{3} \right)^r \left( \frac{2}{3} \right)^k - r \lambda C_r (3.5)
\]

since those \( r \) failures could have happened at any of the \( k \) positions in the fault series and the probability of \( M_f \) failing out of the three modules is \( \frac{1}{3} \).

Calculating \( p_i \)

Recall \( p_i \) is the probability of the \( i \)th repair on a single module finishing before a fault arrives at one of the other two modules. Given that fault arrival is modelled as a Poisson process, the fault interarrival time \( T \) follows an exponential distribution, such that the probability of the next fault in any of two modules occurring later than time \( t_r \) is:

\[
P(T \geq t_r) = e^{-2A_M \lambda d} (3.6)
\]

where \( A_M = \frac{A}{3} \) is module area. Given a set of \( W \) single module repair times for the \( i \)th repair \( r_{ij} \), the probability \( p_{ij} \) of each of these repairs succeeding is \( P(T \geq r_{ij}) \). Provided with a limited sample of \( W \) values of \( r_{ij} \) for each \( i \) the best estimate of \( p_i \) using the frequency interpretation of probability will be:

\[
p_i = \frac{1}{W} \sum_{j=1}^{W} p_{ij} (3.7)
\]

Given the equations above, the experimental data \( r_{ij} \) collected as in §3.2.5 can be used to calculate reliability \( R(t) \).
Table 3.1: Benchmark circuits used to evaluate Jiggling. Columns show benchmark name, number of inputs, outputs and elements, address width, spare count, overhead allocated for spares as a percentage of basic TMR area and how long the imposed 4.32M generation timeout would be at 1Ghz.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>I</th>
<th>O</th>
<th>E</th>
<th>C</th>
<th>S</th>
<th>OH TMR %</th>
<th>Timeout (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm42a</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>18</td>
<td>180</td>
<td>0.69</td>
</tr>
<tr>
<td>cm138a</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>5</td>
<td>16</td>
<td>160</td>
<td>3.03</td>
</tr>
<tr>
<td>decod</td>
<td>6</td>
<td>16</td>
<td>18</td>
<td>5</td>
<td>8</td>
<td>44</td>
<td>3.03</td>
</tr>
<tr>
<td>rd53</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>17</td>
<td>170</td>
<td>1.41</td>
</tr>
<tr>
<td>p82</td>
<td>5</td>
<td>14</td>
<td>36</td>
<td>6</td>
<td>23</td>
<td>64</td>
<td>1.41</td>
</tr>
<tr>
<td>ml1</td>
<td>6</td>
<td>12</td>
<td>30</td>
<td>6</td>
<td>28</td>
<td>93</td>
<td>3.03</td>
</tr>
<tr>
<td>bw</td>
<td>5</td>
<td>28</td>
<td>66</td>
<td>7</td>
<td>57</td>
<td>86</td>
<td>1.41</td>
</tr>
<tr>
<td>rd73</td>
<td>7</td>
<td>3</td>
<td>20</td>
<td>5</td>
<td>5</td>
<td>25</td>
<td>6.64</td>
</tr>
<tr>
<td>misex1</td>
<td>8</td>
<td>7</td>
<td>17</td>
<td>5</td>
<td>7</td>
<td>41</td>
<td>14.65</td>
</tr>
<tr>
<td>5xp1</td>
<td>7</td>
<td>10</td>
<td>36</td>
<td>6</td>
<td>21</td>
<td>58</td>
<td>6.64</td>
</tr>
<tr>
<td>Z5xp1</td>
<td>7</td>
<td>9</td>
<td>43</td>
<td>6</td>
<td>14</td>
<td>33</td>
<td>6.64</td>
</tr>
<tr>
<td>inc</td>
<td>7</td>
<td>9</td>
<td>44</td>
<td>6</td>
<td>13</td>
<td>30</td>
<td>6.64</td>
</tr>
</tbody>
</table>

TMR/Simplex and NMR+S

The Jiggling system reliability under permanent faults will be compared to TMR/Simplex and to NMR with active spare modules. The same per area fault rate will be used in all comparisons and the same total area will be used for NMR+S and Jiggling. Let $A_{SM}$ be the area of the simplex module. Then the reliability of a TMR/Simplex system is Trivedi (1982):

$$R_{TMR}(t) = \frac{3e^{-\frac{A_{SM} \lambda t}{2}}}{2} - \frac{e^{-A_{SM} \lambda t}}{2}$$ (3.8)

and the reliability of an NMR+S system with $N$ modules and $S$ active spares is:

$$R_{NMR+S}(t) = \sum_{i=0}^{N-1} N^i S C_i \left(1 - e^{-\frac{A_{SM} \lambda t}{N}}\right)^i \left(e^{-\frac{A_{SM} \lambda t}{N}}\right)^{N-i}$$ (3.9)

Both expressions ignore faults at the voter (a single point of failure), making the comparison conservative.

3.2.7 Benchmark Circuits

Jiggling will be evaluated by simulating repairs for the benchmarks in Table 3.1 belonging to the MCNC’91 suite. The $OH_{TMR}$ column shows the overhead allocated for spares as a percentage of basic TMR area, the average is 82%. The last column shows how much time the timeout of 4.32M generations would be equivalent to on a real Jiggler. This was calculated by taking into account that on average at least $2^t(1 + \ln 2)$ (Table 3.5) random inputs must be seen per evaluation, that 20 clock cycles would be sufficient between evaluations to count fitness, swap a bit in the configuration stream and perform the required control procedures, and that two evaluations are performed per generation. The clock speed is assumed to be 1Ghz. On average, the simulation gave less than 5 seconds for the stochastic repairs to succeed. Since permanent damage is most likely to happen every 6 months at worst, the timeout heavily skewed statistics negatively as there would be several orders of magnitude more time for repairs to complete.

3.3 Results

Repair time statistics were collected as in §3.2.5 for several combinational benchmarks from the MCNC ’91 test suite found in Table 3.1. History window size $H$ was set to 32. The number of
Chapter 3. Jiggling: Long survival of FPGA systems through evolutionary self-repair

Table 3.2: Fault sequences simulated, average number of accumulated repairs completed per sequence, standard error, time for protected system above 0.99 reliability in terms of mean time between faults for collected data and if all repairs were completed.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$W$</th>
<th>Repairs</th>
<th>Std. Err.</th>
<th>Time $R(t) &gt; 0.99$</th>
<th>Perfect $R(t) &gt; 0.99$</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm42a</td>
<td>37</td>
<td>13.14</td>
<td>0.64</td>
<td>48.6</td>
<td>75.7</td>
</tr>
<tr>
<td>cm138a</td>
<td>41</td>
<td>13.98</td>
<td>0.44</td>
<td>52.8</td>
<td>75.8</td>
</tr>
<tr>
<td>decod</td>
<td>7</td>
<td>0.29</td>
<td>0.18</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rd53</td>
<td>23</td>
<td>34.88</td>
<td>1.91</td>
<td>140</td>
<td>152.1</td>
</tr>
<tr>
<td>p82</td>
<td>8</td>
<td>0.63</td>
<td>0.38</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ml</td>
<td>16</td>
<td>7.29</td>
<td>2.49</td>
<td>9.8</td>
<td>126.6</td>
</tr>
<tr>
<td>rd73</td>
<td>7</td>
<td>17.17</td>
<td>2.01</td>
<td>140.5</td>
<td>159.2</td>
</tr>
</tbody>
</table>

Figure 3.5: Reliability vs. time for cm42. TMR/Simplex (dashed), 8MR (dot-dashed), Jiggling (solid) and Jiggling with every repair successful (dotted).

The imposed timeout of 4.32M generations meant that most repairs either completed before 5 simulated seconds (Table 3.1) or did not complete at all. Given a permanent fault interarrival time in the order of months, reliability would depend on the amount of accumulated repairs that succeeded and would not be strongly influenced by the number of generations in which they repaired. This also means that reliability is not heavily influenced by Jiggling clock speed and can be stated purely in terms of the mean time between fault arrivals ($\lambda A^{-1}$).

Table 3.2 lists the accumulated fault repair count (ie. how many $r_{ij} < \infty$ for each $j$) statistics collected for each benchmark. Once $r_{ij}$ were collected, $p_{ij}, p_i$ and $P_i$ were calculated, providing $R(t)$ through the equations in §3.2.6. The time spent by a Jiggling system above 0.99 reliability is also listed in terms of the mean time between fault arrivals. For cm42a, cm138a, rd53, rd73 repairs were completed nearly until all spares were exhausted. Roughly 87% of spare resources were used up before the system failed. This resulted in a time above 0.99 reliability of on average 95 times the mean time between fault arrivals (Figures 3.5,3.6,3.7,3.8,3.9). This 48 times longer than a TMR/Simplex system using modules with no spares and is 15 times longer than a Jiggling strategy using the same total area with $N = 3 + \lfloor \frac{3 \lambda A}{2 \lambda A} \rfloor$ so the area allocated to spares in the Jiggling system is allocated to other modules instead. The NMR alternatives stay above 0.99 reliability on average for 7 times the mean time between fault arrivals. §3.5 demonstrates that the 4.32M timeout significantly skews results negatively and that in fact they are closer to the theoretical maximum of repairing whenever there is a spare available which is shown as the dotted curves on the graphs. This would result in an average time above 0.99 reliability of 116 times the mean time between
Figure 3.6: Reliability vs. time for cm138a. TMR/Simplex (dashed), 7MR (dot-dashed) Jiggling (solid) and Jiggling with every repair successful (dotted).

Figure 3.7: Reliability vs. time for rd53. TMR/Simplex (dashed), 8MR (dot-dashed), Jiggling (solid) and Jiggling with every repair successful (dotted).

Figure 3.8: Reliability vs. time for m1. TMR/Simplex (dashed), 5MR (dot-dashed), Jiggling (solid) and Jiggling with every repair successful (dotted).
faults. This means that if the permanent local damage interarrival time for FPGAs in outer space were 2 years, the Jiggling system would be 0.99 probable to survive a 232 year mission.

Repair was not successful for decod and p82. Possible reasons for why a fitness function may not be able to guide the evolutionary repair process successfully and how to predict such difficulty are presented in §3.4. Repair count for m1 was either very low failing in the first repair or very large repairing 30 accumulated faults successfully. This is probably due to the circuit’s large size and deep structure. When a fault hits a LUT close to an output, successive mutations to the logic structure hidden by this fault will not affect circuit behaviour and will destroy a large part of the functional circuit. This damage will require many generations to repair. For several fault sequences a large number of faults were successfully repaired. These were likely faults deeper down the structure and their repairability shows that the fitness function was successfully guiding the repair process towards functional solutions. This suggests that given more than 4.32M generations (or 3.03s), the repair process would have completed repairs even when fault hit LUTs closer to the outputs.

3.3.1 Discussion
The reliability of the Jiggling system studied in this chapter has been skewed negatively by the following factors. A fault in one of the modules not being repaired does not necessarily lead to system failure. Firstly, and specially relevant when spare count exceeds mission logic count, faults may arrive at spares of these modules. The probability of a fault hitting a spare at a module after \(i\) repairs have been completed at this module can be calculated from the amount of \(r_{ij} = 0\) for each \(i\). This eventuality could be considered in the calculation of \(P_i\). Secondly, two modules each carrying a fault may produce erroneous outputs for disjoint sets of inputs (ie. they never produce a false output at the same time). In this case there is enough information at the voter output to repair both modules, returning to the healthy state. The 4.32M generation limit equates to under 5 seconds repair time at 1GHz and thus ignores a huge amount of possibilities for repair. This could be made more realistic by allowing the simulator to run for longer. When allowing such a larger limit it is expected (§3.5) that nearly all repairs would succeed making reliability more like the dotted lines in the reliability graphs above. Finally making gate delay variation more realistic may also facilitate repair.

Even with the 4.32M limit, repair was successful most of the times for cm42a, cm138a, rd53 and rd73 when there were spares available. §3.5 suggests that by making this limit slightly more realistic such as 43.2M (under 50 seconds at 1GHz), repair would succeed at nearly every oppor-
tunity and the system would repair up to the number of spares available. Thus each spare added to a module would lengthen its life by roughly the fault interarrival time. Adding spares gives diminishing returns because of the increased area, but it remains possible to compute the number of spares required to provide a desired reliability at a specific point in the mission.

This work assumes the circuit being repaired is being used for normal operation and its input is effectively randomised at every clock cycle. If the circuit were not in use then it could supplied inputs artificially, possibly from the RNG or the counter. This would also be necessary if the full set of input vectors is not frequently encountered during normal operation. If the normal mission input pattern is not uniformly random and its characteristics are known, these could be used during simulated repair to generate the appropriate reliability figures.

This preliminary case study evaluated the Jiggling system repairing a design smaller than itself and increasing its reliability. The high probability of repair before spares are exhausted suggests the method could be applied to larger benchmarks if the timeout is increased. Each of the $W$ fault sequences tested for repairing the benchmarks took about 2 days on a 2GHz processor. The amount of processing power required will vary exponentially with benchmark size due to increased simulation time per configuration and longer evolutionary search.

As Jiggling is evaluated for larger benchmarks and as the fault model used is made more realistic, the possibility of using real hardware is more attractive. The Jiggling system could be implemented on a Xilinx FPGA subjected to radiation. This would give us accurate overhead measures as well as more accurate reliability figures since all cases mentioned above would be taken into account in the real system and generations could be truly performed at millions a second.

### 3.4 Predicting fitness function effectiveness

This section aims to answer the following questions:

1. Why are some fitness functions more effective at completing repairs than others?
2. When is this the case?
3. What can be learnt and how can Jiggling be improved using this knowledge?

The fitness functions to be compared are listed below. $I(Q)$ is the number of circuit inputs (outputs). $y_q(x)$ ($y'_q(x)$) is actual (desired) circuit response at output $q$ to input $x$. Let $m_q = \sum_{t=0}^{2^t-1} y_q(x) y'_q(x)$, $a_q = \sum_{t=0}^{2^t-1} y_q(x)$ and $d_q = \sum_{t=0}^{2^t-1} y'_q(x)$.

**S) Sum** This counts the number of output response bits which are correct and has the smallest hardware implementation requiring only a counter.

$$f = \sum_{i=0}^{2^t-1} \sum_{q=0}^{Q-1} D_{y_q(x), y'_q(x)}$$

where $D_{i,j} = 1$ if $i = j$ and 0 otherwise.

**C) Correlation** The statistical correlation of the output response data series $y_q$ with the desired response data series $y'_q$, summed over all outputs.

$$f = \sum_{q=0}^{Q-1} \left( \frac{m_q - \frac{1}{Q} a_q d_q}{\sqrt{V(y_q)V(y'_q)}} \right)$$

where $V(X)$ is the variance of the data series $X$. This has consistently been found to yield best results but is expensive to implement in hardware. $f$ is limited to positive values by setting it to zero if the RHS expression is negative.
SC) **Simple correlation** A version of C with simpler scaling used for the experiments above.

\[ f = \sum_{q}^{Q-1} \left( \frac{2^q a_q - a_q d_q}{2^{q} d_q - d_2^2} \right) \]  

(3.10)

\( f \) is again limited to positive values.

CC) **Cheap correlation** An approximation of SC with a smaller hardware implementation.

Let the constant \( \sum x'_{q}(x) = 2^{m_0} + 2^{m_1} + \ldots + 2^{m_k} \) where \( m_0 > m_1 > \ldots > m_k \).

\[ f = \sum_{q}^{Q-1} \left( 2^{l-m_0} a_q - a_q \right) \]

### 3.4.1 Random Data Tests

Several experiments were performed on random data as follows. Three random data series \( s_i \) of length \( L \) and \( Q \) outputs were generated. \( s_1 \) was used as \( y' \) and then the fitness of \( s_0 \) and \( s_2 \) were evaluated using different fitness functions. This was repeated 10000 times. For long data series with \( L = 32 \times 30 \) and low output count with \( Q = 1 \), C and S chose the same one of \( s_0 \) and \( s_2 \) as the fittest 88.7% of the time, C and SC agreed 99.9% of the time and S and SC agreed in 87.6% decisions. This shows that \( S \) and \( C \) are very different, choosing a different series as the fittest in 11% of the cases whereas in 0.1% of cases \( C \) and SC choose a different random data series as fitter. For \( L = 32 \) and \( Q = 30 \) the agreed decisions were 81.9% for \( C \) and \( S \), 98.9% for \( C \) and SC and 82.0% for \( S \) and SC. These values were roughly linearly interpolated for intermediate values of \( L \) and \( Q \). The overall reduced agreement is because shorter data series are more likely to be statistically different so more diverse fitness values will be generated amplifying the difference between fitness functions. *For totally random series SC seems to always make a closer decision to C than S does.*

Evolution is usually faced with output responses which are very similar, specially when the population is converged as in a (1+1) ES. Another set of tests was conducted as above with \( s_1 \) generated randomly and \( s_0, s_2 \) being copies of \( s_1 \) with 3 bits flipped at random per output. With \( Q = 1 \) and \( L = 32 \) \( S \) and \( C \) agreed 74.1%, \( C \) and SC 91.0% and \( S \) and SC 76.4%. Agreement is lower than above because fitness functions are less likely to agree when judging subtle differences. *If C is the best fitness function available to guide evolution then SC is not as good but is better than S.* With \( L = 32 \) and \( Q = 30 \) \( S \) and \( C \) agreed 90.5%, \( C \) and SC 88.6% and S and SC 83.6%. Agreement data was roughly linearly interpolated for intermediate values of \( Q \). The fact that \( S \) agrees with \( C \) more than \( SC \) does could be explained by the scaling term missing in the SC formula. Fitness changes at one output are larger than at another depending on the number of high bits in the output series. This could cause corruption of one output series to dominate the overall fitness no matter how much other output series are corrupted. This means that \( SC \) may not be suitable for evolution of circuits with many outputs.

### 3.4.2 Structural Tests

These were performed to collect more realistic data on fitness function effectiveness and to validate the findings described above. The genotype for a circuit synthesised by Sis (E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and Sangiovanni-Vincentelli, 1992) for a given benchmark was mutated at two rates. 10000 mutants were generated with a low mutation rate and 10000 with a high mutation rate. Then each fitness function was used to judge which mutant was fittest. This is a practical test of fitness function effectiveness because their purpose is in fact to judge which individuals are less mutations away from a solution. A Sis synthesised circuit was used as opposed to evolved solutions in order to avoid bias to areas of the fitness landscape which could have been more suitable for the fitness function used when evolving the solution. Moreover, all first fault Jiggling repairs start from such
a synthesised configuration. A fuller structural analysis of fitness function effectiveness could
include mutated evolved configurations with several faults present – where evolution spends most
of its time – extracted from real runs using different fitness functions. Results may or may not
be different under such extended tests depending on if circuit structure or function is the factor
dominating fitness function effectiveness.

Several benchmark circuits from the MCNC ’91 suite (Table 3.1) were used for the tests. Low (high) rate mutants were generated with 1 (10) mutations per genotype making sure they all affected circuit structure and behaviour. For each benchmark the following statistics were collected:

1. Number of times a fitness function decided the low rate mutant was fitter. This defines the
probability \( p_a \) of consistently making the correct choice.

2. Number of times a fitness function decided the high rate mutant was fitter. Defines the
probability \( p_b \) of consistently making the wrong choice.

3. The probability \( p_c \) of making the correct choice assuming that when both individuals are
considered equally fit, one will be picked at random. \( p_c = p_a + \frac{1-p_a-p_b}{2} \). This is actually
the case during Jiggling: when faced with two individuals of equal fitness the (1+1) ES will
choose the mutant. This situation will happen equiprobably with the equivalents of the high
and low rate mutants being elite and mutant.

This data was calculated for the four fitness functions defined above and for two combinations
of these:

**SSC** Combines S and SC so that an individual is considered fitter than another only if it is
fitter under some fitness function and not worse under the other.

**SCC** As SSC but with S and CC.

These combined fitness functions arose from the observation that in many cases SC makes the
correct decision when S does not or considers them equal, while in many cases where SC makes
the wrong decision S makes the correct one. With SSC the algorithm will never consistently
choose an individual over another if it is worse at either S or SC. In a way the combined fitness
functions generate an ‘anding’ of the fitness landscape so that a point is only high if it is high in
both fitness functions. The aim is to provide a richer judgement of fitness and get the best of both
worlds. It is likely the combined fitness function could serve to avoid pathological local maxima
defined by S and SC in the areas where wrong decisions are made. They could also introduce more
smoothness into the landscape as there may be cases in which changes in behaviour affect only
one of S and SC.

**Results**
The best (second best) values ignoring C of \( p_b \) and \( p_c \) will be in bold (underlined). These mark
the two best fitness functions with a reasonably simple hardware implementation at avoiding con-
sistent wrong decisions and at making the correct decision.

**Table 3.2.5: Results from structural tests.**

<table>
<thead>
<tr>
<th>cm42a</th>
<th>( p_a )</th>
<th>( p_b )</th>
<th>( p_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>0.985</td>
<td>0.0126</td>
<td>0.9862</td>
</tr>
<tr>
<td>SC</td>
<td>0.9953</td>
<td>0.0047</td>
<td><strong>0.9952</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.985</td>
<td>0.0126</td>
<td>0.9862</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9806</td>
<td><strong>0.0003</strong></td>
<td>0.9808</td>
</tr>
<tr>
<td>SCC</td>
<td>0.985</td>
<td>0.0126</td>
<td>0.9862</td>
</tr>
<tr>
<td>rd53</td>
<td>$p_a$</td>
<td>$p_b$</td>
<td>$p_c$</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>C</td>
<td>0.9952</td>
<td>0.0046</td>
<td>0.9953</td>
</tr>
<tr>
<td>S</td>
<td>0.9902</td>
<td>0.0082</td>
<td>0.9910</td>
</tr>
<tr>
<td>SC</td>
<td>0.9938</td>
<td>0.0060</td>
<td><strong>0.9939</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.9835</td>
<td>0.0165</td>
<td>0.9835</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9888</td>
<td><strong>0.0040</strong></td>
<td>0.9924</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9822</td>
<td>0.0060</td>
<td>0.9881</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>decod</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9812</td>
<td>0.0188</td>
<td>0.9812</td>
</tr>
<tr>
<td>S</td>
<td>0.9934</td>
<td>0.0051</td>
<td><strong>0.9942</strong></td>
</tr>
<tr>
<td>SC</td>
<td>0.9783</td>
<td>0.0215</td>
<td>0.9784</td>
</tr>
<tr>
<td>CC</td>
<td>0.9781</td>
<td>0.0218</td>
<td>0.9782</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9729</td>
<td><strong>0.0006</strong></td>
<td>0.9862</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9727</td>
<td><strong>0.0006</strong></td>
<td>0.9861</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cm138a</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9971</td>
<td>0.0028</td>
<td>0.9972</td>
</tr>
<tr>
<td>S</td>
<td>0.9530</td>
<td>0.0461</td>
<td>0.9535</td>
</tr>
<tr>
<td>SC</td>
<td>0.9813</td>
<td>0.0185</td>
<td><strong>0.9814</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.9530</td>
<td>0.0461</td>
<td>0.9535</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9348</td>
<td><strong>0.0004</strong></td>
<td>0.9672</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9530</td>
<td>0.0461</td>
<td>0.9535</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p82</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9957</td>
<td>0.0043</td>
<td>0.9957</td>
</tr>
<tr>
<td>S</td>
<td>0.9941</td>
<td>0.0054</td>
<td><strong>0.9944</strong></td>
</tr>
<tr>
<td>SC</td>
<td>0.9908</td>
<td>0.0092</td>
<td>0.9908</td>
</tr>
<tr>
<td>CC</td>
<td>0.9907</td>
<td>0.0093</td>
<td>0.9907</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9871</td>
<td><strong>0.0020</strong></td>
<td>0.9926</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9869</td>
<td><strong>0.0020</strong></td>
<td>0.9925</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>m1</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9965</td>
<td>0.0035</td>
<td>0.9965</td>
</tr>
<tr>
<td>S</td>
<td>0.9890</td>
<td>0.0108</td>
<td>0.9891</td>
</tr>
<tr>
<td>SC</td>
<td>0.9922</td>
<td>0.0078</td>
<td><strong>0.9922</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.9782</td>
<td>0.0218</td>
<td>0.9782</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9858</td>
<td>0.0045</td>
<td>0.9907</td>
</tr>
<tr>
<td>SCC</td>
<td>0.0971</td>
<td><strong>0.0036</strong></td>
<td>0.5468</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bw</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9988</td>
<td>0.0012</td>
<td>0.9988</td>
</tr>
<tr>
<td>S</td>
<td>0.9964</td>
<td>0.0029</td>
<td>0.9968</td>
</tr>
<tr>
<td>SC</td>
<td>0.9981</td>
<td>0.0019</td>
<td><strong>0.9981</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.9924</td>
<td>0.0076</td>
<td>0.9924</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9954</td>
<td><strong>0.0009</strong></td>
<td>0.9973</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9899</td>
<td>0.0010</td>
<td>0.9945</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rd73</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9890</td>
<td>0.0110</td>
<td>0.9890</td>
</tr>
<tr>
<td>S</td>
<td>0.9885</td>
<td>0.0110</td>
<td>0.9888</td>
</tr>
<tr>
<td>SC</td>
<td>0.9890</td>
<td>0.0106</td>
<td><strong>0.9892</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.9849</td>
<td>0.0150</td>
<td>0.9850</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9882</td>
<td>0.0102</td>
<td>0.9890</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9828</td>
<td><strong>0.0092</strong></td>
<td>0.9868</td>
</tr>
</tbody>
</table>
### Chapter 3. Jiggling: Long survival of FPGA systems through evolutionary self-repair

#### 3.4.3 Fitness function tests vs. real runs: do they match?

This section will investigate the capacity of the random data and structural tests as predictors of fitness function efficiency at Jiggling repair.

First note that random data tests do predict structural test results. The similarity between C and SC in the random data tests is reflected in SC performing best only when C did. Another random data test showed that SC differed from C when there were many outputs. This is again reflected in the structural tests with the case of p82.

<table>
<thead>
<tr>
<th>misex1</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9938</td>
<td>0.0062</td>
<td>0.9938</td>
</tr>
<tr>
<td>S</td>
<td>0.9901</td>
<td>0.0099</td>
<td>0.9901</td>
</tr>
<tr>
<td>SC</td>
<td>0.9936</td>
<td>0.0064</td>
<td><strong>0.9936</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.9850</td>
<td>0.0150</td>
<td>0.9850</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9894</td>
<td><strong>0.0057</strong></td>
<td>0.9919</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9808</td>
<td><strong>0.0057</strong></td>
<td>0.9876</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5xp1</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9963</td>
<td>0.0037</td>
<td>0.9963</td>
</tr>
<tr>
<td>S</td>
<td>0.9973</td>
<td>0.0024</td>
<td><strong>0.9975</strong></td>
</tr>
<tr>
<td>SC</td>
<td>0.9965</td>
<td>0.0035</td>
<td>0.9965</td>
</tr>
<tr>
<td>CC</td>
<td>0.9957</td>
<td>0.0043</td>
<td>0.9957</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9961</td>
<td>0.0022</td>
<td>0.9970</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9949</td>
<td><strong>0.0018</strong></td>
<td>0.9966</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z5xp1</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9947</td>
<td>0.0026</td>
<td>0.9961</td>
</tr>
<tr>
<td>S</td>
<td>0.9975</td>
<td>0.0020</td>
<td><strong>0.9978</strong></td>
</tr>
<tr>
<td>SC</td>
<td>0.9976</td>
<td>0.0024</td>
<td>0.9976</td>
</tr>
<tr>
<td>CC</td>
<td>0.9965</td>
<td>0.0034</td>
<td>0.9966</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9970</td>
<td>0.0018</td>
<td>0.9976</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9958</td>
<td><strong>0.0015</strong></td>
<td>0.9972</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inc</th>
<th>$p_a$</th>
<th>$p_b$</th>
<th>$p_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.9983</td>
<td>0.0017</td>
<td>0.9983</td>
</tr>
<tr>
<td>S</td>
<td>0.9951</td>
<td>0.0049</td>
<td>0.9951</td>
</tr>
<tr>
<td>SC</td>
<td>0.9975</td>
<td>0.0025</td>
<td><strong>0.9975</strong></td>
</tr>
<tr>
<td>CC</td>
<td>0.9922</td>
<td>0.0078</td>
<td>0.9922</td>
</tr>
<tr>
<td>SSC</td>
<td>0.9944</td>
<td>0.0018</td>
<td>0.9963</td>
</tr>
<tr>
<td>SCC</td>
<td>0.9873</td>
<td><strong>0.0000</strong></td>
<td>0.9937</td>
</tr>
</tbody>
</table>

The $p_c$ for C is highest for all benchmarks except decode, Z5xp1 and 5xp1. Hence *C is in fact the best fitness function at choosing the mutant closest to the solution.* In all the cases where *C is the best, SC is the best of the rest except in p82 where the number of outputs is large* confirming the predictions of the random data tests. Given that the hardware implementation for C is prohibitively large further discussion will ignore it. When SC is not the best at choosing the closer mutant, S is. *CC is the worst in all cases except cm42a.* As expected, *the combined fitness functions are the two best at avoiding consistent wrong decisions* in all save two benchmarks. *SSC is always best or second best at both $p_c$ and $p_b$.*

The structural test results open up two new questions:

4. Do the structural test results predict anything about repair times?

5. During evolutionary repair, is it more important to choose the correct individual more often (high $p_c$) or to gamble when uncertain and avoid consistent wrong decisions (low $p_b$)?

---

The $p_c$ for C is highest for all benchmarks except decode, Z5xp1 and 5xp1. Hence *C is in fact the best fitness function at choosing the mutant closest to the solution.* In all the cases where *C is the best, SC is the best of the rest except in p82 where the number of outputs is large* confirming the predictions of the random data tests. Given that the hardware implementation for C is prohibitively large further discussion will ignore it. When SC is not the best at choosing the closer mutant, S is. *CC is the worst in all cases except cm42a.* As expected, *the combined fitness functions are the two best at avoiding consistent wrong decisions* in all save two benchmarks. *SSC is always best or second best at both $p_c$ and $p_b$.*

The structural test results open up two new questions:

4. Do the structural test results predict anything about repair times?

5. During evolutionary repair, is it more important to choose the correct individual more often (high $p_c$) or to gamble when uncertain and avoid consistent wrong decisions (low $p_b$)?

---

The $p_c$ for C is highest for all benchmarks except decode, Z5xp1 and 5xp1. Hence *C is in fact the best fitness function at choosing the mutant closest to the solution.* In all the cases where *C is the best, SC is the best of the rest except in p82 where the number of outputs is large* confirming the predictions of the random data tests. Given that the hardware implementation for C is prohibitively large further discussion will ignore it. When SC is not the best at choosing the closer mutant, S is. *CC is the worst in all cases except cm42a.* As expected, *the combined fitness functions are the two best at avoiding consistent wrong decisions* in all save two benchmarks. *SSC is always best or second best at both $p_c$ and $p_b$.*

The structural test results open up two new questions:

4. Do the structural test results predict anything about repair times?

5. During evolutionary repair, is it more important to choose the correct individual more often (high $p_c$) or to gamble when uncertain and avoid consistent wrong decisions (low $p_b$)?

---

The $p_c$ for C is highest for all benchmarks except decode, Z5xp1 and 5xp1. Hence *C is in fact the best fitness function at choosing the mutant closest to the solution.* In all the cases where *C is the best, SC is the best of the rest except in p82 where the number of outputs is large* confirming the predictions of the random data tests. Given that the hardware implementation for C is prohibitively large further discussion will ignore it. When SC is not the best at choosing the closer mutant, S is. *CC is the worst in all cases except cm42a.* As expected, *the combined fitness functions are the two best at avoiding consistent wrong decisions* in all save two benchmarks. *SSC is always best or second best at both $p_c$ and $p_b$.*
Table 3.3: Repair count statistics for some benchmarks and fitness functions. SC, S and SSC list the average number of repairs for each fitness function. \( p(S=SC) \) and \( p(SSC=SC) \) list the probability that their means are actually the same – i.e. that the null hypothesis is true. A strong relationship with structural test results is evident.

<table>
<thead>
<tr>
<th>Bench.</th>
<th>SC</th>
<th>S</th>
<th>( p(S=SC) )</th>
<th>SSC</th>
<th>( p(SSC=SC) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm42a</td>
<td>13.14</td>
<td>7.72</td>
<td>0</td>
<td>14.56</td>
<td>0.07</td>
</tr>
<tr>
<td>cm138a</td>
<td>13.98</td>
<td>2.33</td>
<td>0</td>
<td>14.8</td>
<td>0.33</td>
</tr>
<tr>
<td>decod</td>
<td>0.29</td>
<td>0.33</td>
<td>0.87</td>
<td>0.55</td>
<td>0.46</td>
</tr>
<tr>
<td>rd53</td>
<td>34.88</td>
<td>33.82</td>
<td>0.83</td>
<td>36.08</td>
<td>0.68</td>
</tr>
<tr>
<td>p82</td>
<td>0.63</td>
<td>1</td>
<td>0.43</td>
<td>0.8</td>
<td>0.88</td>
</tr>
<tr>
<td>m1</td>
<td>7.29</td>
<td>1.1</td>
<td>0.06</td>
<td>13.6</td>
<td>0.26</td>
</tr>
<tr>
<td>rd73</td>
<td>17.17</td>
<td>15.5</td>
<td>0.92</td>
<td>18.67</td>
<td>0.65</td>
</tr>
</tbody>
</table>

To measure the capacity of structural test results as predictors of fitness function repair efficiency, repair times statistics were collected for several benchmarks under different fitness functions. CC and SCC did not perform well during any of the tests. This is likely due to their poor \( p_c \) value. The results in Table 3.3 show strong correlation between structural test results and repair count. Whenever SC had the best \( p_c \) in the structural tests it performs well during real repairs and SSC performs even better. When SC had a worse \( p_c \) than S (decod and p82), no fitness function is suitable for repair. The improved performance of SSC over SC could help answer question 5 above. It seems like avoiding consistent wrong choices is better than choosing consistently correct ones, given that the overall chances of making a correct choice are high. It should be noted that S performed better than SC for no benchmark. The structural tests predict when repair using SC or SSC is feasible. Informal experiments using a larger timeout for m1 suggested the timeout is responsible for the low repair count.

### 3.4.4 Conclusions

The above results will be used to attempt answer the questions posed earlier.

**Why are some fitness functions more effective at completing repairs than others?**

Structural tests showed how certain fitness functions are better at differentiating between a low rate and a high rate mutant of a solution. The capacity to do so was shown to be a good predictor (and cause) for effectiveness at repair.

It has been seen how in most cases the C and SC fitness functions are more effective than S. One reason for this may be that *statistical correlation is a better indication of output response “togetherness” than hamming distance*. Eg. if an output should be high a third of the time, C and SC will give 0 fitness to an output tied low while S will give it a misleadingly high \((0.67 \times 2^I)\) score. Another reason could be that *C and SC provide a smoother landscape than S*. This is because there are only \(2^I\) possible fitness values for S per output whereas there are at least \(2^{2I}\) values for SC because for each of the \(2^I\) possible values of desired output high bit count there is that same amount of values for mutual high bit count and actual high bit count. Smoother landscapes clearly provide a better guide to evolution than landscapes with large steps. Both factors probably contribute since the data above shows that S makes the wrong decision more often (has a higher \(p_b\) because it judges using the hamming distance instead of statistical correlation and at the same time it also judges low and high rate mutants as having equal fitness significantly more often \((1 - p_a - p_b\) is higher) than C and SC do.

The fact that C is better than SC could be attributed to the missing scaling term. This reduces the smoothness of the landscapes and also distorts the importance of fitness changes at different outputs as was shown in §3.4.1.
When is this the case?
It seems like SSC is always better than SC which is always better than S, except for benchmarks in which they are all equally bad. When the latter will be the case can be predicted fairly accurately using the structural tests §3.4.3.

What can be learnt and how can Jiggling be improved using this knowledge?
Combining two fitness functions as with SSC can improve evolvability and hence repairability. Upgrading an SC Jiggler to SSC would only require adding a couple of counters and a LUT. Some benchmarks may not be suitable for repair and this can be predicted using the structural tests. In the case that this is due to the output function then a new fitness function may be the only way of making them repairable through Jiggling. In the case that they are not repairable due to their structure then alternate versions could be synthesised with different optimisation algorithms and each alternative could be evaluated using the structural tests to find one suitable to repair with Jiggling.

3.5 Jiggler Variations

This section aims to explore the effectiveness of several alterations to the Jiggling architecture and its parameters. This should help understand its operation and how to improve it. It will also evaluate Jiggling using a longer repair timeout to estimate how pessimistic results are due to the timeout.

All experiments are implemented as changes in the simulation. The benchmark used to test these variations is the cm42a circuit and repair time results are collected as in §3.2.5. Simulation configuration is as described above §3.2.5 and is varied for each experiment as follows:

Std) Standard Variations are compared against the standard Jiggling architecture §3.2.3. A (1+1) ES is used with the SC fitness function. Mutation rate is 1 per genotype and history window size $H = 32$. Output is extracted as the worst value from the last 5 steps of a simulated clock cycle of 30 steps.

M2) Mutation rate Mutation rate has been suggested Schafer et al. (1989) to be one of the main variables affecting GA performance. An accepted Hesser and Mannor (1992) efficient value for mutation rate per genotype is one, but the optimal value is believed to depend on the selection pressure and the amount of junk in the typical individual. One rule of thumb used by Inman Harvey is to set the mutation rate so that one mutation is expected in the functional part of the genotype, i.e. if most genotypes are half junk, set it to two mutations per genotype. Another theory Barnett (2001) suggests that an optimal rate is found when the probability of a neutral (with same fitness) mutant is $1/e$. However it is still impossible to a priori calculate an optimal mutation rate for a given optimisation problem due to the varying particularities of the structure of the fitness landscape. This variation consists of doubling mutation rate to 2 per genotype.

H8) History window size As described in §3.2.3 the probability of reverting to an unstable circuit varies as $p^H$ where $p$ is the probability of a lucky high evaluation of an unstable circuit and $H$ is the window size. This variation consists of setting $H = 8$. $H$ does not affect Jiggler size significantly, but could affect performance.

POP) Full Population This variation uses a full population GA repair process with 30 individuals similar to Lohn et al. (2003a); Vigander (2001). $X$ of the next generation are created through single point crossover, $Y$ by one mutation per genotype and there is one elite.

W15) Sample Window This variation samples output as the worst value from the last 15 steps of each simulated clock cycle.

MS) Multi-sampling This architecture variation requires $2^l(1 + 2Q)$ bits to be allocated for recording output response of the faulty module. $2^l$ bits store if each input has been seen or not and $2^l2Q$ store the actual output response: for each output the first (second) bit stores if there was a 1 at the output of all modules (the faulty module). Given that evaluation finishes when all inputs have been seen and that a fresh random input is available at each clock cycle, then some inputs...
Table 3.4: Repair statistics for each variation in terms of number of successfully repaired accumulated faults. The last column is the probability that the variation has no impact on reliability, i.e. that the null hypothesis is true.

<table>
<thead>
<tr>
<th>Variation</th>
<th>Avg. repairs</th>
<th>Sequences</th>
<th>Std. err.</th>
<th>p(same as std.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Std</td>
<td>13.14</td>
<td>37</td>
<td>0.64</td>
<td>1.0</td>
</tr>
<tr>
<td>M2</td>
<td>13.4</td>
<td>15</td>
<td>0.97</td>
<td>0.82</td>
</tr>
<tr>
<td>H8</td>
<td>0.3</td>
<td>10</td>
<td>0.21</td>
<td>0.00</td>
</tr>
<tr>
<td>POP</td>
<td>18.33</td>
<td>6</td>
<td>1.09</td>
<td>0.00</td>
</tr>
<tr>
<td>W15</td>
<td>11.04</td>
<td>27</td>
<td>0.69</td>
<td>0.03</td>
</tr>
<tr>
<td>MS</td>
<td>14.62</td>
<td>34</td>
<td>0.48</td>
<td>0.07</td>
</tr>
<tr>
<td>T</td>
<td>18</td>
<td>12</td>
<td>0.86</td>
<td>0.00</td>
</tr>
</tbody>
</table>

may be presented multiple times during evaluation. Storing output response to each input allows the stored response to be updated during a later presentation of the same input. This variation penalises unstable circuits: if output is sampled differently during two presentations of the same input during the same evaluation, then the wrong output will be stored.

**T) Longer timeout** Reliability data presented in §3.3 is pessimistic because only 4.32M generations, or 0.64 seconds were allowed for each repair process whereas a real Jiggling system in deep space might have months to complete a repair. This variation increases the timeout to 43.2M, or 6.4 seconds.

### 3.5.1 Results

The reliability of a Jiggling system depends only on the number of simulated successful repairs because the timeout forces repair times to be either several orders of magnitude smaller than the fault interarrival time or infinite (timed out). Several random fault sequences are generated §3.2.5. Faults are injected in turn and a repair process is initiated for each. Jiggler variations will be compared in terms of the number of faults successfully repaired before a repair timed out.

Repair statistics for each variation are presented in Table 3.4. The second and third columns list the average number of repaired faults during each sequence and the number of fault sequences simulated. The cm42a benchmark uses 10 LUTs and 18 are allocated as spares. Repairing 22 faults in a sequence would most probably mean that all spares were allocated and at least 4 faults were at the same LUT as a previous fault in the sequence. The third column lists the standard error for the series of repair count data. The last column evaluates the significance of the difference between this series’ mean and that of the standard architecture using a two tailed Student’s t-test. The probability listed is that of both series having the same mean – i.e. that the null hypothesis is true and the variation does not impact reliability. Even though conclusions drawn from this data apply strictly only to cm42a there is no reason to believe they do not apply to other benchmarks as well. From this data the following observations can be made:

**M2** Increasing mutation rate to 2 does not affect reliability. Other mutation rates could be tested.

**H8** A history window of size 8 proved virtually useless showing the importance of this method for avoiding unstable configurations. To illustrate the difference this variation makes, let the probability of a lucky high evaluation be \( p = 0.66 \). Then the probability of not detecting it as unstable and allowing it to become the elite with \( H = 8 \) is 0.036 and with \( H = 32 \) is 0.0000017.

**POP** The full population repair doing 30 evaluations per generation instead of 2 is clearly superior doing 18.33 repairs on average compared to 13.14. However, since the timeout was 4.32M generations for both, the population variation was allowed 15 times the evaluations as the standard (1+1). This result is mainly for academic comparison since using a full population repair would significantly increase the repair mechanism overhead.
W15) Using the last 15 steps for output sampling significantly hindered repairability showing that care must be taken into how outputs are latched/sampled during repair on a real Jiggler.

MS) Simulations using this variation repaired one extra fault on average with the probability of rejecting the null hypothesis erroneously being 0.07. This means that sampling output more than once to discourage unstable circuits has a small positive effect on repairability. This is likely to transfer to other benchmarks however whether the extra overhead is worthwhile will depend on the particular mission requirements.

T) The imposed timeout of 4.32M significantly skews results negatively. Almost all repairs with spares available were completed with the new timeout of 43.2M and reliability (Fig3.10) is significantly increased so that the system has a 0.99 probability of survival after 75 times the mean time to permanent fault arrival. This confirms that results in §3.3 are pessimistic and that the actual Jiggling reliability solid curves lie closer to the ‘all repairs’ dotted curves.

3.6 Trading overhead for time

The main strategy underlying the Jiggling method is to trade overhead – by using a minimal “blind” repair mechanism – for time – by allowing repair to be non-immediate. Reducing overhead in the repair mechanism reduces the size of the single point of failure and makes repairing the repairer easier §3.2.3 with the possibility of having no single point of failure at all. Repair time can be extended because the Jiggling system is online during repair of a fault and the permanent local fault interarrival time for FPGAs in outer space is likely to be of several months.

This section aims to explore how far this strategy can be taken and how different trade-offs between overhead and time can be achieved by varying the Jiggling architecture. It will also discuss the possibility of non-concurrent repair.

3.6.1 Architecture variations implementing different trade-offs

The overhead and time traded will be that required for collecting fitness data during circuit evaluation. Each variation will now be introduced followed by analysis of how overhead and time are affected. As a reminder, the availability of a fresh random input is assumed at each clock cycle during normal operation, and thus on-line concurrent repair. $I(Q)$ is the number of inputs (outputs) of the circuit under repair.

A) All Once This is the standard architecture described above in §3.2.2 where each of the $2^I$ inputs need only be seen once and in any order. One bit per input is required to signal if it has been

Figure 3.10: Reliability vs. time for TMR/Simplex (dashed), 8MR (dot-dashed), Jiggling with 43.2M timeout (solid) and Jiggling with every repair successful (dotted).
seen plus two counters to collect the output response data necessary for simple correlation fitness calculation.

B) In Order A counter is used to signal which input is currently being sought. When the input is seen, the counter is increased. Another two counters record output response data. Evaluation ends when the first counter has traversed all inputs.

C) Count & Table This is a combination of the previous two methods. A counter is used for the first $m$ input bits and a table for the remaining $I - m$ bits. The $m$ counter bits specify the first $m$ bits of the input currently being sought. There are $2^{I - m}$ of these inputs and the table stores whether they have been seen. Two counters are used to collect output response data. When all inputs in the current partition have been seen the the $m$ bit counter is increased. Evaluation ends when this counter has traversed all $2^m$ values.

D) Keen Counters $k$ counters are used, each covering a different partition of the input space. Two more counters record output response data. Counters in adjacent input space partitions go in opposite directions. This has the consequence that counters facing each other can stop whenever they meet instead of at predefined boundaries. This means that fast (lucky) ones can make up for the lost time of slow ones. When two counters meet they are allocated an unfinished sub-partition. This ensures $k$ inputs are been sought at all times.

3.6.2 Analysis

Table 3.5 shows the storage space and time required for each circuit evaluation under each architecture variation. Proofs of the formulae can be found in Appendix §A.1. As expected, A is fastest and largest, B is slowest and smallest, and the others are configurable trade-offs in between. What is remarkable is that storage space and time are linearly interchangeable, ie. a reduction in space by a factor $x$ produces a lengthening in evaluation time of $x$. Also, it is possible to choose the degree of trade off desired by varying $m$ in C and $k$ in D. This means that the Jiggling system is flexible to adapt to different fault rates, overhead constraints and circuit properties relevant to different missions.

The results in Table 3.5 hint at the presence of a lower bound to the amount both overhead and time can be reduced. In fact, $O(\text{space } \times \text{ time}) = O(2^I)$ for all variations (Fig.3.11). It is not at all obvious why evaluation time multiplied by the storage used for evaluation under random inputs should be constant no matter what scheme is used. It is in fact possible to devise suboptimal strategies for which this is not the case, but these clearly have no place in this discussion. The question hangs, is there a mathematical minimum as to how much information can be collected about a system using certain amount of observations and storage? For example, variation B uses an $I$ bit counter to record which inputs have been seen and which haven’t but requires waiting on average $2^I$ inputs to find the next one. Why should this correspond so well with the $2^I$ storage space and average $I\ln 2$ inputs in variation A? The answer to this question may shed light on the possibility or impossibility of strategies using both lower overhead and evaluation time.

Sub-optimal architectures and large $Q$

In cases where $Q \gg I$ it may be useful to leave $Q$ in the space growth upper bound expression above. Eg. for variation A this would be $O(s) = O(2^I + I\log_2{Q})$. If the architectural variation MS described in §3.5 using $1 + 2Q$ bits per input – 1 bit storing if it was seen and $2Q$ for output

<table>
<thead>
<tr>
<th>Var.</th>
<th>Storage space $s$</th>
<th>$O(s)$</th>
<th>Evaluation time $t$</th>
<th>$O(t)$</th>
<th>$O(s \times t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$2^I + 2\log_2{2^I}$</td>
<td>$O(2^I)$</td>
<td>$&lt; 2^I(1 + I\ln 2)$</td>
<td>$O(2^I)$</td>
<td>$O(2^I)$</td>
</tr>
<tr>
<td>B</td>
<td>$I + 2\log_2{2^I}$</td>
<td>$O(I)$</td>
<td>$2^I$</td>
<td>$O(2^I)$</td>
<td>$O(2^I)$</td>
</tr>
<tr>
<td>C</td>
<td>$m + 2^{I - m} + 2\log_2{2^I}$</td>
<td>$O(2^{I - m} + m)$</td>
<td>$&lt; 2^{I - m}(1 + (I - m)\ln 2)$</td>
<td>$O(2^{I - m}(1 + I - m))$</td>
<td>$O(2^{I - m}(1 + I - m))$</td>
</tr>
<tr>
<td>D</td>
<td>$k(\log_2{2^I} - m) + 2\log_2{2^I}$</td>
<td>$O(kI)$</td>
<td>$O(kI)$</td>
<td>$O(kI)$</td>
<td>$O(kI)$</td>
</tr>
</tbody>
</table>
Chapter 3. Jiggling: Long survival of FPGA systems through evolutionary self-repair

Figure 3.11: $O(\text{storage} \times \text{evaluation time})$ is constant and equal to $I2^I$. Repair time is proportional to evaluation time. Storage overhead is that used when recording output response.

response – were used then $s = 2^I(1 + 2Q)$ and $O(s) = O(2^I Q)$. The benefit of the latter architecture is that it can adjust the stored response if an input is seen twice. Now $O(s \times t)$ for the first is $O(2^I I + 2^I I^2 \log_2 Q)$ and for the second $O(2^I IQ)$. The second clearly grows faster and in the light of observation from Table 3.5 this would suggest that it is a sub-optimal use of space and time to gather full information about a circuit – although it could be useful when response is inconsistent – and may not be practical when $Q$ is large.

3.6.3 A minimal Jiggler

Using architecture B, a Jiggler capable of protecting all circuits in Table 3.1 would only require 97 LUTs. Impact on evaluation time would be minimal for a benchmark like cm42a raising it from an average 73.6 to 276 clock cycles resulting in a reliability curve almost identical to that in fig. 3.5. Even for a benchmark like misex1 with 8 inputs evaluation time is raised to an average 65556 clock cycles. This would mean that the 4.32M timeout would be equivalent to 9.4 minutes at 1Ghz and all repairs completed in simulation would still be completed well before the next fault arrived. However for circuits with more than 16 inputs a single evaluation would require 8.6s and a repair of 4.32M would take just over a year thus dramatically increasing the probability of a second fault before repair is complete.

3.6.4 Non-concurrent repair

Discussion so far has dealt with the Jiggling system performing repair of the faulty module concurrently in the background as the two remaining healthy modules drove output during normal circuit operation. Now discussion turns to the possibility of interrupting system operation temporarily off-line for repair purposes.

In some cases, the system under repair may not be in use or may be in use at a low frequency. By latching voter (system) outputs it would be possible to perform off-line repair between the low frequency use. In this case both overhead and repair time can be minimised since a counter could be used to register how many vectors have been seen and a test pattern generator (TPG) could feed in each input once. The TPG could be simply another counter or a linear feedback shift register. Fitness would be calculated by another counter adding how many correct outputs have been seen so far. This scheme would require $O(I)$ storage and $O(2^I)$ evaluation time, both the lowest of any variation described above. The advantage over the possible lower bound mentioned earlier §3.6.1 is due to inputs not being random any more. Off-line repair would be both faster and use less overhead.

Non-concurrent repair would not be practical if the system under repair were mission critical.
and were being used at speed. However if the Jiggling (FPGA) clock were considerably faster than the external clock then non-concurrent repair could be transparent to the external system.

3.7 Population Fault Tolerance and Repairability

This section will present some experiments performed to research if the Population Fault Tolerance (PFT) concept developed by Layzell (2001) could be useful for improving repairability in Jiggling systems. Layzell speculated that point mutations on evolved individuals could restore functionality by activating ancestral genetic material. The (1+1) ES used during Jiggling works with such point mutations. Hypothesis H3 in Layzell (2001) stated that PFT was not a property of evolved circuits but a property of all circuits. It was assumed false but was not proved nor disproved. This work also sets out to evaluate H3.

3.7.1 Introduction

PFT was introduced in Layzell (2001) as the inherent fault tolerance of a population of evolved circuits. It was found that when a fault was inserted at a particular element in all individuals in a population, some individuals had a higher fitness in the presence of this fault than the fault-free best. It was also found that for circuits evolved using a (1+1) ES, many mutants of the elite were fitter when subjected to the fault that the elite itself. This effect was assumed due to the nature of evolved designs still containing primitive genetic material representing more robust solutions. Keymeulen et al. (2000b) applied the PFT concept for repairing FPTA configurations subjected to faults.

The PFT property is similar to measuring the initial repairability of a design subjected to a fault. If mutants are fitter than the original solution when subjected to the fault, then these mutants are on the way to a repaired solution. Knowing what kind of circuits exhibits this “instant repairability” property would help know which circuits can be repaired using Jiggling.

The question this work set out to answer are:

1. Do redundant circuits exhibit more PFT even if redundancy is added by hand? Do minimized evolved circuits still exhibit PFT?
2. Do evolved circuits have a definite advantage for PFT over hand designed equivalents?
3. Does a final population of individuals evolved in parallel exhibiting a wider range of solutions (greater Hamming distance) exhibit more PFT than a set of mutants of a single individual?
4. Does simply increasing phenotypic diversity of alternative solutions in the population lead to PFT?
5. Does a population with primitive individuals – from earlier generations in the same run – which don’t depend on cells adopted by evolved circuits later in the run exhibit greater PFT than a population with equal phenotypic diversity but with a random selection of functional individuals?

3.7.2 Method

The circuit used for this study was a two bit multiplier. The technology was two input lookup tables and the mapping was as described in §5.1. Several circuits were evolved of different sizes by balancing the application of parsimony and anti-parsimony selection pressure. Circuits were also hand designed using standard logic synthesis techniques and simplified to various degrees using Karnaugh maps. Several versions of the simplified and non-simplified versions were designed by using different versions of the Boolean formulae and different groupings in the Karnaugh maps. Redundancy was added to some circuits by duplicating calculations.
For each circuit a single fault was injected. Then 500 1-bit mutants were generated by mutating only parts of the genotype representing the circuit. Then the fitness of all mutants and the original circuit were evaluated and the number of mutants with higher fitness than the original recorded. This was performed for all faults on all circuits.

Then, populations of circuits were produced. One contained several hand designed circuits. Another contained several evolved circuits with different characteristics. Another contained the elites from several stages of one evolutionary run. Another was simply the last population from an evolutionary run. PFT was evaluated by taking an individual from the population and injecting a fault. Then the number of remaining individuals in the population with higher fitness under the same fault was recorded. This was done for all individuals in each population and all faults at shared elements.

PFT is a measure of how many individuals in the population have a better fitness than the elite when subjected to a fault. Phenotypic diversity was measured by evaluating a hash function based on circuit structure. Different genotypes encoding equal circuits would get the same hash function.

### 3.7.3 Results

The population with most PFT was that composed of diverse hand designed circuits. Of the populations generated by mutants, the one with highest PFT was that generated by a hand designed circuit with added redundancy. This discards the theory that PFT is an exclusive property of evolved circuits. Moreover, the population of mutants of circuits evolved with parsimony pressure had very low PFT, as low as that for optimised hand designed circuits. This discards the theory that PFT is a natural property of all evolved circuits. Whereas there was a definite positive correlation between amount of redundancy and PFT, there was no correlation between evolved or hand designed and PFT. However not all circuits with redundancy had higher PFT and it was not obvious what kind of redundancy led to higher PFT. For example a circuit having an exact spare copy of the original available did not have as high PFT as another which had redundancy spread across the whole design. Populations with larger phenotypic diversity than that of mutants did not always have a higher PFT. However there was a positive correlation between population diversity and PFT. The population composed of elites from various stages of the evolutionary run had a high phenotypic diversity but low PFT suggesting ancient genetic material is not as useful as alternative fully functional solutions.

### 3.7.4 PFT Study Conclusions

It can be safely concluded that hand designed circuits can exhibit PFT, specially if redundancy is added into the designs. The consequence for Jiggling is that in order to make the first repairs easier it may be better to use unoptimised versions of the functional circuit. The additional resources for the redundancy could be taken from spares and the evolutionary algorithm would later be able to optimise the design to its minimal form when spares run out. The consequence for a full population based repair technique is that the best strategy might be to use a population of diverse alternative conventional designs as a starting point for evolutionary repair. Each of these will use different resources and some may depend more or less strongly on the faulty one.

### 3.8 Conclusion

The TMR+Jiggling architecture for FPGA transient and permanent fault mitigation has been introduced and requires no golden memory for holding pre-compiled configurations Lach et al. (1998); Yu and McCluskey (2001b); Carmichael et al. (2001); Abramovici et al. (2001); Mange et al. (1996) nor an external golden microprocessor Zebulum et al. (2003); Abramovici et al. (2001); Lohn et al. (2003a); Vigander (2001). The Jiggling repair module is implemented on the same FPGA and would require only 141 CLBs to service all the benchmark circuits discussed. This
is small enough to be itself repairable by another Jiggling repair module, provided the architecture is extended to sequential circuits. This repair module would also service multiple systems, amortising the overhead.

Reliability analysis for some benchmarks shows the Jiggling system using on average 134% overhead per module can survive with 0.99 probability: 95 times the permanent local damage interarrival rate, which is 48 times longer than TMR/Simplex and 15 times longer than NMR using the same total area. Such a system would be 0.99 probable to survive a 1 every 2 years permanent local fault arrival rate after 190 years. Unlike other methods, Jiggling is capable of replacing any faulty unit with a spare, no matter where they are located within the module. Due to Jiggling’s fine grained use of on average 87% of available spares, it is shown how the number of spares in such a system can be adapted to reach desired reliability guarantees for specific times during a mission. It is also shown that the imposed timeout used to limit the computational effort of simulations makes reliability statistics pessimistic and the real figure for reliability above 0.99 is closer to 116 times the fault interarrival time.

There is reason to believe that this approach would be practical for repairing larger circuits. Related experiments §5 suggest that evolutionary algorithms are capable of finding solutions in spaces as large as $2^{10000}$ in a significantly longer search time. With an average repair time of under 5 seconds and an expected fault interarrival time of 2 years, it would not be a problem if the repair time increased by several orders of magnitude for larger circuits. However, collecting repair time statistics for larger circuits is computationally demanding because the repair process would take longer, circuit simulation time is proportional to its size and because the length of the test pattern used during evaluation varies exponentially with the number of circuit inputs. Using real hardware would get rid of simulation cost speeding the process significantly.

Jiggling was not successful on all benchmarks attempted. A method was presented to predict the effectiveness of the approach on different circuits without running any repair simulations. This can also be used to research different fitness functions and search for more repair-friendly circuit structures. A combined fitness function was presented which only judged an individual as better if it was better with one and not worse with the other of the composing fitness functions. This was found to be more effective by making the correct choice less frequently on average but also making less consistent wrong choices.

The effectiveness of several variations of the Jiggling architecture was explored together with various levels of trade-off between overhead and time. It was found that Jiggling module overhead and repair time are linearly interchangeable. It was also shown that the first Jiggling repair could be speeded up by not optimising the protected designs.

A more thorough evaluation of this architecture would involve larger benchmarks and more accurate statistics. The latter may be achieved by a less pessimistic probability model and by allowing more time for simulated repairs. As the size of benchmarks renders simulation cost prohibitive, the Jiggling system may be implemented onto a real FPGA. This would also allow a more accurate overhead analysis and its true evaluation in the presence of radiation. The reliability of Jiggling to mitigate transient as well as permanent faults should also be studied with the possibility of doing away with Scrubbing.

Further developments to the architecture are required to allow repair of sequential modules including latched data recovery. The fitness evaluation procedure for sequential systems could treat the output and next-state function generators as a combinational block to be repaired through Jiggling, while the latches could simply be replaced by other latches. Challenging aspects of extending Jiggling to repair sequential modules would include the repair of routing between the combinational block and the latches and synchronisation of repaired modules. For larger benchmarks, routing restrictions may be introduced as well as a more comprehensive fault model.
Chapter 4
Evolution across a Public Grid

It has been shown how evolution can be used to evolve functional circuit configurations in the field. For the remainder of this thesis, evolution will be used as an off-line design tool to produce static circuit designs. The computing power required for this evolutionary design process grows roughly exponentially with problem size. This chapter will introduce a way of addressing this scalability problem by distributing any evolutionary process across a grid of computers, for free.

Motivation
A guide is provided for using Open Source package IslandEv to distribute any Java based evolutionary process across an internal network or across volunteers’ computers over the Internet. Later discussion comments upon the distributed computing community and experiences gathered whilst running such a public distributed evolutionary process. After reading this chapter the reader is equipped to distribute their own evolutionary process.

4.1 Introduction
One of the main obstacles for artificial evolution is scalability. As the problem tackled grows, so does evaluation cost and the number of evaluations during evolutionary search, resulting in a roughly exponential increase in the computing power required. Larger versions of easily tackled problems are often practically impossible. Evolution in nature has run in parallel across the whole planet for billions of years and provides our only examples of large complex evolved systems.

Large amounts of untapped processing power are available these days. Most public workstations are used for about 20% of the time, and even when used only an average 1% of the processor’s power is occupied. Another example is home PCs which are constantly switched on and connected to the Internet, with perhaps only a small file server installed. An average university probably has 3000Ghz of untapped computing power. A sum of the amount of idle processing power available across corporations, universities, and homes world-wide would be staggering.

Processing intensive problem solving is sporadically distributed across multiple computers; either across a privately owned network or accepting contributions from the public across the Internet. General architectures to distribute processes semi-transparently across a private network include Beowulf clusters, openMosix(ope, 2004) and X-Grid(x-g, 2004). Several projects have been collating computing power from donors across the Internet. SETI@home(set, 2004) has been combing radio signals for extraterrestrial life since 1998, distributed.net(dis, 2004b) has cracked several cryptographic keys considered uncrackable and distributed folding(dis, 2004a; Dharsee and Hogue, 2000) aids research into protein folding. SETI@home aloneconcerts the concurrent donated processing power of half a million computers world-wide. Most projects like these use
specific architectures, but Distrit(dis, 2004c) and BOINC(boi, 2004) have recently been released as general-purpose architectures for Internet-wide distributed computing.

Evolution is well suited to be distributed across several computers because it is easy to parallelise. There are several ways of parallelising the evolutionary process. One way is to parallelise the evaluation of individuals for a single population. However such an approach requires communication between nodes after each evaluation. The island model (Muhlenbein et al., 1991) parallelises the whole evolutionary process: each computer holds a whole population evolving on its own and individuals migrate between these populations asynchronously such that migration is more likely to occur between islands closer to each other on a 2D grid. This allows full use of sparsely connected computers. The distributed GA (Husbands, 1994) approach is similar except that individuals are spatially distributed on the 2D grid competing for survival locally with high fitness individuals flowing across the grid. The distributed GA approach has been successfully applied to multi-objective optimisation but is less suitable for sparsely connected networks of computers since interactions between individuals on the grid must be frequent. One advantage of both these spatially distributed GAs is that genetic diversity can be maintained by controlling the rate at which individuals flow across the grid. Genetic diversity can avoid stasis at local optima and for this reason the use of spatial distribution may even be beneficial on a single computer. Koza (Koza and Andre, 1995) has distributed a genetic programming process across a Beowulf-like cluster of hundreds of computers. The approach adopted is similar to an island based model with a high migration rate optimised for runs with few generations on a tightly connected network.

The purpose of open source project IslandEv is to distribute any Java based evolutionary process across a network. This software is freely available to all and permanently maintained at (isl, 2004). This chapter is meant to guide the reader in using IslandEv to setup their own distributed evolutionary process across an existing group of computers and in making it accessible to contributors from the distributed computing community over the Internet. First, the IslandEv architecture is described followed by an introduction to the distributed computing community and a report of experiences gathered during large scale use of the software. A step by step programmer’s guide is provided in Appendix A.2.

4.2 Architecture

The IslandEv architecture can be understood at three levels of abstraction and the software reflects this (Table 4.1). This section considers the levels in turn.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolution</td>
<td>The level of the GA programmer using IslandEv.</td>
</tr>
<tr>
<td>Client-Server</td>
<td>Automated intercomputer communications provided by IslandEv to support the GA.</td>
</tr>
<tr>
<td>Distributed process</td>
<td>Generic task distribution provided by Distrit.</td>
</tr>
</tbody>
</table>

4.2.1 Evolution Level

IslandEv distributes a GA by running one GA process on each computer which we refer to as an island. It allocates a position on a 2D grid (see Fig. 4.1) to each island and then controls the migration of individuals between them. For multiple population coevolutionary setups migration occurs between populations of the same type. As islands are added, their grid positions are allocated spiralling outwards from the centre.

An island i will initiate a migration event every time ti, where ti usually a constant but can be configured to vary with the characteristics of the network. At this migration event, a source island is chosen randomly and an individual is chosen from the source’s population using the selection method adopted by the GA. This individual then replaces the worst individual at island i. The
4.2.2 Client-Server Level

A client-server architecture is used where each client runs the GA process. This could be any Java-based GA process. All clients connect exclusively with the server (see Fig. 4.2). The server is in charge of the following:

1. Send the evolutionary task to each client, or update it.
Figure 4.3: The Distributed Process level is implemented by DistrIT. The client requests a task from the server, runs it, and regularly performs I/O.

2. Direct the flow of migrating individuals.
3. Keep track of the fittest individuals found so far.
4. Save a log and snapshot of the population at each island.
5. Save miscellaneous data about the whole process to a MySQL database.

This system is robust to client or server failure since the state is spread across the system. If a client fails, then its genetic material will have spread to neighbours and been logged by the server. If the server fails then all islands will keep their state as they continue evolving individually, and migration will continue as soon as the server is live again.

4.2.3 Distributed Process Level
IslandEv is powered by DistrIT (dis, 2004c) which can distribute any task across multiple computers. This has the consequence that the IslandEv client is generic, so any bug fixes and updates to GA code need only be done on the server and will be automatically distributed to all clients.

4.3 First steps to distribute a GA
First the source must be downloaded for DistrIT(dis, 2004c) and IslandEv(isl, 2004) from the project websites. IslandEv can distribute any Java based GA package. Details of how this can be done are provided in appendix A.2. In order to plug-in any GA package to IslandEv two interfaces must be implemented as described in appendix A.2.1. If JaGa(jag, 2004) is used this has already been done so the reader may skip straight to appendix A.2.2.

4.4 Distributed Computing Community
The server as configured above is ready to accept processing power contributions from anyone connected to the Internet. If the server is behind a firewall, then a typical configuration requires ports 1098, 1099 and 2010 to be opened.

There is a community of people who donate their CPU time to various projects. To reach them the straightforward way to proceed is to build a website explaining what the project is about and providing information on how to download the client to contribute. This website can then be announced in relevant sites such as (asp, 2004), (fre, 2004) if open source, or in relevant news groups.

Members of this community like to have feedback about the progress of their client and how much they have contributed compared to others, and often group up in teams. Most projects provide such feedback as online rankings measuring several aspects of members’ contributions to the project. The IslandEv server is prepared to log enough data for most of these rankings to be
generated for each member and team, such as how many individuals were evaluated, how many fitness improvements to the global best were made, etc..

It is also a good idea to provide a forum for contributors of a project to help each other out and for general discussion.

4.5 Experiences

This section will describe experience collected when using IslandEv for running the distributed hardware evolution project (dhe, 2004) which accepted contributions from over 2000 people worldwide. The problem tackled was the synthesis of large circuits with fault-checking properties and results were found using less resources than conventional equivalents.

4.5.1 Monitoring the cluster

Feedback on the state of the distributed evolutionary process is required to adjust parameters to their optimal state. Graphs are instrumental as a summary of the complex state the cluster may be in. Problems such as continued loss of fit individuals, high convergence or network disconnection may be noticed at a glance.

The most straightforward graph to generate is fitness vs. time, as in Fig. 4.4 where four fitness metrics are plotted in different colours. The fitnesses plotted are those of the elite individuals as reported by each island every $t_i$. The difference in fitnesses amongst elites at any given time accounts for the scattering of fitness values. However, semi-convergence of the cluster results in an increasingly wide line. If the cluster were totally converged this line would be thin. If migration rate was too low then some thin lines would be present at the top with a large wide line at the bottom, indicating that fit individuals are not drifting enough. Loss of highly fit individuals would be displayed as high lines disappearing, indicating noise in the evaluation should be dealt with.

Connection of a large number of clients manifests itself as vertical stripes in the graph as most clients start with a low fitness population which is quickly replaced by incoming fit migrants.

Another useful graph is the colour-coded topology graph such as in Fig. 4.5 where a colour encodes the fitness of the best individual at that coordinate. For single fitness runs, this could simply range from white to black. For multi-objective runs, each RGB value could encode a fitness. Either way, this graph provides a snapshot of the current state of the cluster. A uniform
Figure 4.5: Topology graph encoding the fitness of the best individual at each island. Three fitness metrics are encoded in the RGB channels such that darker means fitter. An efficient migration rate allows a gradual flow of individuals creating areas of semi-converged ‘species’.

Graph denotes a converged cluster most likely due to overly-high migration. An uncorrelated graph would suggest that migration rate should be increased. Figure 4.5 shows a cluster with a balanced migration rate.

4.5.2 Migration and Noise
It was found that a uniform \( t_i \) for all islands in the cluster is best. The value must be high enough to protect the server from overloading. Uniform \( t_i \) will lead to fairer contributor statistics and to a uniform migration rate across the whole grid controlled entirely by \( p \) and \( d \) as in §4.2.1.

An optimal migration rate is key to the efficiency of a cluster. In our experience the ideal situation is when the cluster is neither completely converged nor completely uncorrelated, but at an intermediate state where several areas of semi-convergence can be seen across the cluster as in Fig. 4.5, as if multiple species were present.

Noise in the evaluation will affect the choice of migration rate since it acts against convergence by disturbing the selection process. For an evaluation with no noise a lower migration rate is advantageous possibly with no elitism in each population, so that a high fitness individual does not dominate the whole cluster before a better one is found. A higher migration rate is allowable for noisier evaluations so that high fitness individuals can spread effectively and are not lost.

4.5.3 GA and dynamics
It has been found best to use the genetic operators which were optimal for the single population approach when running an island based cluster. The whole process can be seen as individual searches which now and again catch up with each other and branch out again. Within the SAGA (Harvey, 1992b) framework, the semi-converged species on the islands can spread to other islands.
thence splitting into separate species and they can also become extinct upon immigration of an individual from a fitter species.

4.5.4 Community

Many members of the distributed community want to have some feedback about their contribution, such as a GUI displaying what this island is currently doing and as the rankings mentioned earlier. Many members are grouped into teams and will be more likely to join a project in which teams are supported.

4.6 Conclusions

The reader should now be able to start their own distributed evolutionary process across a private network and accepting contributions from the public. Given space limitations this guide is not complete and the reader is invited to consult an on-line guide (isl, 2004) and contact the authors to access more information on how to make IslandEv work for them.

While using IslandEv to run the distributed hardware evolution project (dhe, 2004; Garvie and Thompson, 2004), an experiment that would otherwise have taken six years on a single processor was performed in a week with donated computing power. This represents the difference between a practical experiment and an infeasible one. Readers with specialist knowledge of other application areas may be able to rethink what challenges may be tackled.

Unfortunately a linear increase in processing power does not match the non-linear demands of scaling some problems beyond an initial increment.
Chapter 5

Evolution of Digital Self-Checking Circuits

An architecture for distributing any evolutionary process across the Internet has been presented. This chapter will now use this technology to explore evolution's capacity at designing digital circuits with CED which increase hardware reliability by signalling erroneous operation.

5.1 Introduction

Circuits with Concurrent Error Detection (CED) are capable of detecting transient and permanent faults and are widely used in systems where dependability and data integrity are important (Sellers et al., 1968; Pradhan, 1986; Spainhower and Gregg, 1999). CED techniques can also enhance off-line testability and reduce BIST overhead (Gupta and Pradhan, 1996) and are increasingly relevant as shrinking process technology makes logic increasingly susceptible to radiation hazards such as Single Event Upsets.

Almost all CED techniques decompose circuits into two modules: the functional logic and the checker as in Fig. 5.1. The functional logic provides output encoded with an error detecting code (EDC) and the checker determines if the output is a codeword. The checker itself traditionally (Pradhan, 1986; Wakerly, 1978) provides a two-rail signal in which an error is indicated by both rails being at the same value. When a systematic code is adopted a codeword is constructed by appending check bits to normal output bits, allowing a further modular decomposition of the functional logic into the output function generator and the check symbol generator. A classic CED technique is duplication in which the output function generator and check symbol generator are functionally identical and the checker simply compares their output. Other systematic codes used for CED include parity-check, Bose-Lin and Berger codes, while a non-systematic code used is the m-out-of-n constant weight code. With the exception of duplication, the use of these codes for CED imposes structural constraints on the output function generator module in order to ensure deviation from normal behaviour in the presence of faults is detectable by the checker.

The behaviour desired of the functional logic in a circuit with CED is often referred to as the “totally self-checking goal” (TSCG) (Smith and Metze, 1978): to produce a non-code word as the first erroneous output due to a fault. When observing the circuit with CED as a whole including the checker, the equivalent required would be to not produce erroneous output before an error is signalled. A more precise definition by Anderson (Anderson, 1971) is that of a Totally Self-Checking (TSC) class of circuits. These fulfil the TSCG when used with a TSC checker and only under the assumption that all input words are applied between fault arrivals. No circuit has been previously published which will always fulfil the TSCG without a similar assumption on fault frequency. The probability of a TSC circuit reaching the TSCG without this assumption is studied in (Lo and Fujiwara, 1996) while (Mitra and McCluskey, 2000; Bolcini et al., 2003) propose techniques to increase it by reducing the amount of common mode failures.
Several methods to design TSC simple functional units such as datapath circuits (e.g. adders, multipliers, etc...) have been published (Sellers et al., 1968; Pradhan, 1986). Automatic synthesis methods for TSC circuits with less than duplication overhead have also been proposed. Jha and Wang (Jha and Wang, 1993) present one using a Berger code for primary outputs and an m-out-of-n code for the state. Their method requires the output function generator to be inverter free. De et. al (De et al., 1994) proposed a synthesis method for combinational TSC circuits similar to (Jha and Wang, 1993) and another using a parity-check code which constrains the output function generator to not share logic between blocks belonging to different parity groups. Touba and McCluskey (Touba and McCluskey, 1997) also proposed a technique using a parity-check code which modifies the original output function generator such that logic sharing is only allowed between parity groups. Zeng et. al (Zeng et al., 1999) presented a synthesis technique for sequential TSC circuits using parity-check codes for both primary outputs and the next state function, again imposing structural constraints on the output function generator logic. Das and Touba (Das and Touba, 1999) describe a method for combinational and sequential circuits using Bose-Lin codes for both primary output and state encoding in which the output function generator circuit is constrained to be inverter-free. Bolcini et al. (Bolcini et al., 2000) proposed a synthesis method for sequential TSC circuits with a constant Hamming distance between the codes for the present and next state and using parity-check or Berger codes for primary outputs. Morozov et al. (Morozov et al., 2000) and Goessel (Goessel et al., 2000) describe techniques for synthesis of TSC combinational circuits using modified Berger codes and 1-out-of-4 codes respectively. Mohanran and Touba (Mohanram and Touba, 2003) propose a method for synthesising self-checking circuits with significantly lower overhead but only partial fault coverage and thus not fulfilling the TSCG under the all inputs between faults assumption. Piestrak (Piestrak, 2002) points out some limitations of previous techniques for synthesis of TSC sequential circuits. One of these is that the checkers are not fully exercised during normal operation thus rendering the whole CED circuits to not be TSC.

All synthesis techniques mentioned so far impose structural constraints on the original output function generator circuit. Techniques for the design of on-line checking circuits that do not imposed structural constraints exist (Leveugle and Saucier, 1990; Parekhji et al., 1995) but have either above duplication overhead or an error detection latency of more than one clock cycle and thus do not fulfill the TSCG. In fact, no automatic synthesis technique capable of adding logic around an unmodified original output generating function circuit to make it TSC with less than duplication overhead has been previously published. Circuits not strictly adhering to the function-checker modular decomposition have been designed by hand (Sellers et al., 1968; Pradhan, 1986) but no automatic synthesis method capable of generating TSC circuits with unconstrained structure has been previously published. Previous research (Sellers et al., 1968; Piestrak, 2002; Bolcini et al., 1997, 2000) has suggested that the use of ad-hoc codes and CED techniques to suit the particularities of a circuit can produce resource efficient TSC circuits. All automatic synthesis methods previously published apply a single CED strategy to every circuit ignoring its particularities.

Evolutionary algorithms such as Genetic Algorithms (GAs) (Holland, 1975; Harvey, 1992b; Mitchell, 1998) and Genetic Programming (Koza, 1992) are widely used in engineering design.
optimisation in many variants sharing the underlying principle of selection acting repeatedly over heritable variation. Evolutionary electronics (Thompson, 1996; Tyrrell et al., 2003; Lohn et al., 2003b) applies evolutionary algorithms to hardware design and there are several examples in the literature of evolved designs exploiting the particularities of the medium to operate efficiently. The evolutionary design process has a holistic view of the circuit at hand and can select changes affecting several aspects of the circuit’s behaviour simultaneously if beneficial. Thus it is capable of generating designs with complex interactions between multiple parts which could not be understood using the traditional divide and conquer design methodology. By operating on the circuit as a whole, at a fine-grained component level, it is not constrained to adopt the modularity and encapsulation required by the standard divide and conquer methodology. Thus the evolutionary design process explores areas of design space within and beyond those containing modular designs. Such design spaces have been mostly unexplored up to now, and there is no reason to believe that non-modular designs with desirable properties do not exist. Another advantage of evolution as a design process is its capacity to adapt to the particularities of the medium at hand to produce efficient well-adapted solutions.

Section 5.2 will introduce some definitions, section 5.3 will present the synthesis method, results and analysis of combinational circuits with CED and section 5.4 will do the same for sequential circuits. Section 5.5 will introduce absolutely fault secure circuits and section 5.6 will discuss the conclusions.

5.2 Definitions and Notation

5.2.1 Definitions

Anderson (1971) defines the TSC class of circuits. A circuit is traditionally combined with a checker as in Fig.5.1 to form a CED system which is TSC if both circuit and checker are TSC. Since the circuits presented in this work do not adopt the function–checker structure, the following will define what it means for a whole CED system to be TSC. Adopting these definitions would have avoided issues pointed out in Piestrak (2002). Given a circuit $G$ and a fault set $F$.

**Definition 1:** $G$ is self-testing (ST) if, for every fault in $F$, the circuit signals an error for at least one input codeword applied during normal operation.

**Definition 2:** $G$ is fault-secure (FS) if, for every fault in $F$, the circuit never produces incorrect output without an accompanying error signal for inputs applied during normal operation.

**Definition 3:** $G$ is TSC with respect to $F$ if it is both self-testing and fault-secure.

If $G$ is sequential then normal operation includes the input vectors that would normally be applied under the states that would normally be visited.

**Definition 4:** A path is sensitised (Armstrong, 1966) from wire $w_i$ to wire $w_j$ if a change in value at $w_i$ will change the value at $w_j$.

**Definition 5:** A gate is non-canalising if any change in value at any of its inputs will always change the value of its output.

5.2.2 Notation

$g_i$ Gate $i$.

$q_i$ The output line of gate $i$.

$a_i$ The first input line of gate $i$.

$b_i$ The second input line of gate $i$. 
Chapter 5. Evolution of Digital Self-Checking Circuits

5.3 Combinational

5.3.1 Synthesis Process

The method proposed in this work for the synthesis of combinational TSC CED circuits consists of a GA searching a space of circuit designs. Each design encountered is instantiated in a digital logic simulator and evaluated on several metrics including how ST and FS they are when subjected to injected faults. These fitness metrics will influence the probability of a design being selected for reproduction into the next generation. The multiple aspects of this process will now be described in greater depth. Many of the choices described were arrived at after experimentation with a significant number of inadequate alternatives. For this work, the logic gate library consists of all possible two-input functions and the fault set includes stuck-at faults at gate inputs and outputs. These were chosen as a standard for comparison yet the approach is general for any gate library and fault set.

Genetic Algorithm

The generational GA used is in the style of Harvey (1992a) and has a fixed size population of 32 binary genotypes. Each genotype is a fixed length binary string encoding a circuit as described in Genotype to Circuit Mapping below, and each circuit has fitness measuring its quality as a circuit with CED. Once the fitness of every individual in the population has been evaluated as in §5.3.1, they are sorted in descending order. The two highest ranking individuals are elites and are copied unmodified into the next generation. Six individuals of the remaining next generation are each formed by the sexual recombination of two individuals through single point crossover: a random point \( p \) is picked and the offspring is formed by the genetic material of parent \( a \) up to \( p \) and by that of parent \( b \) after \( p \). Another 6 are formed by individuals with \( m \) point mutations (bit-flips) at random positions in the genotype. Three other individuals are generated by a block-copy mutation operation which copies a section from one part of the genotype to another. This is done so that a section representing one logic gate and its routing replaces that of another logic gate. The remaining individuals are formed by mutations affecting the encoded routing (see §5.3.1) such that a mutation could change the source of a gate to be the output of any other gate or a primary input. Excepting elites, all individuals are selected for reproduction based on their rank in the sorted population, such that the fittest individual has twice the probability of being selected than the median ranked one, and the remaining probabilities are linearly interpolated. The mutation rate \( m \) above decreases as the average fitness metric \( f_{FS} \) (see §5.3.1) increases, analogous to a Simulated Annealing process moving from “exploring” to “exploiting”. 

\[
m = \left\lfloor k_r \times \ln \left( \frac{1}{\bar{f}} \right) \right\rfloor + m_{\text{floor}}, \quad k_r = \frac{m_{\text{ceil}}}{\ln(1/f_{\min})}, \quad \bar{f} = \text{current average fitness}, \quad f_{\min} = \text{minimum possible fitness above 0}, \quad m_{\text{floor}} = 1 \text{ is the number of mutations applied as } \bar{f} \text{ reaches 1, } m_{\text{ceil}} = g/25 \text{ is the number of mutations that would be applied if } \bar{f} = f_{\min}, \quad \text{and } g \text{ is genotype length}. \]

This assumes that \( \bar{f} > 0 \) which is safe under the settings used in this work. This process of blind variation and selection usually leads to the fitness of the population increasing with subsequent generations. It is likely that as circuits increase in fitness so does the chance of a mutation affecting their behaviour so it is probable that this mutation rate keeps the average mutations affecting behaviour close to one throughout the whole run.

The GA style adopted (Harvey, 1992a) is of a small genetically converged population evolving for a large number of generations with mutation as the main driving operator. A fitness landscape is a conceptual tool where each possible genotype is a point in the landscape with a height proportional to its fitness. Points are neighbours in the landscape if there is a genetic operator that can convert one into the other. A fitness landscape is smooth if the correlation between neighbours’ fitnesses is high, and is rugged otherwise. GAs are better suited to searching smooth landscapes. The neutrality of a fitness landscape is a measure of the amount of neighbours with equal fitness and it can be beneficial to avoid stasis at local optima.
Genotype to Circuit Mapping

Every binary genotype encodes a circuit structure which is instantiated during fitness evaluation. During evolutionary synthesis circuits are constrained to have $q$ outputs, $i$ inputs and a maximum of $2^b - i$ two-input logic gates, where $b$ bits are used for a routing address. The first $q \times b$ bits define what gates will drive primary outputs. The rest of the genotype is formed by $2^b - i$ genes of $4 + 2 \times b$ bits, each gene defining the gate whose address is its position in the gene list. The $i$ largest addresses point to primary inputs. The first four bits within a gene are the truth table for the represented two-input gate, while the remaining $2 \times b$ bits encode the addresses of its inputs. Each of these $b$ bit blocks may be affected as a whole by the routing mutations described in §5.3.1.

This mapping allows for the encoding of circuits with recurrency. For some runs – these will be marked in the results section – genotypes were forced to be feed-forward as follows: a depth-first search is performed on the encoded circuit starting from each output in turn. If during the search of a single path a gate is encountered for a second time, then the loop is broken by re-routing to a random primary input instead. This mapping is more flexible and defines a fitness landscape with richer connectivity than the standard feed-forward mapping used for hardware evolution (Vassilev and Miller, 2000). Other runs allowed circuits with recurrent connections (feedback loops), but it is not yet clear whether the added neutrality in the fitness landscape benefits search as much as it is slowed down by the increased evaluation time required to capture sequential behaviour.

Circuit Simulation

In order to evaluate their fitness as self-checking circuits, the genotypes are instantiated in a simulator, their structure determined by the mapping described above. The simulator used is a simple version of an event driven digital logic simulator in which each logic gate is in charge of its own behaviour when given discrete time-slices and the state of its inputs. Logic gates may perform any two-input function.

For runs where recurrent connections are allowed circuit evaluation must capture behaviour along the time domain. Circuit state is randomised at the start of a trial and then each input word is applied for 30 time steps with outputs sampled in the last 10. Gates are updated “simultaneously” by sending the time-slices to the logic gates in two waves: the first to read their inputs and the second to update their outputs. Gate delays are simulated in time-slice units and are randomised with a Gaussian distribution $\left[\mu = 1.5, \sigma^2 = 0.5\right]$. This avoids dependence on any particular gate delay combination and facilitates transfer to real hardware.

For runs where genotypes are modified to always encode feed-forward circuits, a static evaluation of circuit behaviour is sufficient. Gates are sorted such that no gate in the list drives an input to a gate at an earlier position. All gate delays are set to 0 and gates are refreshed in a single wave in order. This time they read their inputs and update their outputs during a single refresh event. Since all data is propagated to the primary outputs after a single wave, inputs need only be applied for one time-step thus speeding evaluation considerably.

A stuck-at fault at a wire is simulated by fixing it to 0 or 1.

Output Function Evaluation

Once a genotype is instantiated as a circuit in the simulator, the evaluation of several qualities is ready to begin. This section describes the procedure used to evaluate a circuit’s ability at generating the correct output function. All $2^i$ input words are applied once in a random order to avoid circuits with recurrent connections depending on any particular order. The same random order is used for every individual in a generation and a new random order is generated for every new generation. Let $Q_r$ be the concatenated response at output $r$ sampled as described in §5.3.1 to all input words applied, and $Q^r$ be the desired response. Then output function fitness $f_f$ is the modulus of the correlation of $Q_r$ with $Q^r$ averaged over all $q_f$ function outputs:

$$f_f = \sum_{r=0}^{q_f-1} \frac{|\text{corr}(Q_r, Q^r)|}{q_f}$$ (5.1)
Allowing inverted outputs improves the smoothness and connectivity of the fitness landscape while remaining trivial to change an evolved circuit’s outputs to be non-inverting if required. A circuit with \( f_f = 1 \) generates the output function correctly.

### Evaluation of Self-Checking Ability

In order to gauge a circuit’s quality at CED, the behaviour of the extra outputs forming the two-rail error signal \( \delta \) is observed in the presence of injected faults. Fitness metrics \( f_{ST} \) and \( f_{FS} \) measure the degree to which a circuit fulfills the ST and FS criteria. If an error is signalled – ie. \( \delta \) – during fault-free evaluation of \( f_f \) then \( f_{ST} = f_{FS} = 0 \). Otherwise the same evaluation procedure described in §5.3.1 is repeated under every fault in the set of gate output faults \( F_q \) and \( f_{ST} \) and \( f_{FS} \) are calculated as follows:

\[
\begin{align*}
    f_{ST} &= \frac{1}{1 + u_f k_{ST}} \quad \text{where } u_f \text{ is the number of faults for which an error was never signalled, and } k_{ST} \text{ was chosen to be 25.} \\
    f_{FS} &= \frac{1}{1 + u_q k_{FS}} \quad \text{where } u_q \text{ is the number of fault-input word instances for which output was incorrect and an error was not signalled, and } k_{FS} \text{ was chosen to be 200.}
\end{align*}
\]

Constants \( k_{ST} \) and \( k_{FS} \) were chosen to give \( f_{ST} \) and \( f_{FS} \) good sensitivity when \( u_f \) and \( u_q \) are small. It is now informally demonstrated how a circuit which achieves maximum fitness under these metrics must perform TSC CED:

**Theorem 1.** A circuit \( G \) with \( f_f = f_{ST} = f_{FS} = 1 \) when evaluated with fault set \( F \) is a circuit with TSC CED with respect to \( F \).

**Proof.** Since \( f_f = 1 \) and \( f_{ST} > 0 \) then the output function is correct and no errors are signalled during fault-free operation. Since \( f_{ST} = 1 \) then \( u_f = 0 \) and there is no fault for which an error is not signalled during normal operation. Hence \( G \) is ST by Definition 1. Since \( f_{FS} = 1 \) then \( u_q = 0 \) and there is no instance during normal operation for which an incorrect output is not accompanied by an error signal. Hence \( G \) is FS by Definition 2. Thus \( G \) performs TSC CED by Definition 3.

In order to reduce computational effort required during self-checking ability the fault set \( F_q \) used only includes faults at gate outputs. However,

**Theorem 2.** If a circuit \( G \) is FS with respect to all gate output faults \( F_q \) then it is also FS with respect to all gate input and output faults \( F \).

**Proof.** A stuck-at fault \( f_i \) at an input of a gate \( g \) will, depending on circuit input word \( x \), either invert the output of \( g \) or leave it unchanged. If unchanged then circuit output is unchanged and correct. If the output of \( g \) is inverted then \( f_i \) cannot affect any other gate and manifests itself as a gate output fault stuck-at the inverted value. Since \( G \) is FS with respect to \( F_q \) then incorrect circuit output will not be produced without an accompanying error signal. Since this holds for an arbitrary fault \( f_i \), gate \( g \) and input \( x \), then \( G \) is FS with respect to all faults at gate inputs.

Unfortunately this does not hold for ST because the full exercising of every gate during normal operation is not guaranteed. However the ST property of gate input faults was measured during simulation under gate output faults as follows. A gate input fault will generate an error signal under an input word if and only if it manifests itself as a gate output fault at the same gate which generates an error signal. Whenever a gate output fault generates an error signal under some input word during evaluation, the gate input faults that would have manifested themselves as that gate output fault is recorded. This is done by removing the gate output fault and recording which gate input inversions restore the faulty output value. All gate input faults which never manifest themselves as gate output faults signalling an error were counted in \( u_f \) when calculating \( f_{ST} \). Thus all combinational results presented – with \( f_{ST} = f_{FS} = 1 \) – are circuits with TSC CED with respect to all faults at gate inputs and outputs.
Combined Fitness and Parsimony

The evaluation procedures above generate fitness metrics measuring a circuit’s quality at generating the output function \(f\) and at CED \(f_{ST}, f_{FS}\). A final metric \(f_p\) measures the size \(s\) of a circuit such that \(f_p\) varies from 0 (maximum size) to 1 (size 0) as \(f_p = \frac{M - s}{M}\) where maximum size \(M = 2^b - 1\) (see §5.3.1). No functional circuit will have \(f_p = 1\), yet since the smallest possible implementation of a circuit with TSC CED is not known, the evolutionary synthesis process was allowed to increase \(f_p\) as far as possible within the practical constraints imposed by processing power available.

Even though optimisation of all fitness metrics is desirable, evolution is encouraged to prioritise generating the correct output function above performing self-checking, and the latter above minimising overhead. This is achieved by defining a fitness vector \((f, f_{ST}, f_{FS}, f_p)\) and sorting the population for rank selection according to the dictionary total ordering defined, such that later metrics are only compared when earlier ones are equal. The effect of this is to automatically shift main selection pressure to the next fitness function as soon as previous ones have been fully optimised thus defining an incremental evolution (Harvey et al., 1994) path. Even though at any moment there will be only one unoptimised metric with highest priority driving selection, when this metric is equal for two individuals the GA will “look ahead” and use the next (lower) priority metric to judge which individual is best. This is rather like a driver attending to immediate demands but at the same time keeping in mind her long term destination.

Seeding and Locking the Original Output Function Generator

Evolutionary runs performed for this work were initiated with a population of individuals encoding an output function generator circuit synthesised by Sis (E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and Sangiovanni-Vincentelli, 1992) using the procedure described in §A.3. Genetic material of each individual not involved in the encoding of the initial circuit was randomised. For some runs, which will be referred to as locked, this original output function generator circuit was fixed so that no genetic operator could modify it.

Grid Evolution

In order to harness the idle processing power of several workstations to aid the synthesis process, it was distributed using an island based coevolutionary model (dhe, 2004)§4. One evolutionary process as described in the previous sections was setup on each processor which was assigned a location on a 2D grid. Individuals were selected for emigration using the same scheme as for reproduction (see §5.3.1). Their destination was random such that the probability of choosing an island varied inversely with its distance to the migrant’s source. The probability of each migration event succeeding was configured to maintain genetic diversity across populations yet allow the diffusion of fit individuals.

5.3.2 Results

Overhead Comparison

All circuits presented in this section achieve \(f_f = f_{ST} = f_{FS} = 1\) when evaluated with the fault set including stuck-at gate inputs and outputs and therefore perform TSC CED by Theorem 1. Further proof will be presented during the analysis of individual circuits.

Circuits synthesised with the proposed method will be compared to equivalents in the literature mainly in terms of overhead. For those circuits evolved allowing modifications to the original design, no timing requirements are imposed, usually resulting in an increased delay from primary inputs to primary outputs. However timing is seldom discussed in previous publications of synthesis methods, most of which also incur performance penalties due to the structural constraints imposed. Structurally unconstrained circuits with CED such as those presented in this work do possess an advantage concerning timing: the error latency may be lower than the function latency since the error signal may be generated by abnormal state at internal nodes. If certain timing
requirements were desirable then the evolutionary synthesis process could be augmented for example by including an extra fitness metric similar to $f_p$ which increased as the maximum delay between primary inputs and outputs decreased. Circuits evolved with the output function generator locked incurred no performance penalties on the original circuit, leaving the original specification intact.

The gate library including all two-input functions was chosen because it is simple, general and unconstrained. Overhead comparisons are made with duplicated circuits synthesised using the recommended Xilinx scripts for LUT technology using Sis. This will ensure a fair comparison since these tools exploit the full range of gates available. Cell libraries such as G10-p used in previous work also include a rich variety of gates, some with three inputs, which the evolutionary process might also find useful for building self-checking circuits. The synthesis method proposed here can be used with any cell library by adapting the mapping in §5.3.1 and most probably unleashing evolution’s inherent capacity at exploiting the library’s particularities. Choosing an unconstrained mapping also allows analysis of what kind of gate the evolutionary process finds useful for building self-checking circuits.

Hardware cost is measured in this work in terms of gate count, which, as the literal count or gate area used in several previous publications, ignores area required for routing. Previous work (De et al., 1994) has suggested that certain synthesis techniques may be more favourable than others for certain circuits when routing area is considered even when resulting in a higher gate count. Moreover different technologies may have different routing constraints (e.g., FPGA, ASIC) more suitable for certain self-checking strategies and certain circuits. By adapting $f_p$ to measure total area cost including routing, the evolutionary synthesis process would be able to find a suitable ad-hoc self-checking technique adapted to the particularities of the circuit at hand and the routing constraints and demands. Exploration of the evolutionary synthesis of self-checking circuits with different routing constraints is left for future work.

**The Benchmarks**

In order to provide a meaningful evaluation of the proposed technique, circuits belonging to the MCNC’91 and Espresso PLA example set appearing most frequently in previous papers were adopted as benchmarks. The benchmarks selected are listed in Table 5.1 together with their structural properties and a comparative measure of the computational cost during evolutionary search of one fitness evaluation. The main limitation of the evolutionary approach to hardware design is lack of scalability due to the computing cost of fitness evaluation and evolutionary search. Having successfully explored spaces of size $2^{20000}$ evaluation cost was found to be the limiting factor in current work. This cost – listed in Table 5.1 as **Effort** – is dominated by the simulation time which, for evaluating a circuit under the scheme described in Circuit Simulation in §5.3.1 with $s$ gates and $i$ inputs requires $2^i \times s \times (2s + 1)$ individual gate refresh events – $O(2^i s^2)$. Circuit size is calculated to be during evolution on average 1.5 times larger than the original output function generator seed. The amount of computing power required to evaluate circuits with effort above 2000M prohibited the application of the proposed method. However Moore’s law applied to processor speed, research addressing scalability of hardware evolution (Torresen, 2003; Vassilev and Miller, 2000; Louis, 2003), evaluation in real hardware, and grid computing (dhe, 2004), would all contribute to pushing this boundary further.

**The Unconstrained Evolved Circuits with TSC CED**

The computational cost for the evolutionary synthesis of the circuits presented in this section varied from a couple of hours on a single 2.8Ghz processor to a week on roughly 150 2.8Ghz processors.

The evolutionary synthesis process described in §5.3.1 arrived at circuits with TSC CED for all benchmarks for which the computational cost of fitness evaluation did not render the approach impractical. Circuits with TSC CED were synthesised for each benchmark both allowing modifi-
Table 5.1: List of combinational benchmarks with their inputs, outputs, literals in factored form and circuit size after synthesis and optimisation with Sis into ‘all two input gates’ technology §A.3. Also listed is a measure of the computational effort required for each circuit evaluation.

<table>
<thead>
<tr>
<th>Name</th>
<th>I</th>
<th>Q</th>
<th>Lits.</th>
<th>Gates</th>
<th>Effort (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult2</td>
<td>4</td>
<td>4</td>
<td>28</td>
<td>7</td>
<td>0.0035</td>
</tr>
<tr>
<td>b1</td>
<td>3</td>
<td>4</td>
<td>17</td>
<td>5</td>
<td>0.9</td>
</tr>
<tr>
<td>C17</td>
<td>5</td>
<td>2</td>
<td>12</td>
<td>6</td>
<td>5.12</td>
</tr>
<tr>
<td>cm82a</td>
<td>5</td>
<td>3</td>
<td>55</td>
<td>10</td>
<td>13.92</td>
</tr>
<tr>
<td>cm42a</td>
<td>4</td>
<td>10</td>
<td>35</td>
<td>18</td>
<td>22.22</td>
</tr>
<tr>
<td>wim</td>
<td>4</td>
<td>7</td>
<td>103</td>
<td>22</td>
<td>33.07</td>
</tr>
<tr>
<td>dc1</td>
<td>4</td>
<td>7</td>
<td>56</td>
<td>29</td>
<td>57.26</td>
</tr>
<tr>
<td>cm138a</td>
<td>6</td>
<td>8</td>
<td>35</td>
<td>16</td>
<td>70.38</td>
</tr>
<tr>
<td>rd53</td>
<td>5</td>
<td>3</td>
<td>182</td>
<td>23</td>
<td>72.25</td>
</tr>
<tr>
<td>decod</td>
<td>5</td>
<td>16</td>
<td>68</td>
<td>26</td>
<td>92.17</td>
</tr>
<tr>
<td>p82</td>
<td>5</td>
<td>14</td>
<td>211</td>
<td>81</td>
<td>887</td>
</tr>
<tr>
<td>m1</td>
<td>6</td>
<td>12</td>
<td>233</td>
<td>64</td>
<td>1108</td>
</tr>
<tr>
<td>rd73</td>
<td>7</td>
<td>3</td>
<td>741</td>
<td>49</td>
<td>1301</td>
</tr>
<tr>
<td>misex1</td>
<td>8</td>
<td>7</td>
<td>119</td>
<td>47</td>
<td>2395</td>
</tr>
<tr>
<td>bw</td>
<td>5</td>
<td>28</td>
<td>553</td>
<td>139</td>
<td>2606</td>
</tr>
<tr>
<td>5xp1</td>
<td>7</td>
<td>10</td>
<td>391</td>
<td>85</td>
<td>3905</td>
</tr>
<tr>
<td>inc</td>
<td>7</td>
<td>9</td>
<td>348</td>
<td>92</td>
<td>4573</td>
</tr>
<tr>
<td>Z5xp1</td>
<td>7</td>
<td>10</td>
<td>436</td>
<td>103</td>
<td>5729</td>
</tr>
<tr>
<td>x2</td>
<td>10</td>
<td>7</td>
<td>71</td>
<td>37</td>
<td>5950</td>
</tr>
<tr>
<td>f51ml</td>
<td>8</td>
<td>8</td>
<td>169</td>
<td>75</td>
<td>6083</td>
</tr>
<tr>
<td>sao2</td>
<td>10</td>
<td>4</td>
<td>498</td>
<td>122</td>
<td>64270</td>
</tr>
<tr>
<td>clip</td>
<td>9</td>
<td>5</td>
<td>1026</td>
<td>193</td>
<td>80339</td>
</tr>
<tr>
<td>alu2</td>
<td>10</td>
<td>6</td>
<td>453</td>
<td>252</td>
<td>273817</td>
</tr>
<tr>
<td>tcon</td>
<td>17</td>
<td>16</td>
<td>40</td>
<td>24</td>
<td>322028</td>
</tr>
<tr>
<td>pm1</td>
<td>16</td>
<td>13</td>
<td>85</td>
<td>36</td>
<td>360571</td>
</tr>
</tbody>
</table>

Table 5.2: Overhead comparison of unconstrained evolved \(E_U\), locked evolved \(E_L\) and previous best PB in literature. All circuits are TSC except those from (Gossel et al., 2000) which are partially self-checking.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>(E_U)</th>
<th>(E_L)</th>
<th>Effort</th>
<th>(E_U)</th>
<th>(E_L)</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Oh. gate count</td>
<td>Oh./Dupl. Oh.</td>
<td></td>
<td>PB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>8</td>
<td>8</td>
<td>25</td>
<td>0.32</td>
<td>0.32</td>
<td>.74</td>
</tr>
<tr>
<td>b1</td>
<td>3</td>
<td>5</td>
<td>23</td>
<td>0.13</td>
<td>0.22</td>
<td>.74</td>
</tr>
<tr>
<td>C17</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>0.67</td>
<td>0.75</td>
<td>.75</td>
</tr>
<tr>
<td>cm82a</td>
<td>8</td>
<td>9</td>
<td>22</td>
<td>0.36</td>
<td>0.41</td>
<td>.89</td>
</tr>
<tr>
<td>cm42a</td>
<td>10</td>
<td>12</td>
<td>72</td>
<td>0.14</td>
<td>0.17</td>
<td>.53</td>
</tr>
<tr>
<td>wim</td>
<td>8</td>
<td>-</td>
<td>58</td>
<td>0.14</td>
<td>-</td>
<td>.53</td>
</tr>
<tr>
<td>dc1</td>
<td>17</td>
<td>65</td>
<td>0.26</td>
<td>0.14</td>
<td>0.17</td>
<td>.60</td>
</tr>
<tr>
<td>cm138a</td>
<td>9</td>
<td>10</td>
<td>58</td>
<td>0.16</td>
<td>0.17</td>
<td>.81</td>
</tr>
<tr>
<td>rd53</td>
<td>5</td>
<td>29</td>
<td>0.17</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>decod</td>
<td>16</td>
<td>20</td>
<td>116</td>
<td>0.14</td>
<td>0.17</td>
<td>.60</td>
</tr>
<tr>
<td>rd73</td>
<td>6</td>
<td>47</td>
<td>0.13</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td></td>
<td></td>
<td></td>
<td>0.24</td>
<td>0.32</td>
<td>0.72</td>
</tr>
</tbody>
</table>
cations to the original output function generator circuit and keeping it locked. Thus the approach becomes the first automatic synthesis method for making an existing circuit have TSC CED without any modifications to the original output function generator logic with less than duplication overhead. Checking logic is appended around the existing design such that the only structural change to the output function generator is the extra routing required to feed the checking logic. The consequences of being able to make existing designs have TSC CED with such minimal structural change at less than duplication cost are evident. For an example of such a circuit see Circuit C below.

Table 5.2 compares the overhead of evolved circuits with TSC CED to that required for duplication. Duplication overhead includes one extra copy of the original circuit (with all outputs inverted at no extra cost) and the required amount of two-rail checkers to check all outputs and produce a single two-rail error signal. Each two-rail checker requires 6 gates, so for a circuit with \( g \) gates and \( q \) outputs duplication overhead amounts to \( g + 6(q - 1) \) gates. It then compares the overhead required as a fraction of duplication overhead for the proposed locked and unconstrained evolutionary techniques and for previous techniques found in the literature. Only in the cases of rd53 and rd73 did evolution optimise the output function generator logic beyond the version synthesised by Sis. This smaller output function generator is then used to calculate duplication overhead instead of the larger one produced by Sis. It is impossible to know when a circuit is maximally optimised (Gunther and Drechsler, 1999) but the fact that for all other circuits evolution could not beat Sis shows that the latter was reasonably good.

For all benchmarks attempted, circuits produced with the proposed unconstrained approach require less overhead as a fraction of duplication overhead than all previously published techniques (Jha and Wang, 1993; De et al., 1994; Touba and McCluskey, 1997; Das and Touba, 1999; Morozov et al., 2000; Gossel et al., 2000) by a considerable margin. This is true even for all of those circuits for which the original output function generator design was not modified. It must be noted that these circuits are by no means the best evolution could find since informal tests showed that leaving it run for longer allowed it to find smaller solutions. This clearly could not go on forever but it is an open question as to what the theoretically minimum overhead is for TSC CED and if evolution can find it. Results shown here are produced by allowing evolution to run within the constraints of processing time available. Fully evolved circuits required on average 22% of the overhead required by the duplication approach. This was equivalent to on average an overhead of 64% of the logic used for the original function generating logic. Circuits evolved with no modifications to the original design required on average 32% duplication overhead, or 94% overhead with respect to the locked function generator. Such low overhead figures are unheard of in the literature. Immediately several questions spring to mind. How can it be that evolved designs require so little overhead? How do they operate?

Simple inspection of evolved designs such as that in Fig. 5.2 reveals their structure is not constrained to the function-checker model. Thus the proposed method becomes the first automatic synthesis technique to generate circuits with TSC CED not adopting the function-checker decomposition. This opens up a whole class of circuits with TSC CED to the evolutionary design process, some previously discussed in the literature (Sellers et al., 1968; Mohanram and Touba, 2003). Belonging to this class are circuits such as that in Fig. 5.3 in which the function-checker distinction is blurred and individual gates are used for calculating primary outputs and controlling the propagation of error signals. Also belonging to this class are circuits with dependency between the outputs such that only some need be checked, or circuits where the function-checker is roughly adopted but the checker checks internal lines as well as outputs. Clearly not all circuits belonging to this class will have lower checking overhead than those with the function-checker decomposition but the properties mentioned above explain how many of them do. The operating principles

1Note that adding logic around a fixed existing circuit to make it ST relies on it being irredundant (Armstrong, 1966).

2this is done in (Mohanram and Touba, 2003) for partially self-checking circuits.
Chapter 5. Evolution of Digital Self-Checking Circuits

behind evolved circuits with TSC CED and arbitrary structure will be presented at greater depth in the next sections.

Further inspection of evolved circuits reveals that they use multiple strategies to achieve TSC CED behaviour, across different circuits and even within the same one. For example the circuit in Fig.5.2 uses a strategy akin to parity group checking while the one in Fig.5.3 uses mostly output cascading and non-canalising gates to achieve CED. The strategies used by evolved designs are not random but seem to be well adapted to the particularities of the original circuit. Thus the proposed technique becomes the first capable of synthesising circuits with TSC CED adopting multiple strategies suitable to the particularities of the circuit at hand. As shall be demonstrated below some of the strategies employed by evolved designs are to the writers’ knowledge novel, and efficiently tailored for the circuit at hand. Thus the proposed technique is capable of creating novel ad-hoc strategies to achieve TSC CED exploiting the particularities of the original circuit. The advantage of using ad-hoc TSC strategies for particular circuits have been previously discussed (Sellers et al., 1968; Piestrak, 2002; Bolcini et al., 1997, 2000) but this is the first automatic synthesis method to design them.

Given these claims, the reader may not find it surprising any more that such overhead gains are available to evolved designs. To back them up, the following sections will present several evolved circuits together with proof of their TSC CED ability and analysis of the principles behind their operation. The circuits chosen for analysis are small enough to be presentable in this publication yet complex enough to substantiate the claims above. Study of larger evolved designs whose overhead figures are quoted in Table 5.2 would provide deeper insight into the principles of operation at work in evolved circuits with TSC CED.

The following theorem will be used to prove a circuit performs TSC CED by observation of its structure:

**Theorem 3.** Consider a circuit $G$ for which (d) outputs $z_0$ and $z_1$ always differ during normal operation. If every line $l \in L$ in circuit $G$ has a path sensitised to either $z_0$ or $z_1$ for (a) at least one input word making $l = 0$ and another making $l = 1$ and (b) every word which sensitises a path from $l$ to any primary output, then $G$ has TSC CED with respect to all faults at lines in $L$.

**Proof.** By (a) a stuck-at-0 and a stuck-at-1 at $l$ will cause an error signal ($z_0 = z_1$ by (d)) for some input vector, so $G$ is ST. By (b) if an error caused by a fault is propagated to any output then an error is signalled. Hence no incorrect output is ever present without an error signal and $G$ is FS, and $G$ has TSC CED.

For this work $L$ contains all gate input and output lines.

**Circuit A - Unconstrained cm42a**

Circuit A (Fig.5.2) performs the output function defined by the cm42a benchmark with 4 inputs and 10 outputs and has been synthesised using the evolutionary process described in §5.3.1 allowing modifications to the original seeded output function generator logic. This logic was itself synthesised by Sis and used 18 gates. Gate addresses were chosen to be represented by 6 bits allowing a maximum circuit size of 60 gates during synthesis. CED using duplication would require 18 gates for the copy and 54 for the checker totalling 72 gates overhead. Circuit A is the fittest individual produced by the evolutionary process. Circuit A has 10 gates of checking overhead. The maximum delay from function inputs to outputs increased from 3 to 5 gate delays and the maximum error latency is 4 gate delays.

To prove circuit A has TSC CED, it will be shown how it meets the (d), (a) and (b) conditions from Theorem 3. (d) is met because the simplified formula for $z_0 = \overline{x_0} + \overline{x_1} + x_2 + x_3$ and $z_1 = x_0 x_1 x_2 x_3$ so $z_0 = z_1$ during fault-free operation. Table 5.3 shows how (a) is met by every gate input and output line in circuit A, and how (b) is met by every gate output line. This shows that circuit A is ST with respect to all gate input and output faults, and FS with respect to all gate output faults.
Table 5.3: Proof that circuit A meets the (a) and (b) requirements of Theorem 3.

<table>
<thead>
<tr>
<th>Line</th>
<th>Sens. path to</th>
<th>by input</th>
<th>is 0</th>
<th>is 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q11 = y0</td>
<td>z0</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q10 = y1</td>
<td>z0</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q13 = y2</td>
<td>z0</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q27 = y3</td>
<td>z1</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q6 = y4</td>
<td>z0</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q2 = y5</td>
<td>z0</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q2 = y5</td>
<td>z0</td>
<td>x1</td>
<td>x1</td>
<td>x1</td>
</tr>
<tr>
<td>q15 = y6</td>
<td>z0</td>
<td>all</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q9 = y7</td>
<td>z0</td>
<td>all</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q12 = y8</td>
<td>z0</td>
<td>all</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q14 = y9</td>
<td>z0</td>
<td>all</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a11</td>
<td>q11</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b11</td>
<td>q11</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a10</td>
<td>q10</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b10</td>
<td>q10</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a13</td>
<td>q13</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b13</td>
<td>q13</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a27</td>
<td>q27</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b27</td>
<td>q27</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a16</td>
<td>q16</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b16</td>
<td>q16</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a2</td>
<td>q2</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b2</td>
<td>q2</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a15</td>
<td>q15</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b15</td>
<td>q15</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a9</td>
<td>q9</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b9</td>
<td>q9</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a12</td>
<td>q12</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b12</td>
<td>q12</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>a14</td>
<td>q14</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>b14</td>
<td>q14</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q5</td>
<td>q11</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
<tr>
<td>q5</td>
<td>q16</td>
<td>x0</td>
<td>x0</td>
<td>x0</td>
</tr>
</tbody>
</table>
by Theorem 3. This is enough to prove circuit A is FS with respect to all gate input faults by Theorem 2 and thus is has TSC CED with respect to the full fault set.

The first ten rows in Table 5.3 indicate that all input words sensitise a path from all function output lines to either \( z_0 \) or \( z_1 \); in most cases through non-canalising gates, in the case of \( y_5 \) through other outputs or \( g_{25} \). Thus (b) holds for function output lines. These rows also list the input words making each line high or low demonstrating how (a) holds for these lines. Next, the table lists the lines which feed the function outputs either directly or indirectly, and to which function output a path is sensitised under what inputs. It can be observed that none of the gate output lines ever has a path sensitised to more than one function output under the same input, and that no path is sensitised to \( z_0 \) nor \( z_1 \) that does not go through a function output. This shows (b) holds for these gate output lines because each path to a single function output will always be extended by the constant sensitised path from function outputs to \( z_0 \) or \( z_1 \) mentioned earlier. Showing that for every gate input and output line there are input words for which they take low and high values when a path is sensitised to a function output similarly shows (a) to be true. Most lines not feeding function outputs (meeting (b) by definition) form a tree of non-canalising gates ending in \( z_0 \) and always have a sensitised path to \( z_0 \). They are shown to take low and high values during normal operation thus meeting (a). \( g_{22} \) is shown to meet (a) and (b) separately completing the proof.

At first sight of Fig. 5.2 a rough division between output function generator and checking logic is discernible. However the “checker” reads the output of internal \( g_3 \) and hence the design does not adopt the function-checker structural constraint. The NXOR tree fed by most primary outputs is akin to parity-checking specially considering the fact demonstrated above that any fault will only affect at most one primary output. The difference between the scheme adopted and parity-checking is that an explicit parity bit is not generated to ensure even or odd parity, instead the parity of a group of outputs is made to be always opposite to the value at \( y_3 \). Another difference with
strict parity-checking is that not all outputs are checked since $y_5$ feeds the NXOR tree through other function outputs. Observation of an ancestor of circuit A revealed that it evolved from circuits in which all outputs fed the NXOR tree directly. However parsimony pressure must have selected variants in which less outputs were checked. This is achieved by calculating $y_8$ and $y_9$ from $y_5$ using non-canalising gates where possible. The strategy of cascading outputs with non-canalising gates for achieving low overhead self-checking is present in several evolved designs, such as Fig.5.3 and (Garvie and Thompson, 2003a). This strategy is combined with one similar to parity-checking to achieve efficient CED demonstrating the use of multiple strategies within a single circuit. If run for more generations, the evolutionary synthesis process might have found similar alterations reducing overhead even further. Given the fact that the original design contained many outputs with paths usually sensitised to only one output at a time, parity-checking is a well-adapted strategy to the particularities of this circuit. The output cascading used is also well-adapted to the last three primary outputs. This shows how the evolutionary synthesis process is capable of applying multiple suitable TSC strategies to the design of a circuit with TSC CED not adopting the function-checker structural constraint.

Circuit B - Unconstrained cm82a

Circuit B was also evolved using the fully unconstrained approach from the seed synthesised for the cm82a benchmark which used 14 gates. CED by duplication would require 14 gates for the copy and 12 for the checker totalling 26 gates overhead. 6 bits were used per address allowing a maximum circuit size of 59 gates during synthesis. Circuit B performs TSC CED with 4 gates overhead. Maximum delay from function inputs to outputs was increased from 7 to 9 gate delays and maximum error latency is 6 gate delays.

Circuit B is TSC by Theorem 1. An additional proof as that in Table 5.3 will not be provided for space limitations. When observing Circuit B we are met by a non-modular structure with no clear division between function and checking logic. Some gates seem to be generating the function and controlling the propagation of errors simultaneously. This accounts for the above average proportion of non-canalising gates in the design because they always propagate errors. Apart from the previously encountered outputting cascading with non-canalising gates strategy, the structure is highly unintuitive. This is because there is high inter-dependency across it and it is hard to understand through decomposition. It may be simpler to comprehend this structure by thinking of error propagation at the gate level or block level §5.3.3. Only two gates delays were added to the maximum function latency by the chaining of outputs, even when timing was not
considered throughout the synthesis process. Such low performance hit from chaining all outputs is because there already was a fair amount of reuse of logic by calculating some outputs using others in the output function generator synthesised by Sis. Thus it can be said that evolution found a well-adapted strategy for the particularities of the original circuit and added TSC CED with only four extra gates above the original 14.

**Circuit C - Locked Multiplier**

Circuit C was evolved from a two bit multiplier found in (Miller et al., 2000) which uses 7 gates. The evolutionary synthesis process kept this original output function generator locked so that it could not be modified by mutations. 5 bits were used for addresses allowing a maximum circuit size of 28. CED by duplication would require 7 gate for the copy and 18 for the checker totalling 25 gates overhead. The evolutionary process added 8 gates around the original design to produce a TSC CED circuit.

First observation of Fig.5.4 reveals that the unmodified output function circuit is, as could be no other way, clearly separable from the checking logic. However the traditional functional logic-checker structure is not adopted since the checking logic reads several internal lines.

Checking logic does not read all outputs either which is allowed by the cascading outputs present in the original circuit. Here it is evident how the checking logic has adapted around the particularities of the original output function generator circuit. The large amount of non-canalising gates in the checker suggests some kind of parity-checking of outputs and internal nodes may be going on, but it may be more productive to understand circuit C’s operation in terms of error propagation. The checker is designed such that every error at any output will be propagated to $z_0$ or $z_1$ with the contraint that $z_0 = z_1$ during normal operation. It is not unreasonable to think that a method not powered by an evolutionary process could synthesise TSC CED circuits by focusing on these constraints.

**5.3.3 Learning from Evolved Designs**

This section will begin to explore the possibility of extracting design principles from evolved designs. It is meant to demonstrate that this can be done rather than to exhaustively collate the whole breadth of principles used in them. Such knowledge could be applied to develop non-evolutionary synthesis methods for making certain types of circuits TSC CED.

The main advantage of using the evolutionary design approach, ie. that an efficient ad-hoc solution is found exploiting the particularities of the circuit structure blending the function logic
Figure 5.5: Examples of error propagating blocks extracted from evolved designs. Errors are always propagated from an input to a single output. Errors generated by faults within the blocks also propagate to a single output.

with the checking logic seamlessly, in some sense goes against our capacity to learn from these designs. Being used to a divide and conquer approach to understanding and designing structure it is hard for us to gather design principles from a structure designed by something that was "aware" of all its particularities simultaneously throughout the design process. The more evolution was capable of breaking down the difference between function and checking logic and of adapting self-checking strategy to a design the harder it will be for us to extract general design principles.

Error Propagating Blocks

However, as was hinted at earlier, it may be appropriate to think about these circuits at the lower level of error propagation instead of codes and functional decomposition. For example non-canalising gates are often used in these designs which clearly propagate all errors at inputs to their output. Closer observation reveals that higher level structures with error propagation properties are also present in evolved designs. One example is the two-gate structure extracted from the self-checking adder in (Garvie and Thompson, 2003b) and displayed in Fig. 5.5a, for which an error at \( w_i \) or \( w_j \) will always be propagated to exactly one of \( w_k \) or \( w_l \). Also, any error generated within this block will be propagated to at most one block output. Similarly the block in Fig. 5.5b, extracted from Circuit A with gate 3 inverted for clarity, will propagate errors in \( c \) to exactly one output. If we imagined a similar block added below but without the inversion of \( c \) at gate inputs, then all errors at any block input would always be propagated to exactly one block output. Also all errors generated within this block would propagate to exactly one output. This structure could be enhanced further by having four such blocks, each with \( cd, cd, \bar{c}d, \bar{c}d \) as the second input. This enhanced version can be found in the evolved self-checking circuit for the decod benchmark and is likely to be useful for decoders in general. A similar structure to that in Fig. 5.5a can be observed with gates 7,8,9,15 in circuit B only that here errors are always propagated to both or none of 9 and 15 but since they always propagate to 17 then errors only ever reach exactly one of \( z_0 \) or \( z_1 \). It is not hard to imagine other such blocks based on the same principles and such a set could easily be generated.

These error propagating blocks could now be combined to form a TSC circuit in the following manner. Since they all propagate all errors at an input and all errors caused by internal faults to exactly one output, then when combining the blocks together (including non-canalising gates) it is guaranteed that all faults generated will be propagated to exactly one of the outputs which do not feed any other blocks. If the circuit were constrained so that these outputs are two – \( z_0 \) and \( z_1 \) – and so that \( z_0 \neq z_1 \) in the absence of errors, then all errors would generate the error signal \( z_0 = z_1 \). Function outputs would be taken from internal nodes in this structure and the blocks combined so that these functions are generated parsimoniously. This whole collection of blocks would be FS since all errors lead to \( z_0 = z_1 \). It would be ST with respect to gate output faults because the blocks are irredundant. The ST property with respect to gate input faults is not always guaranteed
and would have to be another explicit constraint. It has been shown how error propagating blocks could be combined to generate a circuit with TSC CED.

A less constrained variant on the approach described above would be to allow blocks of irredundant unconstrained logic to feed the inputs of the error propagating blocks such that between function outputs and the error signals there are only error propagating blocks. Clearly, the above constraints of $z_0 \neq z_1$ and ST of gate input faults would have to be maintained. The resulting circuit would still have TSC CED because all errors at outputs would always cause an error signal (FS) and all unconstrained blocks are irredundant and feed only one input of all error propagating blocks (ST).

In order to synthesise circuits automatically using these extracted design principles, an existing synthesis process could be instructed to use these blocks as a technology to map to while maintaining the constraints such as $z_0 \neq z_1$ described above. Alternately, a GA could be used with these blocks as its finest granularity thus constraining the search space to circuits with good error propagation characteristics. This would simplify search time while fitness evaluation cost would also be reduced because of the properties these circuits would be guaranteed to have. It is unclear if such coarse level evolution would be able to generate as low overhead circuits as those found in this work and it would not be able to generate radically different strategies which might be more appropriate for certain circuits.

5.3.4 Summary: Evolution of Combinational Self-Checking Circuits

An automatic synthesis method for generating combinational circuits with TSC CED has been presented. Circuits generated with this technique required less overhead as a fraction of duplication than equivalents in all previous literature by a considerable margin. This is achieved because of the following properties. It has been shown to generate circuits not adopting the function-checker structural decomposition in which logic is reused for generating function outputs and controlling error propagation. It has also been shown to generate ad-hoc checking strategies for each circuit it encounters well-adapted to the particularities of the original circuit. Finally it has also been shown to be capable of adding TSC CED to an existing circuit without any modifications to the original circuit. The generated circuits do not suffer from incomplete checking of the checker and are guaranteed to fulfil the TSCG under the “all inputs between faults” assumption.

The design principle of error propagating blocks has been extracted from these designs and possibilities of how this could improve current self-checking synthesis techniques were discussed. This could lead to fast synthesis of low overhead self-checking circuits for large benchmarks or accelerate GA search.

5.4 Sequential

5.4.1 Synthesis Process

The synthesis process for sequential circuits with TSC CED is mostly the same as that for combinational circuits except for the alterations set out below.

Genetic Algorithm

Evaluation of a sequential circuit is noisy because output dependance on internal state may lead to a circuit producing a different output function depending on the ordering of the inputs. Each circuit evaluation is performed with a test pattern randomly generated as described below. In order to increase the accuracy of the fitness values, circuits were evaluated 20 times and each fitness metric was set to the average across all 20 trials.

Genotype to Circuit Mapping

During synthesis, sequential circuits are formed from four-input gates and edge triggered D-latches. The mapping from §5.3.1 is modified as follows. After the $q \times b$ bits defining the gates
driving primary outputs, \(2^{b-1} - i\) genes of \(16 + 5 \times b\) bits are defined. 16 bits define the gate function, 4\(b\) define the gate inputs and the remaining \(b\) bits specify the routing address for the data input to a latch accompanying the defined four-input gate. Latch clock input is always connected to the clock primary input. The last \(2^{b-1}\) bits in each address determine as before the gene defining the logic element to connect to, and the extra bit at the start now specifies whether a connection is made to the four-input gate or the latch. Circuits with recurrence are always allowed.

**Circuit Simulation**

Inputs are applied for 30 time steps. The clock signal fed to the latches is high during the first 15 and low during the last 15. Function and error output is sampled from the last 10 time steps. Gate delays are randomised equiprobably at either 1 or 2 time steps.

**Output Function Evaluation**

Random test patterns exercising all FSM transitions are generated as follows: a directed graph is built from the Moore FSM of the benchmark circuit such that some nodes are assigned as reset states and enough edges are marked as permanent so that all states can be reached by moving along them. We first choose a reset state to start from and add the input pattern necessary to bring the FSM to this state from any other onto the generated test pattern. A random walk is now begun such that walking along an edge appends its input vector to the generated pattern and removes it from the graph unless it is marked permanent. The walk ends when all edges have been walked along thus ensuring that the random test pattern generated will test all FSM state transitions. This is a novel technique which provides a more accurate evaluation with a shorter test pattern than the traditional approach of generating fully random long patterns.

**Evaluation of Self-Checking Ability**

\(f_{ST}\) and \(f_{FS}\) are calculated as above such that \(u_f\) is the number of faults for which an error is never signalled during normal operation, and \(u_i\) is the number of fault-state-input instances for which output is incorrect and an error is not signalled. An extra fitness metric \(f_t\) is used to measure the amount of erroneous unsignalled state transitions:

\[
f_t = \frac{1}{1 + k_t u_t}
\]

where \(u_t\) is the number of fault-state-input instances where output is correct, there is no error signal, but the transition to the next state is incorrect. \(k_t\) was chosen to be 50.

A circuit with \(f_t = f_{ST} = f_{FS} = f_i = 1\) will have TSC CED by Theorem 1 completed with the fact that all incorrect state transitions produce an accompanying error signal. This latter condition makes the circuit truly FS since all incorrect output due to errors in the function generator logic and the transition logic will be signalled.

The evaluation procedure to measure \(u_f\) and \(u_i\) is as follows. To test circuits at particular states they are run fault-free with the randomly generated test pattern and a circuit state snapshot is saved as they enter each state for the first time. At each state, for each fault, all inputs are applied and undetected failure instances are counted. When testing a new fault at a given state, the circuit state snapshot is restored before the fault is inserted. When all faults have been tested the snapshot is restored and the test pattern execution is resumed until the next untested state. The ST property for gate input faults is measure as with combinational circuits \(\S 5.3.1\) except that latch input faults do not require testing because they always manifest themselves as latch output faults.

\(u_t\) is measured as follows. Every time a state-fault-input combination neither affects an output nor signals an error, the circuit is clocked into the next state in order to evaluate whether the state transition is performed correctly. The response of the circuit under all inputs in the new state is compared to the stored fault-free response under the desired state. This requires responses to be unique for each state which is the case for all benchmarks tackled in this work. If the response is different and an error signal is not raised, then \(u_t\) is incremented.

**Combined Fitness and Parsimony**

Circuit size, as before, measured by fitness metric \(f_p\). The fitness vector used for sequential circuit synthesis is \((f_f, f_{ST}, f_{FS}, f_t, f_p)\).
Table 5.4: List of sequential benchmarks with their inputs, outputs, states, literals in factored form and circuit size after synthesis and optimisation with Sis into four input LUT technology §A.3. Also listed is a measure of the computational effort required for each circuit evaluation.

<table>
<thead>
<tr>
<th>Name</th>
<th>I</th>
<th>Q</th>
<th>States</th>
<th>Lits.</th>
<th>LUTs</th>
<th>Effort (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dk27</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>36</td>
<td>8</td>
<td>22.4</td>
</tr>
<tr>
<td>mc</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>42</td>
<td>10</td>
<td>73.9</td>
</tr>
<tr>
<td>beecount</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>51</td>
<td>11</td>
<td>192.7</td>
</tr>
</tbody>
</table>

Table 5.5: Overhead comparison of unconstrained evolved, locked evolved and previous best (PB) in literature for sequential benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Overhead</th>
<th>Oh./Dupl. Oh.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E_U$</td>
<td>$E_L$</td>
</tr>
<tr>
<td>dk27</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>mc</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>beecount</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Avg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4.2 Results

Overhead comparison and benchmarks

Benchmarks used for comparison are extracted from the MCNC’91 test suite. Four-input gate – or lookup table – technology is used for these experiments. Overhead will again be compared in terms of percentage of duplication required. The TSC voter tree will be assumed possible to implement using half the number of gates as with two-input gate technology. The number of simulator gate refreshes required to evaluate a circuit with $i$ inputs, size $s$, $w$ time steps per input application, $e$ noisy evaluations, and $t$ state transitions with a randomly generated test pattern of length $p$ is $ews(2s+1)(p+2t)$. The circuits tackled are listed in Table 5.4.

Evolved Sequential TSC Circuits

The maximum amount of processing power required was one week of around 100 2.8Ghz computers to evolve the beecount benchmark. All runs successfully arrived at TSC solutions with $f_f = f_{ST} = f_{FS} = f_t = 1$. Their overheads are listed in Table 5.5. Unconstrained evolved circuits use on average just below a third of the overhead required by duplication. The circuits evolved with the output function generator locked use on average a half of the overhead used by duplication. These overhead figures are again unheard of compared to the average requirement of 87% duplication overhead for the previous best found in the literature. In fact, Piestrak (2002) shows that these previous best would not even meet the TSCG because the checker would not be totally exercised and 94.4% of duplication overhead would be required for mc instead of 86%. Again the evolved circuits provide full fault detection with radically lower overhead. All evolved TSC circuits duplicate every latch once. It seems like they must keep track of the current state independently to have any chance of performing full diagnosis. Analysing, visualising and understanding evolved circuits formed with four-input LUTs is difficult but will be attempted for one circuit.

Circuit D - unconstrained mc

Circuit D performs the mc sequential benchmark and is TSC. It requires 15 LUTs and latches as opposed to 32 for the duplication and comparison approach. Fault latency is less than a clock cycle. Circuit D (Table 5.6) has a complex interconnectivity of four-input LUTs and latches. Output function generator units are influenced by all other logic units so it is not constrained to the traditional function logic-fetcher structural decomposition. There are some duplicated LUTs ($U_7 = U_{10}$) and latches ($U_5 = U_{12}$). Output $Y_1 = U_1$ is fed by $U_{11}$ which in turn depends on outputs $Y_2 = U_2$ and $Y_4 = U_3$ suggesting an output cascading approach. This output $Y_1 = U_1$ is
Table 5.6: Circuit D is a TSC sequential mc benchmark using four-input LUTs and latches. C is clock input. Circuit outputs \( Y_0 \ldots Y_4 \) are read from \( U_0, U_1, U_2, U_3 \) and \( U_4 \) respectively.

<table>
<thead>
<tr>
<th>Unit</th>
<th>LUT function / Latch</th>
<th>Unit inputs (abed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U_0 )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( X_2, U_3, U_10, U_13 )</td>
</tr>
<tr>
<td>( U_1 )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( U_8, U_{10}, U_{11}, U_{13} )</td>
</tr>
<tr>
<td>( U_2 )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( U_8, U_9, U_{10} )</td>
</tr>
<tr>
<td>( U_3 )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( U_8, U_5, U_0, U_6 )</td>
</tr>
<tr>
<td>( U_4 )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( U_{13}, U_1, U_8, U_8 )</td>
</tr>
<tr>
<td>( U_5 )</td>
<td>( Edge Triggered D-Latch )</td>
<td>( C, U_6 )</td>
</tr>
<tr>
<td>( U_6 )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( X_2, U_5, U_7, X_1 )</td>
</tr>
<tr>
<td>( U_7 )</td>
<td>( abed + abed + abed + abed + abed + abed + abed \times abed )</td>
<td>( X_0, X_1, U_8, U_5 )</td>
</tr>
<tr>
<td>( U_8 )</td>
<td>( Edge Triggered D-Latch )</td>
<td>( C, U_9 )</td>
</tr>
<tr>
<td>( U_9 )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( X_2, U_8, U_5, U_0 )</td>
</tr>
<tr>
<td>( U_{10} )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( X_0, X_1, U_8, U_5 )</td>
</tr>
<tr>
<td>( U_{11} )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( U_3, U_2, U_3, U_{12} )</td>
</tr>
<tr>
<td>( U_{12} )</td>
<td>( Edge Triggered D-Latch )</td>
<td>( C, U_6 )</td>
</tr>
<tr>
<td>( U_{13} )</td>
<td>( Edge Triggered D-Latch )</td>
<td>( C, U_{14} )</td>
</tr>
<tr>
<td>( U_{14} )</td>
<td>( abed \times abed + abed + abed + abed \times abed )</td>
<td>( X_2, U_8, U_5, U_4 )</td>
</tr>
</tbody>
</table>

then fed into the error output \( E = U_4 \) together with latches \( U_8 \) and \( U_{13} \) whose sources \( U_9 \) and \( U_{14} \) are similar. Even though at first sight its operation seems to be using output cascading together with some recalculation as some combinational circuits above, a more thorough investigation by observing dynamic circuit operation and analyzing LUT functions would provide more insight possibly uncovering novel useful principles of operation.

5.5 Absolutely Fault Secure Circuits

If the *all inputs between faults* assumption is dropped, TSC circuits fail to meet the TSCG. An alternative are Path Fault Secure (Smith and Metze, 1978) circuits which in turn assume that a circuit will be stimulated by a subset of inputs between fault arrivals and no other inputs will ever be applied to the circuit. There may be cases where neither of these assumptions holds and a highly dependable circuit with CED meeting the TSCG is desired.

*Definition 5*: A circuit \( G \) is Absolutely Fault Secure (AFS) if, for every sequence of faults in \( F \) arriving at any time during operation, the first incorrect output is preceded or accompanied by an error signal.

If a circuit is AFS it will always meet the TSCG by their definitions. No assumption is made about the timing of fault arrivals, they may all arrive at completely different times, during the application of the same input, or simultaneously. The AFS property defines the most stringent class of circuits with CED.

A circuit with a two-rail error signal could get both rails stuck at different values simultaneously, thus disabling all future signalling of errors and could never be AFS. One possibility for a robust error signal is oscillation. During this section all error signals \( z \) will signal healthy operation with oscillation and will signal an error with non-oscillation. This allows all stuck-at faults in the oscillation mechanism to be automatically detectable.

During this section the discussion will be limited to gate output faults because it is not possible to generate AFS circuits with respect to gate input faults using static data encoding: all gate inputs reading primary inputs may get stuck at correct values. An alternative encoding is conceivable in which different oscillation modes represent different data values for which AFS circuits could be generated to be AFS with respect to the full fault set.

A trivial example of an AFS circuit is depicted in Fig. 5.6a. Oscillation stops if \( x \neq y \), no
Chapter 5. Evolution of Digital Self-Checking Circuits

75

Figure 5.6: Examples of trivial Absolutely Fault Secure circuits.

Figure 5.7: A network of gates connected in a ring will oscillate when the next recurrent function output is always opposite to its current output.

matter what other faults are present. In Fig. 5.6b, oscillation stops if \( y \neq x_0 \oplus x_1 \) no matter what other faults are present.

A network of gates forming a loop as in Fig. 5.7 will oscillate if the loop function returns the inverse of the recurrent loop input. For gates with functions \( f_i \) and inputs \( x_{i0..r} \) the network will oscillate if and only if equation 5.2 holds:

\[
f_0(x_{00..r-1}, f_1(x_{10..r-1}, \ldots f_k(x_{k0..r-1} x_k), \ldots)) = \overline{x_k}
\]  

(5.2)

For example the loop in Fig.5.6b can be shown to oscillate: \( f_0 = \oplus, f_1 = \overline{\oplus}, f_2 = \oplus, x_{00} = a, x_{10} = b, x_{20} = y \) and using Eq. 5.2:

\[
f_0(af_1(yf_2(bx_{21}))) = a \oplus y \oplus (b \oplus x_{21}) = a \oplus y \oplus b \oplus x_{21} = \overline{x_{21}}
\]

5.5.1 Synthesis Process

This section will describe the alterations to the process described in §5.3.1 to synthesise AFS circuits.

Genotype to Circuit Mapping

Some circuits were synthesised using four-input gate technology where all \( 2^{16} \) gate types were allowed, as if they were look-up tables. For these runs a gene was formed of \( 16 + 4 \times b \) bits.

Evaluation of AFS Ability

Instead of observing the behaviour of circuit \( G \) with \( g \) gates under all \( g!2^g \) fault sequences, it is sufficient to observe behaviour under all \( 3^g \) fault combinations.

Theorem 4. If a combinational circuit \( G \) is FS with respect to all fault combinations in \( F \), then \( G \) is AFS with respect to \( F \).

Proof. If \( G \) is combinational then it will also be combinational under any combination of stuck-at faults. Then circuit \( G \) will behave in the same way after a set of faults \( f \) arrived separately or after they arrived simultaneously. Since \( G \) is FS with respect to any combination of faults then it will never produce incorrect output without an accompanying error signal in either of these cases.
Since $f$ is an arbitrary fault sequence with arbitrary timing between faults then this is true for all sequences of faults, and $G$ is AFS by Definition 5.

The evaluation procedure of AFS ability iterates over all $3^g$ fault combinations in ascending cardinality order. The FS property is verified as in §5.3.1 by counting the number of fault-set–input instances under which erroneous output is not signalled. Evaluation is optimised by not checking fault-sets which include other fault-sets already known to cause undetected errors. The supersets are assumed to also cause undetected errors since extra faults are unlikely to improve diagnosis capability and a safe pessimistic evaluation is arrived at. The resulting evaluation algorithm is as follows. $A$ is the set of undetected fault combinations, $F$ is the set of all single faults, $C(D)$ is the set of fault combinations tested in the current (previous) stage and $u_f$ is the number of fault-set–input instances producing un-signalled output errors. Comments will be provided after $//$.

1. $C \leftarrow \{\emptyset\}, A \leftarrow \emptyset$;
2. $D \leftarrow C, C \leftarrow \emptyset$;
2.1. $\forall d_i \in D$, $//$generate fault-sets to test next
2.2. $\forall f_j \in F$ such that $f_j$ is not at the same gate as any fault in $d_i$,
2.3. $d_{ij} \leftarrow d_i \cup \{f_j\}$ $//$create potential fault-set to test next
2.4. if$(\forall a_k \not\subset d_{ij} \forall a_k \in A)$ //if no undetected fault-set is included in $d_{ij}$
2.5. $C \leftarrow C \cup \{d_{ij}\}$ $//$add fault set to test next
2.6. else
2.7. $u_f \leftarrow u_f + 2^i$ $//$since some subset $a_k$ produced undetected errors, assume superset $d_{ij}$ also does
3. if$(C = \emptyset)$ HALT; $//$no more fault-sets to test
4. $\forall c_i \in C$, $//$test fault-sets in list
4.1. Evaluate circuit under faults $c_i$,
4.2. $t_f \leftarrow$ number of inputs producing erroneous output with no error signal;
4.3. if$(t_f > 0)$
4.4. $u_f \leftarrow u_f + t_f + 3|F| - |c_i|$
4.5. $A \leftarrow A \cup \{c_i\}$; $//$add fault combination to undetected list
5. Goto 2.

Note that fault sets $\{g_1.0, g_2.1, g_3.0\}$ and $\{g_3.0, g_1.0, g_2.1\}$ are identical and will not be added to set $C$ twice. This algorithm naturally terminates when all fault combinations have been visited resulting in $C = \emptyset$ in step 3. To limit processing cost it was run up to combinations of four faults. Full verification of the AFS property was carried out with no such limit.

In order to persuade the evolutionary algorithm to prioritise the detection of errors due to fewer faults over that of errors due to more faults a fitness metric was computed based on the number of undetected error instances for each fault combination cardinality:

$$f_{ui} = \frac{1}{1 + k_u}$$

where $u_i$ is the number of fault-set–input instances for which the fault-set had cardinality $i$ and incorrect output was produced with no error signal. $k_u$ was chosen to be 990. $u_i$ is the part of the $u_f$ in the algorithm above corresponding to fault sets of cardinality $i$. These metrics were then placed in ascending order in the fitness vector. During verification of the AFS property the above algorithm was run with no limit and $u_f$ was checked to be equal to 0.

**Theorem 5.** If $u_f = 0$ after a circuit $G$ with $g$ gates has been evaluated using the above algorithm with fault set $F$, then $G$ is AFS with respect to $F$.

**Proof.** If $u_f = 0$ then $t_f$ was never greater than 0, so nothing was ever added to $A$. If $A$ was always empty then no fault-set was discarded from the next set at 2.4. The algorithm generates fault-sets of the next cardinality by combining every previous fault-set with every single fault that does not result in a position clash – i.e. a gate being stuck-at two values. Some of the resulting sets are identical and all possible ones for each cardinality are generated. The algorithm will only stop when $C = \emptyset$ at 3 which will only happen when no single fault could be added to any lower cardinality fault set without a position clash. This will only happen when the lower cardinality
fault sets contained faults at all $g$ gates. Since the algorithm evaluates operation under all fault-
sets of every cardinality ranging from 1 to $g$ then behaviour under all possible fault combinations
is observed. Since $u_f = 0$ then there was no fault-set–input instance producing an undetected error
and $G$ is FS with respect to all combinations of faults in $F$. Thus $G$ is AFS by Theorem 4.

5.5.2 Results

The synthesis process did not always generate full fitness solutions. This is probably due to the
difficulty of the problem however evolvability might be improved by modifications in the algo-

Circuit E

Circuit E was synthesised using the process described above §5.5.1 using four-input gates and
starting from a fully random population. The output function is that of a full adder with two
outputs $s = x_0 \oplus x_1 \oplus c_{in}$ and $c_{out} = x_0 x_1 + x_0 c_{in} + x_1 c_{in}$. Circuit E has $u_f = 0$ when evaluated as
in §5.5.1 with no cardinality limit and is thus AFS by Theorem 4. Proof that the circuit performs
correctly under fault-free behaviour is laid out below. Two gates are used – and are the minimum
required – for the adder task and two more are added to perform checking. This is even less
overhead than required for a duplication approach which would require 3 extra LUTs and would
only be TSC and not AFS.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Function in disjunctive normal form (DNF)</th>
<th>Routing (a,b,c,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_0$</td>
<td>$a_0 + c_{in} + b_0 + d_0 + e_0 + f_0$</td>
<td>$x_1, c_{in}, g_1, x_0$</td>
</tr>
<tr>
<td>$g_1$</td>
<td>$a_1 + c_{in} + b_1 + d_1 + e_1 + f_1$</td>
<td>$x_0, x_1, c_{in}, x_0$</td>
</tr>
<tr>
<td>$g_2$</td>
<td>$a_2 + c_{in} + b_2 + d_2 + e_2 + f_2$</td>
<td>$g_1, g_3, g_0, x_0$</td>
</tr>
<tr>
<td>$g_3$</td>
<td>$a_3 + c_{in} + b_3 + d_3 + e_3 + f_3$</td>
<td>$x_1, g_1, c_{in}, g_2$</td>
</tr>
</tbody>
</table>

Table 5.7: Description of Circuit E. Routing is visualised in Fig.5.8.
The fault-free function (Table 5.7) was calculated by setting those rows in the truth table which would never happen – due to the dependencies between gates and primary inputs – to don’t care and then simplifying the function. $g_0$ is shown to generate the inverse of $s$ which is correct by the earlier definition in equation 5.1, $g_1$ generates $c_{out}$ and $g_2 = z$ is shown to oscillate using equation 5.2:

$$f_0(g_0, x_0, f_1(x_1, c_{in}, x_1)) = g_0 \oplus x_0 \oplus x_1 \oplus c_{in} \oplus x_1 = x_0 \oplus x_1 \oplus c_{in} \oplus x_0 \oplus x_1 \oplus x_1 = 1 \oplus x_1 = x_1^r$$

It is clear that oscillation will stop for fault sets containing faults at $g_2$ or $g_3$. Proving this for the remaining 8 fault combinations involving $g_0$ and $g_1$ would involve replacing them in the formulas with the stuck-at value of the fault and then verifying if the oscillation property still holds. This will not be attempted in this work and the reader will have to rely on Theorem 5 for proof of the AFS property.

At first sight of Circuit E, it can be seen that output $c_{out}$ cascades into $s$ which will propagate errors in most cases. $g_2$ and $g_3$ seem to do some duplication and comparison while keeping oscillation going. Further analysis of Circuit E is required to fully comprehend its operation and extract any design principles.

The fault model chosen of all gate output faults may be inadequate for gates of this size and further work could concentrate on gates of smaller sizes. It is still an open question whether AFS exist for any output function and any gate technology.

Circuit E is, to the author’s knowledge, the first automatically synthesised circuit to be AFS, ie. to meet the TSCG under no assumptions of fault arrival timing.

**Circuit F**

This circuit, shown in Fig. 5.9 was synthesised using two-input gate technology and the output function is also a full adder. The function can be, and is, implemented with a minimum of 5 gates. Circuit F adds 8 gates of checking overhead, less than that required for duplication. Circuit G does not have $u_f = 0$ when evaluated and is thus not AFS. However due to the incremental optimisation nature of the synthesis process, it is very close to being so. Table 5.8 shows the amount of fault-set–input combinations producing undetected errors for each fault-set cardinality $i$. For each cardinality $i$ there are $C_1^3 \times 2^i$ combinations of faults and 8 different inputs.

Table 5.8 shows that upon immediate arrival of two random faults at a random input, there is a 2.4% chance of an unsignalled error being produced. The probability increases with the fault-set size but is always very low.

Observation of Fig. 5.9 reveals that $z$ oscillates when $g_4 = g_5 = 0$ which is the case during normal operation. Any single fault causing an error at a function output will make $g_4 = g_5 = 1$ stopping oscillation. In fact circuit F has TSC CED with respect to single faults since the circuit is irredundant and ST. However the strength of circuit G is its robustness in the presence of multiple
faults. Double faults, one causing an error at an output and the other in the checker, are mostly detected because of the duplication in the checker so one of $g_4$ and $g_5$ will go high stopping oscillation. Only 2.4% of double fault–input combinations generate output errors without stopping oscillation. This duplicated checker structure is also enough to make this percentage small for fault-sets of up to 5 simultaneous faults. The probability of meeting the TSCG for this evolved circuit is a lot higher than that for comparable circuits in Mitra and McCluskey (2000).

A more accurate judgement of the value of this circuit in the field would be to analyse the probability of an output error occurring before an error signal. In the case of AFS circuits this would be 0, however it is not clear whether AFS circuits exist for every output function and gate technology. Calculating this reliability figure would involved plugging in the above probabilities into a fault model. This reliability would be improved if the circuit signalled errors more often, ie. where more ST. Thus the ST metric could be introduced into the fitness evaluation in §5.5.1 in order to reduce the chance of erroneous output occurring before an error signal.

### 5.6 Conclusions

A new synthesis method to generate combinational and sequential TSC circuits using a Genetic Algorithm has been introduced. The method was used to generate TSC circuits for several MCNC’91 benchmarks of up to 741 literals in factored form before optimisation. Evolved combinational and sequential circuits required on average 22% and 31% of duplication overhead respectively to perform full TSC CED. Such low overhead figures were previously unheard of in the literature and can be explained by the following. The proposed method is the first to automatically synthesis circuits with TSC CED with no structural separation between functional logic and a checker. It is also the first to automatically synthesis circuits adopt self-checking strategies which are well suited to the particularities of each circuit. It is also the first synthesis method to generate TSC circuits using multiple strategies to achieve SC.

The synthesis method was also shown to be able to add logic around an existing fixed output function generating circuit to produce a circuit with TSC CED. The overhead required is 32% and 50% of that used by duplication for combinational and sequential benchmarks respectively. Thus the method is capable of adding TSC CED to any irredundant circuit without modifying any of its structural or timing characteristics using less overhead than existing methods which modify circuit structure and do not provide full error detection. The value of the evolutionary unconstrained approach has been demonstrated. It is an open question whether it would be practical in industry. For a large production run producing a small circuit, one week of unattended computing power may be a small price to pay for the considerable reduction in overhead.

The design principle of error propagating blocks was extracted from analysing evolved designs. A finer level of hardware resource optimisation may be possible when dropping the constraints of producing and checking an encoding and instead focusing on ensuring all errors are propagated to one error signal. Error propagating blocks are building blocks to construct circuits under such constraints. Such knowledge may lead to a faster method based on conventional synthesis technology to produce TSC circuits with overhead comparable to the evolved ones.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$u_i$</th>
<th>Total instances</th>
<th>Undetected %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$C^1_1 \times 2^1 \times 8$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>$C^1_2 \times 2^2 \times 8$</td>
<td>2.4</td>
</tr>
<tr>
<td>3</td>
<td>924</td>
<td>$C^1_3 \times 2^3 \times 8$</td>
<td>5.0</td>
</tr>
<tr>
<td>4</td>
<td>6064</td>
<td>$C^1_4 \times 2^4 \times 8$</td>
<td>6.6</td>
</tr>
<tr>
<td>5</td>
<td>23684</td>
<td>$C^1_5 \times 2^5 \times 8$</td>
<td>7.2</td>
</tr>
</tbody>
</table>

Table 5.8: Fault-set–input combinations producing undetected errors in circuit F.
Absolutely Fault Secure circuits were introduced as the class of circuits which always meet the TSCG without any assumptions on fault and input timing. Some hand designed examples were presented and an evolved AFS full-adder using an oscillating error signal and made of four-input LUTs. Another adder using two-input gates is not AFS but detects almost all errors caused by multiple-faults. It has been shown how the evolutionary approach can generate circuits with various degrees of protection.

Future work includes the application of this technique to ALU components such as adders and multipliers possibly by evolving at the module and system scales simultaneously. Larger benchmarks should also be tackle to demonstrate the general practicality of the approach. Mathematical analysis of self-checking circuits could provide insight into the theoretical minimum hardware required and the existence of AFS circuits for different benchmarks and technologies. Synthesis of super-low overhead circuits without full error detection but maximising the probability of achieving the TSCG may also be interesting to industry.
Chapter 6
Evolution of Analog Self-Checking Circuits

Evolution has been shown to be very competent at synthesising digital circuits with CED comparing favourably against previous methods. However no such previous method exists to synthesise generic analog circuits with CED. This chapter presents evolutionary design as the first such proposed method and will discuss why it has so much potential to increase the reliability of analog hardware.

6.1 Introduction
Testing of analog integrated circuits (ICs) takes up a significant fraction of the design cycle and achieving a high quality test requires expensive high resolution equipment. Also as IC density increases and mixed signal systems are embedded in larger wafers, their controllability and accessibility becomes increasingly difficult and test signals less reliable. Analog built-in self-test (BIST) renders testing cheaper and faster by placing the test logic on the same silicon as the design under test (DUT). BIST can be used to test for permanent faults produced by physical defects during production and for on-line testing in the field. Errors in the field may be hard errors caused by structural permanent faults and parametric deviations of circuit properties or soft errors which are transient voltage spikes caused by ionised particles travelling through the silicon. Concurrent error detection (CED) checks for errors while the circuit is operational in the field and is required to detect soft errors. As mixed signal systems are used increasingly ubiquitously and in more safety-critical applications such as sensors for transport or medicine and mission-critical applications such as space probes, and as shrinking feature size results in an increasing soft error rate, there is a growing need for CED in analog circuits. CED with full coverage could also drastically speed-up manufacturing test. CED can be provided by duplication and comparison with high overhead which is too costly for most applications. Several methods and synthesis tools exist for adding CED to random digital logic with below duplication overhead. Most of these work by encoding circuit outputs and then checking if the code is broken. Significant amounts of research has gone into analog self-checking (SC) but attaining a comparable maturity as in the digital domain has remained elusive.

Vinnakota and Harjani (1994) propose an informal definition of analog SC circuits and show how these could be designed using fully differential circuits (FDC) with a differential analog code checker. Resulting designs provide CED but require significant designer effort and only detect transient faults. Chatterjee et al. (1996) propose a DC analog BIST using real number checksum codes for detecting errors in matrix vector operations, somewhat like the encode and check approach popular in digital SC. Their method interrupts circuit operation to feed in test vectors, requires effort from a designer to solve complex equations for each circuit, and an external AC tester to reach full fault coverage. This technique is extended to multiple faults by Zhou et al.
Chapter 6. Evolution of Analog Self-Checking Circuits

(1998). Vinnakota et al. (1995) describe a method for reducing the need for precision in the analog automated test equipment (ATE) used to test FDC by using analog circuit observer blocks (ACOB) by exploiting the redundancy in the differential analog code at the complement pair. ACOBs include thresholded checkers to detect when the code is broken due to a fault. Testing is offline, an external ATE is still required for test pattern generation (TPG) and the designer must adapt the technique to each circuit by hand.

Arabi and Kaminska (1996, 1997) introduce the oscillation test strategy for analog circuits. The circuit is altered so that it oscillates during test mode which is induced by turning on switches at key places in the design. Faults are detected by noting differences in the oscillation frequency. This eliminates the need for TPG, greatly simplifies the test response analysis (TRA) by allowing it to be fully digital and also reduces test time significantly. They report full coverage for an example circuit. Testing interrupts circuit operation and is sensitive to temperature variations while considerable effort by a designer is required to apply this technique to each circuit. J.W.Lin et al. (2001) develop the oscillation test further by eliminating the need for feedback reducing routing problems and overhead. Sophisticated analog designer effort is required for each circuit. Sharif et al. (1998) propose a system using partial duplication to increase fault-tolerance, and reconfiguration to increase fault detection by fully exercising the hardware. Their technique is demonstrated in an automotive application. Fault detection is on-line but not concurrent while significant designer effort is required to apply this technique to each circuit.

Lubaszewski et al. (2000) provide a formal definition for analog self-checking circuits: the Finitely Totally Self-Checking (FTSC) class. It also provides a methodology for designing such circuits using FDCs. Results do not provide full fault coverage CED, the method is only applicable to linear analog circuits, and significant manual analog designer effort is required. Venuto et al. (1999) apply a testing strategy for hard and soft faults in linear analog circuits to the testing of a chip used in CERN when subjected to high-dose irradiation. Testing is not concurrent while the described technique is ad-hoc for the chip. Cota et al. (2000) propose a digital signal processor (DSP) based test and diagnosis system for linear analog circuits which learns a reference circuit behaviour in one stage and compares this with actual circuit behaviour in a second testing stage. A third stage locates faults by comparing output to that of a mathematical model of the circuit subjected to injected faults. This method is generic for linear circuits but requires an external DSP and does not provide CED.

Carro et al. (2000) describe a method for testing circuits composed of biquad filters. Each biquad is adapted so it can reconfigure to perform multi-frequency TPG and TRA and test other biquads. The technique is made temperature independent by estimating behaviour variation at different temperatures. This technique works only for circuits composed entirely of biquad filters and requires the circuit to go off-line for testing. Venuto et al. (2002) describe a technique for testing analog circuits using window comparators built entirely of digital gates and thus with very low overhead. However this method still requires an ATE for TPG, some design effort to determine the comparator details and the circuit must go off-line during test. Temperature variations were found not to affect this technique whilst lot-to-lot variations did. Stratigopoulos and Makris (2003) propose a method for the design of linear analog circuits with CED using less than duplication overhead by generating an estimated output from some monitored internal signals. Designer effort must go into the selection of these signals and fault coverage figures are not provided. Negreiros et al. (2004) propose using a statistical sampler to provide on-line testing of radio frequency (RF) circuits. This technique is ad-hoc for this class of circuits and requires a digital TRA performing FFT.

Most techniques described above reduce the requirements on the ATE or provide on-line BIST which must interrupt circuit operation to test it in the field and can not detect soft errors. A few of the techniques do provide CED but without full fault coverage. All methods for designing analog SC circuits to date require considerable effort from the designer to be applied to each particular circuit and many are ad-hoc strategies applicable to only one class of circuits. No automatic
technique for the synthesis of analog circuits with CED has been reported in the literature. Also, no systematic method for adding CED to analog circuits has been proposed. Analog SC techniques suffer from the fact that the checker is subject to the same process variations and noise as the DUT and this must be taken into account to provide a robust error signal even when the checker is not within healthy operating parameters.

The lagging behind of analog SC compared to its digital counterpart could be attributed to the increased complexity of the problem due to the wider range of behaviours and large interdependency present in analog designs. Some analog circuits today are still designed by highly skilled people due to the difficulty of analog automatic synthesis in general.

Evolutionary techniques applied to analog circuit design have produced circuits operating in surprising or intricate ways (Thompson, 1998b; Tyrrell et al., 2003) and competitive with state of the art human designed equivalents (Koza et al., 2004; Lohn et al., 2003b). As the evolutionary process acts on the whole design at once it can deal effectively with the complexity and interdependency present in analog circuits. Evolutionary techniques have also been successfully applied to the design of digital circuits with CED (Garvie and Thompson, 2003b,a, 2004), §5.

6.2 Motivation

This section will postulate why the evolution of analog circuits with CED is such a promising area of research. Most importantly, there is as yet no automatic method for producing analog circuits with CED. Analog circuit design is a very complex process not unlike an art for the most sophisticated designs. There are few highly skilled analog engineers and they are very well paid whilst automatic synthesis tools are primitive. This could be attributed to the large amount of flexibility and complex interdependencies present in analog designs. Evolutionary techniques can deal effectively with such search spaces because they are “aware” of the whole circuit during the design process.

The evolution of digital circuits with CED has produced circuits with full fault coverage and lower overhead for all benchmarks attempted (Garvie and Thompson, 2004). These designs were found with no knowledge of previous SC techniques, all approaches to SC were found entirely by the evolutionary process. The approach would have worked just as well if the field had been just started, and this is not too far from the current state of the analog SC field. The evolution of analog CED is likely to be easier than for digital CED because of the increased evolvability when working with a smoother non-discrete fitness landscape. The increased flexibility in circuit response is a benefit to the evolutionary process. Hence evolution is likely to automatically find undiscovered SC strategies for analog circuits as it did for digital SC circuits but on a more evolvable landscape. Evolution is also capable of using resources extremely parsimoniously such as the discriminator in Thompson (1997) and §7. In Koza (1994) and in the digital self-checking circuits in Garvie and Thompson (2003a, 2004) it arrived at designs in which hardware resources were used both for the function and for CED simultaneously. In the digital domain, evolution was capable of designing an efficient checking strategy well adapted to the particularities of the circuit. It is very likely that the increased flexibility of analog circuits will allow an even greater adaptation to particularities of the circuit and technology to arrive at very low overhead analog circuits with CED.

Evolution of analog circuits can be made fast by using reconfigurable real hardware or distributing the simulation across a grid (Koza and Andre, 1995)§4. Furthermore, analog CED could provide a richer error signal perhaps indicating “how wrong” the function outputs are due to faults or parametric variations due to changes in operating conditions such as noise or temperature.

If this line of research is pursued, evolutionary synthesis could not only be the first automatic method for the design of circuits with CED, but would remain a highly suitable one.
6.3 Transfer and Noise

This section will discuss central issues when evolving hardware which, if not resolved, renders the whole approach practically useless.

Transfer refers to the capability of taking a circuit evolved in one medium such as an FPTA (Langeheine et al., 2000) and producing it in another such as ASIC. This is fundamental as industry cannot be expected to always incur the overhead and fragility of deploying final designs on reconfigurable devices which were suitable for evolution – although sometimes (e.g., small production runs) FPGAs have been deployed in industrial products. Several evolved circuits did not transfer such as Thompson’s discriminator evolved on an FPGA (Thompson and Layzell, 1999) whose operation was never fully understood, some of Koza’s early circuits evolved in simulation (Koza et al., 1997b) which would have drawn prohibitive amounts of current when built, and some of Layzell’s (Layzell, 2001) circuits evolved on the Evolvable Motherboard (EM) which never transferred fully to simulation. However there are examples of successful transfer such as Zebulum’s OpAmp (Thompson et al., 1999), the more conventional tone discriminator evolved on The Evolvatron (Thompson, 1998c) and Trefzer’s circuits evolved on an FPTA (Trefzer et al., 2004).

Robustness across varying process and operating conditions likely to be found in the field is also a requisite for evolved designs to be practical and is intimately related to transfer: it measures if a design transfers to another lot or to a different section of the same chip, or if a design transfers to other operating conditions such as different temperatures. The goal is for evolved designs to be robust tolerating process lot-to-lot variation and varying operating conditions in the field. Achieving this robustness will also help transfer. Thompson (1998c) defines the whole set of variation a design must be robust to as the operational envelope.

A final issue relevant to conventional circuits as much as evolved designs is robustness in the face of noise. Noise may be produced by the analog circuit itself, clock feed-through, switching, supply noise, electro-magnetic sources and irradiation. This is also related to the above since it deals with a circuit’s robustness to transient changes in the environment. The goal again is to be robust to likely noise sources, and this robustness would most likely aid achieving robustness to varying operating conditions and help transfer. For particular circuits to be deployed in specific environments there may be a well characterised noise source the circuit must be robust to.

Robustness to lot-to-lot process variations (Venuto et al., 2002), varying temperature (Carro et al., 2000) and noise (Vinnakota et al., 1995) is also an issue for conventionally designed circuits but is especially troublesome for evolved designs because the evolutionary design process is extremely good at adapting the design to the particularities of the medium. This can result in extremely efficient designs but may also lead to over-adaptation. For example Thompson’s discriminator only worked properly within a temperature range, that defined by the temperature of the lab where the evolvable chip was during the night and day. The same circuit was also found not to function when programmed on nominally identical FPGAs from different lots nor even on a different section of the same FPGA it was evolved on. Here evolution has over adapted to a certain temperature range and to the parasitic particularities of the pieces of silicon it was evolved on. The design never transferred properly to simulation or to a breadboard. Another example of over-adaptation in the face of noise is an oscillating circuit evolved in Layzell (2001) on the EM which oscillated by amplifying the signal from a local radio station. This oscillator is clearly not robust to a change in this noise source. Over-adaptation must be avoided if transfer to other mediums under varying operating conditions and noise is desired.

The most straightforward method for avoiding over-adaptation is to vary those factors evolution must not depend on throughout the evolutionary process. This is analogous to a Minimal Simulation (Jakobi, 1997) in which transfer from a simulated to a real robot is facilitated by introducing varying amounts of noise at sensors which would be too expensive to simulated accurately. Then the evolved robot controller does not depend on them and can function robustly no matter what these sensors measure in the real robot. Analogously, for rendering an evolved circuit robust to temperature it would have to be evolved under various temperatures. For robustness to partic-
ular parasitics it could be evolved so it gets programmed in different parts of the reconfigurable device, while lot-to-lot variation robustness could be achieved by evolving on several reconfigurable devices. This technique was first embodied in Thompson (1998c) with The Evolvatron, where a circuit was evaluated on several Xilinx 6200s from different lots and at different temperatures and its fitness was taken as the minimum across all of these. There may be alternative ways of introducing sufficient noise during evolutionary design using a single reconfigurable chip so that evolved circuits transfer and are robust in the field. Techniques such as these are necessary for the evolutionary approach to be feasible as an industrial process. Such methods will be presented for the FPTA and the EM in §7.5, and these examples should provide insight into how this could be achieved on other platforms. Introducing selection pressure for smaller designs would also facilitate transfer by getting rid of unnecessary complexity.

A more unwieldy way of achieving transfer is by simulating the particularities of the evolution medium on the target medium. This is clearly a time-consuming process as the parasitics may be elusive (Thompson and Layzell, 1999). However a balance between this and the noise variation technique may be practical. Achieving transfer by this means may provide insight into what are the main difficulties and how they could be addressed automatically. A study of the successful transfer of a tone-discriminator using only two bipolar transistors evolved on the EM onto a breadboard by simulating the EM parasitics on the breadboard is provided in §7.

It must be noted however that some over-adapted designs may reach levels of efficiency beyond those of robust designs specifically because they exploit more of the physics around them. In some cases the benefits of such circuits may outweigh the cost of maintaining them within certain operating conditions, eg. the brain is kept within a narrow temperature range at great cost.

6.4 Method

This section will describe how the evolution of analog self-checking circuits could be achieved. Unfortunately not all of these ideas were implemented due to time constraints and the author would like to invite any reader to do so with the prospect of plenty of fruit-picking in this fertile unexplored line of research. Actual implementation and ideas will be distinguished with “was done” and “could be done”.

6.4.1 Evolutionary Medium

Analog circuits can be evolved extrinsically – in simulation – as in Koza et al. (1997a) allowing unconstrained topologies but requiring prohibitive amounts of processing time for accurate simulation of the hardware. Also, evolved circuits would not be able to use any of the physics omitted from the simulation. By evolving on real hardware it is even possible for evolved designs to make use of laws undiscovered by humans, plus also those considered too subtle or too expensive to simulate.

Intrinsic hardware evolution – on real hardware – can be performed on any physical medium capable of instantiating several configurations. Intrinsic evolution was first performed on an FPGA (Thompson, 1998b) exploiting analog properties of a device designed to be used digitally. This research spawned efforts to produce reconﬁgurable devices aimed solely to hardware evolution. Layzell (2001) developed the EM, an array of reconﬁgurable analog switches allowing all possible connectivity between daughter boards holding any kind of analog component. Because of the low density of its printed circuit implementation and open architecture it is highly observable allowing deep analysis of evolved circuits. However it could only ﬁt six transistors comfortably and is too small for the evolution of complex analog circuits such as OpAmps. The evolvable hardware group at JPL, NASA, designed their FPTA (Stoica et al., 2001) and FPTA2 (Keymeulen et al., 2004). Both of these are composed of blocks of CMOS transistors with reconﬁgurable switches placed so as to facilitate the evolution of typical analog components such as OpAmps. Langeheine et al. (2000) from the Electronic Visions group at Heidelberg University developed another FPTA
independently in which a 16x16 checkerboard array of 256 NMOS and PMOS transistors with configurable widths and lengths are available with analog multiplexers controlling the routing.

Langeheine’s FPTA was chosen due to its flexibility and size. It is large enough to instantiate analog circuits of considerable complexity and provides an unbiased fully reconfigurable structure that should be highly exploitable by evolution. However the method to evolve SC analog circuits is generic to any platform. Each transistor cell in the FPTA has connection nodes to its four (NEWS) neighbours. Each transistor terminal (GDS) can be connected to one of the interconnect nodes, to GND, V++ or left floating. The transistor within each cell can be configured to have one of 15 widths and 5 lengths.

6.4.2 Genetic Algorithm

Most of the evolutionary algorithm implementation details were adopted from Langeheine et al. (2002, 2004) and will now be outlined. A binary genotype is used to program the SRAM configuration cells on the FPTA directly. Evaluation of a population is speeded up by hardware implementation of a significant portion of the evaluation procedure on an intermediate card. Population size is 30 and rank selection is used with one elite. Crossover and mutation operate at the phenotypic level. Crossover swaps rectangular areas of configuration between two individuals. Crossover parents were selected from the top 90% of the population with the rest truncated. Background mutation can affect transistor length and width or routing. Transistor sizing mutation rate was set to 0.002 while routing mutation rate was 0.001 per genotype allele. Fitness varied from 0 (best) upwards with no limits.

This setup was augmented by the author by adding a Simulated Annealing like variation in the mutation rates, so that they decreased as the average fitness decreased (improved). Mutation rate for routing was set at 0.08 for perfect fitness individuals with a 3.5 factor multiplied per additional decade of fitness. So \( m = 0.08 \times 3.5 \times \log_{10}(10^{1/3.5} - 0.08 \times f_b) \) where \( f_b \) is the best (smallest value) fitness in the current population. Work by Jörg Langeheine and Martin Trefzer was also used which implemented the capacity to reset the FPTA to a random configuration between fitness trials aiming to prevent circuits depending on charges existing on the silicon before a fitness evaluation took place.

6.4.3 Evolution of Filters

In order to evolve an analog SC circuit it is first necessary to evolve a functional circuit – ie. to have something to check. This section will describe how this was and could be achieved for analog filters. Task evaluation procedures described were either developed entirely by Jörg Langeheine and Martin Trefzer or in cooperation with the author. All experiments described below were carried out by the author.

Analog filters allow only certain “pass” frequencies through them. They are diverse in specification and complexity and are widely used.

A low-pass filter is the simplest type of filter. It ideally allows all frequencies below the cut-off frequency and blocks the rest. An ideal high-pass filter only allows frequencies above the cut-off frequency while a band-pass filter allows a range of frequencies through. The order of a filter is related to the sharpness of the gradient of the attenuation around the cut-off frequencies.

When evaluating an FPTA configuration’s fitness as a filter, its frequency response is obtained by evaluating the configuration’s time response to a step function and performing a fast Fourier transform (FFT). Fitness is calculated using the squared errors fitness function defined in equation 6.1:

\[
    f = \sum_{s=0}^{S} (R(s) - R_{tar}(s))^2
\]  

(6.1)

\( S \) is the number of samples, \( R(s) \) is a particular observed sample and \( R_{tar}(s) \) is the target value for that sample. What an observed sample is depends on the task and can be an output voltage in
Chapter 6. Evolution of Analog Self-Checking Circuits  87

Table 6.1: Attributes of various Op-Amps.

<table>
<thead>
<tr>
<th></th>
<th>Ideal</th>
<th>Desired</th>
<th>CMOS</th>
<th>741</th>
<th>1st attempt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain (dB)</td>
<td>∞</td>
<td>&gt; 60</td>
<td>&gt; 70</td>
<td>&gt; 60</td>
<td>&gt; 40</td>
</tr>
<tr>
<td>Offset (mV)</td>
<td>0</td>
<td>&lt; 20</td>
<td>&lt; 10</td>
<td>&lt; 1</td>
<td>&lt; 50</td>
</tr>
<tr>
<td>Slew rate (V/µs)</td>
<td>∞</td>
<td>0.5</td>
<td>5</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>Settling time (µs)</td>
<td>0</td>
<td>&lt; 10</td>
<td>&lt; 10</td>
<td>&lt; 10</td>
<td>100</td>
</tr>
<tr>
<td>Common mode rejection ratio (dB)</td>
<td>0</td>
<td>60</td>
<td>&gt; 60</td>
<td>90</td>
<td>40</td>
</tr>
</tbody>
</table>

response to an input voltage, or an attenuation value in response to a frequency. The squared errors fitness function must be minimised in order to find better solutions. Error is the difference between the actual attenuation and the desired attenuation and is summed across all frequencies. Desired attenuation is $0\,\text{dB}$ for pass frequencies and $-40\,\text{dB}$ for filtered frequencies. A don’t care frequency region was also defined for some experiments which was ignored during fitness calculation. For wider don’t care regions, lower order filters will get higher fitness.

6.4.4 Evolution of Digital to Analog Converters
An $n$-bit digital to analog converter (DAC) will output the analog value represented at its input. Commonly the input is binary encoded such that $5\,\text{V} = 1$ and $0\,\text{V} = 0$ while the output voltage is equal to $5\,\text{V} \times \left( \frac{x}{2^n - 1} \right)$ where $x$ is the value encoded at the input ranging from 0 to $2^n - 1$. DACs are easier to construct in CMOS than their inverse function analog to digital converters (ADCs).

Evolving DACs were evaluated by applying the $2^n$ input vectors at the input in a random order, each vector applied for roughly $1\,\mu\text{s}$. The squared errors fitness function was used measuring the difference between actual output and that of an ideal DAC.

6.4.5 Evolution of Operational Amplifiers
This section will describe the first steps of a possible method to evolve OpAmps on an FPTA. Industry competitive OpAmps were evolved in simulation by Zebulum et al. (1998).

An OpAmp has two inputs (+ and -) and one output with high voltage gain. There are many complex characteristics defining an OpAmp. Some of the more basic ones will be outlined:

- DC Gain: or open loop DC gain is the amplification obtained at the output voltage level when compared to the voltage difference at the inputs, when the inputs are stable.
- Offset: the average difference between input voltages so that the output is at 0V.
- Slew rate: maximum rate at which the output can change.
- Settling time: maximum time for the output to reach a stable value after input has changed.
- Common mode rejection ratio: measure of how much the output changes when both inputs are changed together.
- Frequency and phase response: measure of response to AC signals.

Having certain values for the first four qualities is essential for a circuit to be called an OpAmp. Table 6.4.5 presents the ideal and desired values for some OpAmp attributes. It also presents values for a typical CMOS OpAmp and reasonable values to aim for on a first attempt at evolving one. The evaluation of the essential properties of an OpAmp was divided in two stages. The first stage measured the DC gain and offset while the second measured the slew rate and settling time.

Gain and offset were measured as follows. The $+$ input was fixed at seven voltages ranging from $1\,\text{V}$ to $4\,\text{V}$ in $0.5\,\text{V}$ intervals. While this was fixed the $-$ input was swept through the whole $0\,\text{V}$ to $5\,\text{V}$ range. The desired response for an OpAmp is to act as a comparator so that $V_Q > 4.5$
when $V_- < V_+$ and $V_Q < 0.5$ when $V_- > V_+$. Gain is measured as the gradient of the curve as it falls in the active region. The offset is the difference between the centre of this falling gradient on the $V_-$ coordinate and the value of $V_+$. During evaluation there were seven such curves, one for each value of $V_+$. From a circuit’s response to this stimulus it is also possible to estimate the common mode input range and rejection ratio.

Fitness was calculated using two different methods. The first one summed the squared errors over all values of $V_-$ for the seven curves, such that the ideal curves were step functions from 4 to 1 at $V_- = V_+$. In effect this penalised any deviation from outputting 4V when $V_- < V_+$ and 1V when $V_- > V_+$. Gain and offset requirements are implicitly captured since improving fitness will make the circuit produce a steeper gradient nearer to $V_+ = V_-$. One problem with this fitness calculation is the unnecessary selective pressure for circuits outputting exactly 4V and 1V instead of >4V and <1V outside the active region.

The second fitness calculation method generated five fitness metrics normalised from 0 (worst) to 1 (best) measuring gain and offset explicitly:

1. $f_{\text{gain}}$: Let $V_M$ ($V_m$) be the maximum (minimum) voltage at $V_Q$ for a $V_Q$ vs. $V_-$ curve at a particular value of $V_+$. Let $V_{TH} = 0.9 \times (V_M - V_m) + V_m$ and $V_{th} = 0.1 \times (V_M - V_m) + V_m$. Now let $V_h$ be the last $V_-$ value for which $V_Q > V_{TH}$, and $V_l$ be the first $V_-$ value for which $V_Q < V_{th}$. Let $L$ be the total number of $V_-$ samples. Then the maximum possible gain is when $V_l = V_h + 1$ and the minimum gain is when $V_l = L - 1$ and $V_h = 0$. A fitness function normalised to the [0,1] range is:

$$f_{\text{gain}} = \begin{cases} 0, & V_h > V_l \\ \frac{V_h - V_l}{L}, & \text{otherwise} \end{cases}$$

2. $f_{\text{offset}}$: Let transition voltage $V_T = \frac{V_l + V_h}{2}$ and offset voltage $V_o = |V + V_l|$. The minimum offset is 0 and the maximum is $L$. A normalised fitness function is:

$$f_{\text{offset}} = \begin{cases} 0, & V_h > V_l \\ \frac{V_h}{L}, & \text{otherwise} \end{cases}$$

3. $f_{\text{envabove}}$: This fitness function penalises precocious excursions below 4V before the transition. The minimum is 0 while the maximum is the area under $V_Q = 4V$ for $L - 2$ values of $V_-$. 

$$f_{\text{envabove}} = \sum_{V_-=0}^{V_h} \max(4 - V_Q(V_-), 0)$$

4. $f_{\text{envbelow}}$: Is similar to the above for excursions above 1V after the transition.

5. 

$$f_{\text{prize}} = \begin{cases} 0, & V_h > V_l \\ 1, & \text{otherwise} \end{cases}$$

These fitness metrics are averaged over all seven curves corresponding to each value of $V_+$ to arrive at five final explicit fitness metrics for the first evaluation stage normalised from 0 to 1. In order to use these metrics together with the others they were scaled and inverted so that the fitnesses actually used $f_i = 1000(1 - f_i)$.

The second evaluation stage measured slew rate and settling time. This was achieved by connecting the circuit’s output to $V_-$ and stimulating $V_+$ with a step function increasing from 1.5V to 3.5V and with the corresponding step down. Analysis of the circuit’s response allows measurement of slew rate, settling time and a FFT could provide some measurement of its frequency and phase response. The squared errors fitness function was always used for the second stage. Error was the difference between the step response at $V_Q$ and the step function itself.
Incremental Breeding  The method used to evolve OpAmps was a combination of incremental evolution and breeding. This was required because none of the fitness functions mentioned above provided a complete specification of behaviour suitable to be used during all stages of evolution. The fitness function explicitly measuring gain and offset was most effective at optimising these qualities, but only when they were already present. When starting from a random population, the squared errors fitness function was considerably more efficient at finding circuits with gain. Similarly, measuring the slew rate and settling time was only relevant after the output followed the step function at the input.

This problem is pictured in Fig. 6.1. Let set $X$ contain all individuals with good fitness with respect to fitness function $f_X$. Let $f_C$ be a finer gauge of a circuit’s fitness with respect to the goal, but only relevant when the circuit achieves good fitness $f_B$. Thus $f_C$ is only defined within $B$. Then let $f_B$ only be relevant when an individual already has good $f_A$ fitness. Then $f_B$ is only defined within $A$ and $C \subset B \subset A$. To arrive at an individual with good $f_C$ the evolutionary process will first have to be guided to enter $A$, then $B$ and then $C$. This can be achieved by changing what fitness function enforces the strongest selection pressure at each stage. Another reason for doing this is that like gears, different fitness functions may be more effective at driving evolution through different areas of the landscape. This problem is similar to that encountered during the multi-objective optimisation of digital SC circuits Garvie and Thompson (2003b) where such strong dependencies exist between fitness metrics. In that work a dictionary sort approach effectively resolved the problem. This works by establishing a priority of fitness metrics (e.g. $f_A \gg f_B \gg f_C$) so that when sorting the population for rank selection, when two individuals are compared lower priority metrics are only checked if higher priority ones are all equal. This is effective in the digital domain because evolution was strongly driven to achieve full $f_A$ fitness (enter $A$) and then all pressure was on keeping full $f_A$ and improving $f_B$ (enter $B$) and so on. This is exactly the desired effect mentioned earlier. This dictionary sort approach could be adapted to the analog domain §6.4.8 and could replace the incremental breeding approach.

When evolving the OpAmps the change in fitness function was done manually. The runs were started with the squared errors fitness function for measuring gain and offset. Several populations were saved once the evolutionary process reached good fitness levels. The whole populations were saved, not just the best individual, because PFT experiments in Layzell (2001) suggested that evolution may be more efficient at continuing from a whole population containing ancestral genetic material. Several populations were saved in order to provide a diverse choice of behaviour for the next stage of breeding. Evolution was then continued from one of these good squared error fitness populations using the five explicit scaled normalised gain and offset fitness functions summed together until good fitness values were again achieved and several populations saved. The final stage of incremental breeding added the squared errors fitness function measuring slew rate and settling time to the explicit normalised gain and offset fitness function. Saving of good populations also allowed their restoration when the evolutionary process had “fallen off a ridge” and was struggling to slowly get back to similar fitness heights. The transfer from one fitness function to another was done when the evolutionary process had not been capable of further optimisation for a considerable amount of time.
6.4.6 Other tasks

Random Number Generator
Transistor level hardware evolution is a promising avenue for synthesising parsimonious analog RNGs. Evolution’s capacity to tap functionality out of physical resources may lead to a design using chaotic physical processes to generate truly random voltages with a handful of transistors. Transfer would be a major issue with such evolved designs. The fitness function could test compliance against a standard such as FIPS 140.

Voice Recognition
A setup similar to Thompson (1998a) could be used to discriminate between two words or a more complex setup could be used to build a system that could evolve in the field to distinguish between several names. This could be used for mobile devices where power consumption is to be minimised.

Image Recognition
A transistor array may also be able to be evolved to distinguish between several images at high speed with few resources.

6.4.7 Evolution of Analog Self-Checking

This section will describe a set of ideas of how analog self-checking circuits could be evolved. These are largely based on the method used to successfully evolve digital self-checking circuits Garvie and Thompson (2003b) §5. The error signal would be provided as two outputs $z_0, z_1$ such that the smaller the difference between them, the stronger the error signal.

Definition of the Goal
SC circuits commonly aim to reach the Totally Self-Checking (TSC) goal mentioned in §5.1: the first erroneous output of the functional circuit results in an error indicated in the checker output. As defined earlier in §5.2.1, TSC circuits are self-testing (ST) and fault-secure (FS) and meet the TSC goal under the assumption that all inputs are applied between fault arrivals. This assumption cannot be made for analog circuits where there are infinite possible input values. The finitely TSC (FTSC) definition Lubaszewski et al. (2000) is suitable for analog circuits:

Finitely self-testing: G is finitely self-testing for a fault set F if there is a subspace $A_f$ of the input code space A such that: $A_f$ has a finite number of elements, and for each fault in F there is at least one input value belonging to $A_f$ that produces an error signal. Finitely totally self-checking: G is finitely TSC for a fault set F if it is fault-secure and finitely self-testing for the fault set F.

A FTSC circuit will reach the TSC goal if all inputs in $A_f$ are applied during fault arrivals.

Fault Injection in CMOS
In order to evaluate a circuit’s effectiveness at detecting errors due to faults, the method adopted is to observe its behaviour in the presence of faults. Faults can be injected into an FPTA by altering its configuration.

Faults affecting CMOS logic can be categorised as catastrophic or parametric. Catastrophic or hard faults are more common and cause large disturbance to circuit behaviour. These may be open faults at the gate, source, drain or interconnect and completely interrupt current flow, or shorts between any of the three terminals, totalling six hard faults. Parametric or soft faults are variations of transistor and interconnect properties and are especially problematic for circuits with parameter matching such as FDCs.

For some circuits, certain faults may not be realistic or may be redundant. Reduction of the amount of faults to inject would reduce evaluation time. A simple method for redundant fault removal could remove all faults which do not change the circuit, eg. a drain-source short fault when the drain and source are already connected, a gate open fault when the gate is already floating, etc... A more sophisticated method could use the $L^2RFM$ technique for unrealistic and unlikely fault removal developed in Ohletz (1996).
Faults can be injected onto an FPTA cell by altering its configuration:

- **Open**: Connect the terminal to nothing. Interconnect opens could be simulated by removing connections between nodes.
- **Short**: If neither of the terminals is connected to Vdd, Gnd or nothing then a short can be injected simply by adding a connection between the nodes the terminals are connected to. For example if the gate is connected to N and the source to E then a gate-source short can be introduced by a NE connection. Unfortunately, if the gate were connected to N and the source to Vdd or Gnd the situation is more complex. Injecting a short would require connecting the gate to Vdd (or Gnd) but this would disconnect it from the N node resulting in an undesired open. One solution would be to allocate neighbouring spare transistors which were constantly switched on and provide a Vdd or Gnd signal through routing. This difficulty arises from the design goal of the FPTA to be safe under any configuration to avoid damage by shorting Vdd and Gnd.
- **Parametric**: These could be injected as variations in transistor length and width. Variation in interconnect resistance and capacitance could be introduced by adding redundant connections in parallel. The latter would also require spare resources to be allocated on the FPTA.

### Self-checking ability evaluation

The procedure to evaluate an analog circuit’s FST ability and fault-secureness (see definitions above) is adapted from that used in digital circuits §5.3.1 and extended to provide a finer measure of these abilities. Such a fine measure is possible due to the continuous nature of analog signals and makes for a smoother fitness landscape. A circuit $G$ with output function $y$, output function under fault $f$ $y_f$, error signal $z_0z_1$, fault set $F$, normal operation input space $A_f$ (these may be voltages or frequencies depending on $G$), error signal threshold $e_{th}$, output error threshold $q_{th}$ is evaluated using the following algorithm:

1. $u_f = s_e = u_i = s_q = 0$
2. Evaluate $G$ under all $a \in A_f$. If an error signal is produced (ie. $|z_0 - z_1| < e_{th}$) for some $a \in A_f$ then go to step 3, otherwise go to step 4.
3. Set $f_{STD} = f_{FSD} = 0$ and $f_{STC} = \text{MAX}_e - \sum_{a \in A_f} |z_0 - z_1|$ and then terminate.
4. $\forall f \in F$.
   (a) Set fault $f$.
   (b) Evaluate $G$ under all $a \in A_f$.
   (c) If there is no $a \in A_f$ which produces an error signal then increment the number of undetectable faults $u_f$.
   (d) Add integrated error signal $\sum_{a \in A_f} |z_0 - z_1|$ to the cumulative error signal $s_e$.
   (e) $\forall a \in A_f$.
      i. If output error exceeded the threshold $(|y_f(a) - y(a)| > q_{th})$ and no error signal was produced then increment number of undetected error instances $u_i$.
      ii. Add integrated undetected output error $\sum_{a \in A_f} \text{max}(0, |y_f(a) - y(a)| - |z_0 - z_1|)$ to cumulative undetected output error $s_q$.
   (f) Clear fault $f$. 
5. Calculate discrete normalised FST fitness \( f_{STD} = \frac{1}{1+k_{STD}} \)

6. Calculate continuous normalised ST fitness \( f_{STC} = \frac{g}{MAX} \)

7. Calculate discrete normalised FS fitness \( f_{FSD} = \frac{1}{1+k_{FSD}} \)

8. Calculate continuous normalised FS fitness \( f_{FSC} = \frac{MAX - s_q}{MAX} \)

where \( MAX_a (MAX_p) \) is the maximum possible integrated error signal (output error) over a circuit evaluated under all \( a \in A_f \).

This algorithm calculates four fitness metrics normalised in the 0 (worst) to 1 (best) range, \( f_{STD} \) and \( f_{FSD} \) take discrete values and the others are continuous. If \( f_{STD} = 1 \), \( G \) is FST because \( u_f = 0 \) which means there was an \( a \in A_f \) for each \( f \in F \) producing an error signal. If \( f_{FSD} = 1 \), \( G \) is FS because \( u_i = 0 \) which means an error was signalled every time the output was erroneous. \( f_{STC} \) provides a continuous measure of FST by integrating the error signal over all faults and inputs. Optimisation of this metric guides evolution by selecting for circuits producing more and larger error signals even when the error threshold \( e_t \) is not met. This enables evolution to select between two individuals with equal \( f_{STD} \) but where one has a slightly larger error signal. Selecting for more and larger error signals is not only beneficial in guiding evolution when the error signal is below the threshold but also because by signalling an error under more inputs, faults become more detectable and the “all inputs between faults” assumption can be relaxed. \( f_{FSC} \) provides a finer evaluation of a circuit’s fault-secureness.

A circuit evaluated using the above algorithm with \( f_{STD} = f_{FSD} = 1 \) would be FTSC and will always meet the TSC goal assuming that all inputs are applied between fault arrivals. Continuing evolution to optimise \( f_{STC} \) would improve a circuit’s probability of meeting the TSC goal when the “all inputs between faults” assumption is relaxed.

### 6.4.8 Combination of Multiple Continuous Fitness Functions

Multi-objective optimisation deals with artificial evolution towards multiple, sometimes conflicting, goals and is a well researched area. A simple strategy is to calculate a weighted average and generate one fitness metric from several. When diversity is desired and some goals have no priority over others then multiple individuals on the Pareto front can be kept. When there is a definite priority between goals then the dictionary sort approach (Garvie and Thompson, 2003b) has proved successful. This works by comparing fitness metrics associated with low priority goals only when all higher fitness metrics are equal. Such a goal hierarchy exists for SC analog circuits: it is more important to produce the desired output function than to provide SC, and it is more important to provide SC than to have a low transistor count. Priorities may exist within these categories as for example making faults detectable (ST) being more important than detecting every error instance. Straightforward adoption of the dictionary sort technique is not possible because analog fitness functions may be continuous and all comparisons would then be based entirely on the highest priority fitness.

**Analog Dictionary Sort**

A solution would be to somehow discretise the fitness metrics without loosing their continuous nature which makes the landscape smooth and so get the best of both worlds. This can be done as follows. Let there be \( n \) fitness metrics \( f_i \). Now for each metric \( f_i \) define \( m \) target values \( t_i^j \) to be reached throughout evolution. Now when two individuals \( g \) and \( h \) are compared when sorting the population for rank selection then \( f_1(g) \) is compared to \( f_1(h) \). If one of them is below \( t_1^1 \) then this is the final comparison. Otherwise – i.e. if \( f_1(g) > t_1^1 \) and \( f_1(h) > t_1^1 \) – then \( f_2(g) \) is compared to \( f_2(h) \). If any is below \( t_2^1 \) then this is the final comparison. Otherwise move on to \( f_3 \) and so on until \( f_n \) is reached. If \( f_n(g) > t_n^1 \) and \( f_n(h) > t_n^1 \) then \( f_1(g) \) is again compared to \( f_1(h) \) but this
time using $t_i^2$ to judge whether to compare $f_2$. If $f_n(g) > t_m^n$ and $f_n(h) > t_m^n$ then $f_n$ is used as the comparison. This is likely only to occur when the full specification has been arrived at.

For example let $f_t$ be task fitness, $f_{ST}$ self-testing fitness, $f_{FS}$ fault-secure fitness and $f_p$ circuit size and their priorities be as listed. Then let

$$
\begin{align*}
  t_1^t &= 0.9 & t_1^{ST} &= 0.6 & t_1^{FS} &= 0.5 & t_1^p &= 0 \\
  t_2^t &= 1 & t_2^{ST} &= 0.9 & t_2^{FS} &= 1 & t_2^p &= 0 \\
  t_3^t &= 1 & t_3^{ST} &= 1 & t_3^{FS} &= 1 & t_3^p &= 0
\end{align*}
$$

These $t_i^j$ values would result in a run that optimised the task up to 0.9 fitness, then concentrated on self-testing until it reached 0.6, then optimised fault-secureness to 0.5. This would all be done while keeping the fitnesses above the previously attained thresholds. It would only start optimising $f_{ST}$ again when $f_t$ reached 1 and would only optimise $f_p$ when all higher priority fitness values reached 1.

This method allows fine-grained control over the direction evolution optimises at different stages in the evolutionary process. Incremental breeding could be automated by giving lower priority to fitness functions to be used during later stages – ie. after good fitness has been achieved for higher priority fitness metrics. Using the example in Fig. 6.1, the analog dictionary sort method here described could be used to automate selective pressure throughout evolution to first enter A, then move to B, and then to C. This would define an automatic incremental evolution (Harvey et al., 1994) process.

### 6.5 Results

#### 6.5.1 Filters

**Low pass filter**

The goal was a circuit which filtered out frequencies above 100Khz with at least -40dB attenuation, but allowed frequencies below 10Khz and was free to respond in any way to frequencies in between.

The step function was fed into I/O pin 27 on the FPTA, into the S terminal of the fourth transistor left of the bottom right one. The output was read from I/O pin 35, the E terminal of the third transistor above the bottom-right one. Mutations were only allowed to affect the bottom-right quarter of the FPTA.

Figure 6.2 shows a typical fitness graph of such an evolutionary run. It begins flat at a high (bad) fitness while the evolutionary process randomly searches for an FPTA configuration connecting the input to the output. Until then the best average and worst fitnesses are all equal except...
Chapter 6. Evolution of Analog Self-Checking Circuits

94

(a) Frequency response.  

(b) FPTA configuration.

Figure 6.3: (a) Response and (b) configuration for best LPF from a typical run. Circuit is in lower right corner of FPTA. Input is fed into the south node of the cell labelled 11/15/P, output is read from the east node of the cell labelled 15/12/N.

for some noisy spikes. Once an individual was found connecting input to output then a wider range of behaviour is available and the best gets a lower fitness and the worst gets a higher fitness. A fast optimisation period then begins in which the core filtering function is attained. Analog circuit response is continuous and so is the fitness evaluation function which constructs a smooth fitness landscape suitable to evolutionary exploration and optimisation. Small changes in circuit structure providing small fitness improvements are now continually selected until a good solution is found. The frequency response for this solution is shown in Fig. 6.3(a). It provides only slight attenuation below 10Khz and almost full - but noisy - attenuation of -40dB above 100Khz. The diagram for this circuit is shown in Fig. 6.3(b). The solution was found in roughly 5 minutes, or 2500 generations.

High pass filter

The goal was a circuit that attenuated frequencies below 10Khz with at least -40dB and allowed frequencies above 100Khz through unfiltered with no specification for frequencies in between. I/O was performed as with the low pass filter. A HPF is not trivial to build using only CMOS transistors and this time more generations were required and evolution was run for 8 hours. The response of the resulting circuit is near perfect (Fig. 6.4(a)): only frequencies just next to the cut-off are slightly out of specification. When attempting to simplify the circuit (Fig. 6.4(b)) by resetting the configuration of transistors which are disconnected from the input and output, the fitness of this individual decreases. This suggests it is using subtle physical effects of the silicon such as cross-talk, coupling and other parasitics. This dependency may hinder transfer to other mediums such as ASIC and is an obstacle to understanding it.

6.5.2 Digital to Analog Converter

The goal was for a circuit with $n$ inputs to output $5 \times \frac{x}{2^n}$ where $x$ was the binary encoded value at the input. The input encoded a 0 (1) with 0V (5V). Each of the $2^n$ voltages was applied for 20 clock cycles ($= 0.5\mu s$) and the output was read during the last one. 15 random orderings of the series of $2^n$ voltages was applied. The final output voltage for each $x$ was the average of that produced during the 15 series applications. Inputs were fed along the south side of the FPTA and output was read from the middle of the east side. Mutation was only allowed to affect the bottom
6.5.3 Op-Amp

Figure 6.7 shows fitness versus time for the first stage of the incremental breeding (§6.4.5) evolutionary process. This stage used the squared errors fitness function measuring gain and offset. After an initial optimisation period a circuit with comparator-like behaviour was found around generation 15K. This resulted in a dramatic improvement in fitness across the next 1000 generations. Optimisation continued but with strong noise in the evaluation. This noise caused evolution to go back and forth as fit individuals were lost and then formed again. 25K generations later an improved comparator circuit was found and steady optimisation continued until generation 53K where the fit circuits were lost and such fitness levels were never recovered in the next 60K generations.

The response of the best individual from the population saved at generation 52500 is shown in Fig. 6.8. As can be seen offset is extremely low as most curves perform the transition exactly at the value of $V_+$. This can be attributed to the high squared error incurred by having any offset. Amplification is not strong when $V_-$ is just below $V_+$.

Evolution was continued from this population using the explicit gain and offset fitness functions for approximately 15K generations. The response of the best individual from one of the saved populations is shown in Fig. 6.9. The offset is still very good but not quite as perfect as that of its ancestor. Gain has improved and voltage swing has grown from 1-4V to 0.5-4.5V.

The population with this best individual was then bred into the next stage adding the slew rate and settling time squared error fitness metric. At the start of this stage the response to the step function was like in Fig. 6.10(a) and Fig. 6.10(b). A slight overshoot or glitch can be seen in the response just after the step and the output settles after 1.5µs. After 5000 generations the best individual had the response shown in figures 6.11, 6.12(a) and 6.12(b). Settling time has gone down to 1µs but gain and offset deteriorated.
Figure 6.5: (a) Fitness vs. generation and (b) $V_Q$ vs. $V_{in}$ response for best from typical run evolving 6-bit DAC, where $V_{in}$ varies from 0 to 64 and is the decimal version of the binary input at the 6 inputs.

Figure 6.6: Diagram of typical evolved 6-bit DAC. Inputs are fed into the south nodes of cells labelled 3/15/P, 5/15/P, 7/15/P, 9/15/P, 11/15/P, 13/15/P and output is read from the east node of the cell labelled 15/7/P.

Figure 6.7: Average (top) and best (bottom) fitness vs. generation for typical first stage of OpAmp evolution.
Figure 6.8: $V_Q$ vs. $V_-$ response for OpAmp at end of first stage. Each curve corresponds to a different $V_+$ value ranging from 1V to 4V in 0.5V steps.

Figure 6.9: $V_Q$ vs. $V_-$ response for OpAmp at end of second stage.

(a) Step up response.

(b) Step down response.

Figure 6.10: $V_Q$ vs. time response to step (a) up and (b) down for OpAmp at end of second stage.
Figure 6.11: $V_Q$ vs. $V_-$ response for OpAmp at end of third stage. Each curve corresponds to a different $V_+$ value.

(a) Step up response.  
(b) Step down response.

Figure 6.12: $V_Q$ vs. time response to step (a) up and (b) down for OpAmp at end of third stage.

Figure 6.13: Diagram of evolved OpAmp.
Table 6.2: Attributes of best evolved Op-Amp.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain</td>
<td>20dB</td>
</tr>
<tr>
<td>Offset</td>
<td>20mV</td>
</tr>
<tr>
<td>Slew rate</td>
<td>2.95 V/µs</td>
</tr>
<tr>
<td>Settling time</td>
<td>1.85 µs</td>
</tr>
</tbody>
</table>

Figure 6.13 shows the FPTA configuration for the best circuit of the second stage. Trimming of the circuit was attempted manually by removing connections one by one. Most deletions resulted in minimal performance deteriorations - i.e. graceful degradation. Oddly, changing the configuration of disconnected cells also affected the response showing that the circuit was using parasitics and coupling. This circuit is robust but probably hard to transfer.

Table 6.5.3 lists the qualities of the best evolved Op-Amp from the second stage.

### 6.6 Conclusion

Evolution of analog self-checking is presented as a fertile field of research. Lack of existing design tools or even methods to generate analog self-checking circuits, combined with evolution’s suitability at automatically searching landscapes of analog circuits and credentials evolving digital self-checking circuits, suggest this could be a very productive research area.

A method is laid out for the evolution of analog finitely totally self-checking (FTSC) circuits which reach the TSC goal. The method extends the one used successfully for digital TSC circuits and exploits the continuous nature of analog signals to generate a more searchable fitness landscape. An adaptation of the dictionary sort approach to multi-objective optimisation is presented and it is shown how it could drive an incremental evolutionary process through different stages. The fundamental issue of transfer of evolved designs to production technology is raised and some solutions are discussed.

The FPTA designed by Langeheine et al. (2000) was chosen as a suitable medium to evolve such circuits. Some initial experiments were undertaken evolving filters, DACs and an OpAmp.

Future work could attempt achieving transfer of designs evolved on an FPTA as was done with the EM in §7, implement and evaluate the analog dictionary sort for hierarchical multi-objective optimisation, evolve other analog functions such as RNG or voice recognition, and most importantly use the method set out in §6.4.7 to evolve the first analog FTSC circuit.
Chapter 7

From an Evolvable Motherboard to a Breadboard

It has been shown how evolutionary electronics could increase the reliability of analog hardware. However, if evolved circuits are to be used in industry then it is crucial for them to transfer from the medium they were evolved in to a different medium. This chapter will present one way of achieving this.

7.1 Introduction

Transfer was defined in §6.3 above as the capability of taking a circuit evolved in one medium and producing it in another. Transfer is crucial since freeing evolved circuits from their evolutionary medium may be the only way of making them competitive in industry.

Transfer is unlikely unless measures are taken to achieve it as shown by numerous examples of unsuccessful transfer such as Thompson’s discriminator evolved on an FPGA (Thompson and Layzell, 1999) and some of Layzell’s circuits evolved on the Evolvable Motherboard (EM) (Layzell, 2001) which never transferred to simulation. Zebulum (Thompson et al., 1999) did transfer an OpAmp from simulation to a breadboard by ensuring circuits would be buildable by penalising those using too much current during evolution. Thompson (1998c) transferred a more conventional tone discriminator from several FPGAs to simulation by evaluating circuits under diverse operating conditions and Trefzer et al. (2004) transferred a comparator evolved on a FPTA into simulation by limiting search space to configurations with no floating terminals or connections and simulating some FPTA properties in the simulation.

Evolutionary electronics research at Sussex began with Thompson’s pioneering work on evolving an FPGA configuration exploiting hardware resources in unconventional and efficient ways. This configuration was never fully understood and never transferred to another technology. Layzell’s research at Sussex focused on the creation of the EM which was designed to be flexible and observable so circuits evolved with it could be analysed and understood more thoroughly. A natural progression from this line of research at Sussex is to evolve a tone discriminator on the EM, analyse it, attempt to understand it and transfer it.

This work sets out to evolve a tone discriminator on the EM and transfer it to a breadboard. This could help understand how to build an efficient tone discriminator and how to achieve transfer of intrinsically evolved circuits automatically.

7.2 Setup

The EM will now be briefly introduced. A comprehensive description can be found in Layzell (2001). The EM is a triangular matrix (Fig. 7.1) of analog switches controlling the interconnections of 48 nodes. Any node can be connected to any other node but the amount of vertical and
horizontal buses and of analog switches limits the number of simultaneous independent connections. The nodes can host any kind of electronic component and all buses can be probed with an oscilloscope. Analog switches control the connections between vertical and horizontal buses and can be turned on or off from a PC. Software designed by Layzell is used to evolve switch configurations.

The whole of the EM was subjected to evolution for this work and 10 bipolar BC108 transistors were connected to the nodes, half PNP and half NPN. These transistors will be referred to as $m_1 \ldots m_{10}$ with odd ones PNP and even ones NPN. The power, input and output lines were always connected to specific vertical buses. The multiplex genotype to phenotype mapping was used such that each horizontal bus could connect to a maximum of 6 vertical buses. The EM software was extended to allow real-time reconfiguration of the board via mouse clicks on a diagram.

### 7.2.1 Genetic Algorithm

A population based GA was used with 30 individuals, single point crossover, linear based rank-selection and elitism. Per bit mutation rate was set to 0.008. GA operation is fully described in Layzell (2001).

### 7.2.2 Evolving tone discrimination

A tone generator and analog integrator similar to those used in Thompson (1997) were connected to the EM and controlled by a PC. Either a 1Khz or 10Khz tone within the 0-5V range is generated at the EM input on demand and integration is performed for the duration of the tone. During evaluation, ten frequencies are applied to the EM for 50ms each, such that 5 are at 1Khz and 5 at 10Khz. The order of frequencies is randomised for each evaluation. The integrated response is recorded for each frequency applied. Fitness is calculated as:

$$f = k \left( \sum_{t \in s_1} i_t - \sum_{t \in s_{10}} i_t \right)$$  \hspace{1cm} (7.1)$$

where $s_1$ ($s_{10}$) is the set of trials at 1Khz (10Khz) and $i_t$ is the integrated output during trial $t$. $k = 10^{-3}$ is a scaling constant and its value does not affect GA operation.
7.3 Results

The early evolutionary period during the run (see Fig. 7.2) was characterised by a menagerie of configurations whose behaviour was not stable but at times got a lucky high fitness of up to 5. Due to their unreliability none of them stayed in the population for long. This period is followed by one in which a configuration with stable behaviour is found. This configuration is improved from fitness 2.5 to 10 by stages resembling punctuated equilibrium. Then a new stage begins in which the fitness becomes noisy again varying from 12 to 15. This noise caused the elite to be lost at generations 5400, 7300 (arrow) and 9500 at which points a hand-selected previous elite was reinserted into the population.

7.3.1 Early configurations

The first configurations with unstable but above random fitness levels adopted several strategies including:

- Charging some capacitance at the start of the evaluation and discharging it throughout.
- Affecting the input line so that 1Khz had a 2V voltage swing and 10Khz had 1V swing.
- Affecting the input line so 10Khz became a stable signal at 1V and 1Khz oscillated with less than a 2V swing.

Some of these properties are still recognisable in later configurations.

7.3.2 Functional configurations

After around 8000 generations a prototype with tone discriminating properties was evolved. Several elites from subsequent generations will now be briefly described and the elite from 8465 will then be analysed in greater depth:

5210 Output follows the input but centered around 1.5V for 10Khz and 0.7V for 1Khz and with a voltage swing of 0.3V.

8465 Output is reasonably stable at 2V when the input is 10Khz and stable at 0.3V when the input is 1Khz. The configuration is robust and performs the desired function roughly 99% of the time.

8940 Similar to 8465 but with a cleaner more stable output. However the configuration is not as robust and at times does not function at all.

9405 As 8465 but output for 10Khz reaches higher peak and dies down throughout the evaluation. This circuit would not be useful for longer frequency applications since discrimination is lost after around 100ms.
Figure 7.3: EM configuration of elite at generation 8465. Circle indicates analog switch turned on.

7.3.3 Analysis of 8465

The elite from generation 8465 was chosen to analyse mainly because of its robustness, as other tone discriminating circuits did not always function. The circuit’s EM configuration is shown in Fig. 7.3. Circuit behaviour was monitored for applications of 200ms of the input frequencies. This is four times longer than during fitness evaluation and output was verified to be stable during the period. Secondly, transistors of the same type were interchanged and this made no difference to fitness. This shows that 8465 has not adapted to the particularities of each transistor and exchanging them for nominally identical ones does not affect performance. However a slight difference in the rest voltage before each rising edge was detectable when fast successive 10Khz tests were applied and transistors $m_7$ and $m_8$ were swapped for $m_9$ and $m_{10}$.

Next, the circuit was trimmed by hand by clicking connections on or off and observing resulting circuit behaviour. This was a lengthy operation in which disconnections resulting in complete loss of functionality were rolled back. Most other disconnections resulted in a cumulative slight loss of performance. This shows that transistors and the parasitics of wires and connections were being used to produce a clearer signal. The EM configuration diagram for the trimmed 8465 is shown in Fig.7.4 while a circuit diagram for it is shown in Fig.7.5. Only three transistors and 12 EM switches are used. Tone discrimination is still performed but output is less clear. Behaviour of the trimmed 8465 was affected if different columns or rows were used for the connection points.
Figure 7.4: 8465 (Fig.7.3) hand-trimmed to minimal functional configuration.

Figure 7.5: Circuit diagram for trimmed 8465 (Fig.7.4).

Figure 7.6: Schematic of voltage levels at different parts of the trimmed 8465 in response to 10Khz (left) and 1Khz (right). Hairline is voltage at EM input after 100K resistor, dotted is $w_{78}$, dashed is $w_{85}$ and thick is EM output.
indicating that the configuration had adapted to the particularities of the analog switches it was evolved on. Making \( w_{78} \) in Fig.7.5 go through more analog switches made the stable high output during 10Khz become a 10Khz oscillation at 1.8V with a 0.2V amplitude. However adding any amount of switches at \( w_{85} \) did not affect performance. Also changing the vertical nodes on which transistors were connected and thus using different analog switches often resulted in serious performance degradation.

Figure 7.6 is a schematic of the voltage levels at different parts of the trimmed 8465 in response to the test frequencies. The originally 0-5V input tone is affected by the circuit at both frequencies. It always oscillates at the correct frequency but at 10Khz it does so within the 1.5-3.5V range and at 1Khz within the 0-3.5V range. Rising edges are sharp but the falling edge at 1Khz shows signs of some capacitance discharging. Voltage at \( w_{78} \) follows the input with a delay of roughly 20\( \mu s \) but with softer edges. This is expected as a high value at the tone will allow current to flow from the voltage source to \( w_{78} \) while a low value will block current flow from the voltage source but allow leakage through \( m_8 \). This high value at the tone is possible due to the 100K\( \Omega \) resistor between the tone generator and the tone input to the circuit. \( w_{85} \) shows signs of tone discrimination as it has its mean roughly 0.5V above the minimum voltage at the tone input which is 1.5V or 10Khz and 0V for 1Khz. It also shows peaks when \( w_{78} \) and \( w_{85} \) differ. Output voltage follows \( w_{85} \) but peaks 0.4V lower.

### 7.3.4 Transferring 8465 to a breadboard
Straightforward implementation of the circuit in Fig.7.5 did not produce a tone-discriminator, it produced a circuit which oscillated at the tone frequency centered around 0.4V. This shows that the design exploits the parasitics at wires and switches to operate and highlights the problem of transfer. The strategy adopted to achieve transfer in this work was to simulate the evolutionary medium particularities. The first attempt to do this involved introducing 150\( \Omega \) resistors in the places where analog switches were in the original EM implementation. Neither this nor using 100\( \Omega \) nor 300\( \Omega \) resistors improved behaviour. Next, recalling that the response in Fig.7.6 hinted at the presence of capacitance, the resistors were removed and a capacitor in the nF range was introduced at \( w_{78} \). This made voltage at \( w_{78} \) in response to the tones similar to that at the tone input on the EM: a primitive tone discriminating behaviour. Adding a similar capacitor at \( w_{85} \) had no impact on behaviour. However adding a 53nF capacitor at \( w_{85} \) restored tone discriminating functionality. This is true even when \( m_5 \) is completely removed bringing the transistor count to two. The diagram of the transferred evolved tone discriminator is shown in Fig.7.7. Given that tone discriminating behaviour was restored after only two attempts at inserting different capacitances, it is not believed that the capacitances restoring functionality must be exact. Also, the values working
for the particular breadboard used may differ from those required to implement this circuit in a different medium such as a simulator.

### 7.4 Conclusion

A tone discriminator was evolved intrinsically on the EM. This required 8000 generations to evolve yet it only depended on 3 of the 10 transistors available for functioning. One evolved circuit was analysed by exchanging the transistors and interconnect resources it used, trimming it down to a minimal functional form, and observing its internal nodes during operation using an oscilloscope. The understanding gathered was not very deep given the author is not a specialist in analog electronics, but was crucial for achieving transfer. The evolved design was successfully transferred to a breadboard by simulating the EM parasitics using capacitors. This was a tedious process but is a proof of principle that such hardware to hardware transfer is possible and that circuits evolved with the EM with the current method could be put to practical use without requiring an EM. *Hardware to hardware transfer is required if an evolved design which uses physical properties not present in any simulator is to be used out of its evolutionary medium.* The final evolved design implemented on the breadboard used two bipolar transistors and two capacitors to discriminate between 10Khz and 1Khz.

### 7.5 Future Transfer

This section will discuss how transfer could be facilitated in future. During this work it was found that evolved circuits were over-adapting to particularities of the interconnect parasitics. This could be avoided by varying the interconnect used to implement each circuit during evaluation.

On the EM this could be achieved in several ways. One way would be to allocate a spare column with two spare transistors. During each evaluation this spare would replace another column at random thus varying both interconnect used and transistors. Another way would be to allocate half of the EM as spare. This would be enough to maintain a nominally identical configuration but completely randomise the interconnect and transistors used by using rerouting. Another method could be to add another set of transistors on the horizontal buses and rotate the configuration by 90 degrees. In addition to all of these techniques the properties of each connection between two nodes could be altered by adding more connections in series or in parallel thus varying parasitic resistance and capacitance.

Similar techniques could be used on the FPTA. Robust circuits could be evolved by varying the section of the chip used for fitness evaluation. It may also be possible to evaluate several circuits on different sections of the chip simultaneously by rotating the designs so their I/O is always on the border. Another possibility is to allocate spare rows and columns and use these to inject extra interconnect between cells introducing extra connections in series, parallel or through permanently switched on transistors. This would dramatically influence the parasitic capacitance and resistance. To reduce cross-talk the configuration for cells around the design could also be randomised during each evaluation.

The techniques above could be used in conjunction with an automatic minimisation algorithm. This could be implemented as extra evolutionary pressure towards parsimony as in §5.3.1 or an process automating the trimming process used above in §7.3.4. This would not only produce circuits that would be cheaper to produce but they would also be simpler to transfer. They would also depend less on interconnect because less of it would be involved. This could also be combined with a technique like the one described in Trefzer et al. (2004) which ensures no floating terminals or connections are present in the designs throughout evolution.
Chapter 8

Conclusions

It will now be demonstrated how this thesis shows the objectives listed in §1.5 to be satisfied.

1. Jiggling has been introduced as a novel mitigation method for dealing with permanent local damage in FPGAs which can be used in parallel to Flatrolling to mitigate transient faults. It is shown how the Jiggling repair mechanism itself implemented on the FPGA can protect many other systems and is small enough that a pair could mutually repair each other if the method were extended to sequential circuits. Reliability analysis has been provided showing that some benchmarks protected with Jiggling could survive 190 years above 0.99 reliability with a high permanent local fault interarrival rate of 2 years. Methods for predicting the effectiveness of Jiggling for different benchmarks are demonstrated and several trade-offs between repair mechanism overhead and repair time are shown to be available.

2. The IslandEv architecture to distribute an evolutionary process across the Internet has been introduced. It was used to spread a process evolving circuits with concurrent error detection (CED) using hundreds of gates across the spare processing power of 150 university workstations and 300 computers from donors worldwide. A supercomputer of an average 840Ghz was created with no cost. This was the first example of a GA being distributed across the Internet on computing power from the public.

3. Evolutionary synthesis has been shown to be the first automatic method to generate circuits with CED having no explicit separation between functional and checking logic.

4. This method was also the first to add logic around an irredundant original circuit to give it full CED without imposing any structural constraints, succeeding for all benchmarks attempted.

5. This method was also shown to be the first to adapt the self-checking strategy to the particularities of each circuit.

6. The method was also shown to be the first to apply multiple well suited self-checking strategies within single circuits.

7. Evolved circuits with CED used on average less than 30% of duplication overhead, a figure previously unheard of in the literature. Overhead was still under 40% duplication even when no modifications were allowed to the original function generating module.

8. The design principle of error-propagating blocks was extracted from evolved self-checking circuits and could be used to augment a standard circuit synthesis tool so it only generates TSC networks.
9. The class of Absolutely Fault Secure circuits was introduced as those meeting the TSCG under no assumption of fault arrival timing. It was shown that such circuits do exist and that evolutionary synthesis is the first method to automatically generate them.

10. It has been postulated how analog circuits with CED could be synthesised using evolution and how the complexity of analog reliable hardware design is well matched to evolution’s capacity at searching spaces encoding analog circuits by iteratively improving a design keeping a holistic view of its behaviour.

11. A tone-discriminator evolved on the EM was simplified and transferred to a bread-board such that the final design distinguished between two tones using two transistors and two capacitors. This was done by simulating the parasitics of the EM on the breadboard. Such hardware to hardware transfer is a requisite if a circuit evolved to use physical properties not present in a simulator is ever to be implemented in a different medium.

General contributions made to the field of EE include a more flexible feed-forward mapping (§5.3.1) containing more interconnectivity and neutral networks in the fitness landscape, a mutation rate (§5.3.1) similar to simulated annealing that decreases as circuit fitness increases which probably dynamically ensures roughly one mutation affecting behaviour per genotype, a block-copy genetic operation (§5.3.1) enabling replication of useful sub-structures throughout a design, a novel method for generating random and compact test-patterns for sequential circuits (§5.4.1), and a fitness function based on the statistical correlation of the actual and desired output series (§3.4).
Appendix A

Appendices

A.1 Proof of expressions in Table 3.5

A.1.1 A - All Once

Storage space \( s = 2^I + 2 \lceil \log_2 (2^I Q) \rceil \).

**Proof.** There are \( 2^I \) inputs and one bit is required for each to signal if it has been seen. The two counters must be large enough to count to \( 2^I Q \) since that is the amount of output data and it may all be high. \( \lceil \log_2 x \rceil \) bits are necessary to represent a number \( x \) in base \( n \).

\[
O(s) = O(2^I).
\]

**Proof.** Assume circuit output count grows as fast as input count. Then the first term in \( s \) dominates.

Time \( t \) \( < 2^I (1 + I \ln 2) \). The proof for this expression uses Lemma 1:

**Lemma 1:**

\[
\sum_{x=1}^{n} \frac{1}{x} < 1 + \ln n
\]

**Proof of Lemma 1.** Consider the area within the bars defined by \( y = \frac{1}{\lceil x \rceil + 1} \) from 0 to \( n \). Each bar is of width 1 and their height varies as the series \( 1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{n} \); so the area is \( \sum_{x=1}^{n} \frac{1}{\lceil x \rceil + 1} \). Now consider the area under the curve \( y = \frac{1}{x} \) from 1 to \( n \). Since \( \frac{1}{x} > \frac{1}{\lceil x \rceil + 1} \) then \( \int_{1}^{n} \frac{1}{x} dx > \int_{1}^{n} \frac{1}{\lceil x \rceil + 1} dx \). Since \( \int_{0}^{1} \frac{1}{x+1} dx = 1 \) then \( 1 + \int_{1}^{n} \frac{1}{x} dx > \sum_{x=1}^{n} \frac{1}{\lceil x \rceil + 1} \). Hence, \( 1 + \ln n > \sum_{x=1}^{n} \frac{1}{x} \).

**Proof of the expression for \( t \):** Inputs are assumed to be fully random at each clock cycle. If there is a probability \( p \) of a desired input being applied then by the geometric distribution the expected number of clock cycles to wait until it is applied is \( g(p) = \frac{1-p}{p} \). When evaluation starts any input will be new, so \( p = 1 \). The first input seen, any of the remaining inputs will be new so then \( p = \frac{2^I - 1}{2^I} \). Then only \( 2^I - 1 \) are unseen and so forth until the last unseen input where \( p = \frac{1}{2^I} \). Total evaluation time is the sum of the expected waiting time to see all inputs once plus the time to actually see them:

\[
t = \sum_{x=1}^{2^I} \left( g \left( \frac{x}{2^I} \right) + 1 \right) = \sum_{x=1}^{2^I} \left( \frac{1 - \frac{x}{2^I}}{\frac{x}{2^I}} + 1 \right) = \sum_{x=1}^{2^I} \left( \frac{2^I - x}{x} + 1 \right) = 2^I \sum_{x=1}^{2^I} \frac{1}{x}
\]
Now using Lemma 1:
\[ t < 2^l (1 + \ln 2^l) = 2^l (1 + l \ln 2) \]

\[ O(t) = O(2^l) \]

**Proof.** The second term dominates and the constant is irrelevant.

\[ O(s \times t) = O(2^l) \]

### A.1.2 B - In Order

Storage space \( s = l + 2^l \left\lfloor \log_2 (2^l Q) \right\rfloor \).

**Proof.** \( I \) bits are needed for the input counter because it must count over \( 2^l \) values and the other two counters are as in A.

\[ O(s) = O(I) \] as in A.

Time \( t = 2^l \).

**Proof.** There is a probability \( \frac{1}{2^l} \) of finding the sought vector. \( 2^l \) vectors must be found and it takes one step to see each one:

\[ t = 2^l \left( g \left( \frac{1}{2^l} \right) + 1 \right) = 2^l \left( \frac{2^l}{1} - 1 + 1 \right) = 2^l \]

\[ O(t) = O(2^l) \]

\[ O(s \times t) = O(2^l) \]

### A.1.3 C - Count & Table

Storage space \( s = m + 2^{l-m} + \left\lfloor \log_2 (2^l Q) \right\rfloor \).

**Proof.** \( m \) bits are used for the counter indicating the current input partition. There are \( 2^{l-m} \) inputs per partition and each requires a bit to signal if it has been seen. Finally, two counters are used as in A.

\[ O(s) = O(2^{l-m} + m) \]

**Proof.** Which term dominates will depend on the size of \( m \).

Time \( t < 2^{l+m} (1 + (I-m) \ln 2) \).

**Proof.** There are \( 2^m \) input partitions of \( 2^{l-m} \) inputs each. For each partition, inputs are sought similarly to A only that the first input (when none from the partition have been seen) has \( p = \frac{2^l}{2^m} \), the second \( p = \frac{2^{l-m-1}}{2^m} \) and so on. Total evaluation time is:

\[ t = 2^m \sum_{x=1}^{2^{l-m}} \left( g \left( \frac{2^x}{2^l} \right) + 1 \right) = 2^m 2^l \sum_{x=1}^{2^{l-m}} \frac{1}{x} \]

Using Lemma 1:

\[ t < 2^{l+m} (1 + \ln(2^{l-m})) = 2^{l+m} (1 + (I-m) \ln 2) \]

\[ O(t) = O(2^{l+m} (1 + (I-m))) \].
Proof. Which of 1 or $I - m$ will dominate depends on the size of $m$.

$$O(s \times t)=O(2^{2I/I})$$

Proof. $O(s) \times O(t) = (2^{l-m} + m)(2^{l+m} + 2^{l+m}(I - m)) = 2^{2I} + 2^{2l}(I - m) + m2^{l+m} + 2^{l+m}m(I - m)$. When $I > m$, the expression is dominated by the term $2^{2I}$. For the case $I = m$ the expression also reduces to $I2^{2I}$.

A.1.4 D - Keen Counters

Storage space $s = k\lceil \log_2 \frac{2^{2l+1}}{k} \rceil + 2\lceil \log_2 (2^l Q) \rceil$.

Proof. There are $k$ counters. Each counts over a partition of size $2^{2l}$ since it must be allowed to overlap into a neighbouring one. Finally, two counters are used as in A.

$$O(s) = kI$$

Proof. $s < kI\log_2 2 + 2l\lceil \log_2 Q \rceil$ as in A. The dominating term in this expression is $kI$ since $k > 1$.

Time $t = \frac{2^{3l}}{k}$.

Proof. At each step there is a probability $\frac{k}{2^l}$ of finding a new input. $2^l$ of these must be found and it takes one step to see each one:

$$t = 2^l \left( g \left( \frac{k}{2^l} \right) + 1 \right) = 2^l \frac{2^l}{k} = \frac{2^{2l}}{k}$$

$$O(t) = O\left(\frac{2^{2l}}{k}\right).$$

$$O(s \times t) = O(2^{2I/I}).$$

A.2 Distributing a GA Step by Step

A.2.1 Interfacing any GA package to IslandEv

To interface IslandEv with your evolutionary process you must implement the following interfaces found in the islandev package:

- **Individual**: Must be implemented by the object representing a member of your population. An individual is assumed to have a fitness value represented as a double and may also hold more information accessed through the `getProperty` method. The methods to be implemented are:

  - `double getFitness()`: This should return the fitness of this individual. For multi-objective runs this could be the fitness component associated with the first objective.

  - `Object getProperty( int index )`: This should return the `index`th property of this individual. Eg. a property could be a `Double` representing another fitness value, or tracing information referencing the individual’s parents. Providing these properties is not essential but may be useful since their values are printed and logged by the server.
- int compareTo( Object o ): Compares this individual with the specified individual for order. Returns a negative integer, zero, or a positive integer as this individual is less fit than, equal to, or fitter than the specified individual. This is used by IslandEv for finding out which is the fittest solution found so far and if it has reached target fitness. A simple implementation for single fitness individuals could use the pre-existing comparison method for doubles: return new Double( getFitness() ).compareTo( new Double( (( Individual ) o ).getFitness() ) );. The comparison may be more complex for individuals with multiple components of fitness.

- EvolutionaryInteractiveTask: This interface should be implemented by an object in your evolutionary engine which has access to information about the current state of evolution. The following methods must be implemented:

  - int[] getPopulationSizes(): Should return the sizes of the populations in your evolutionary process. For a single population setup use int[] rv = { popSize }; return rv; where popSize is a variable in your code holding the population size.

  - Individual getSolutionTemplate(): Should return an Individual object sol such that, if for any other Individual object ind, the expression sol.compareTo( ind ) < 0 is true, then ind is considered a solution to the problem being tackled by evolution.

  - SnapshotPainter getSnapshotPainter(): A SnapshotPainter object is simply one capable of returning a JComponent. This component will be placed into the GUI displayed on the client’s screen so they have a graphical view of the progress of their island. For an empty panel use return new EmptySnapshotPainter();.

  - Object run( Object params ): This method should launch the evolutionary process. It will be called as soon as the EvolutionaryInteractiveTask object is received by the client.

  - void set( Object params ): This method is used to send information to this evolutionary process from the server. The server will do this in two cases:

    1. When it starts up – and the EvolutionaryInteractiveTask object is still on the server – and finds a log file with the latest snapshot of the population at the coordinate assigned to this evolutionary process. Then it will provide the log file name through params as a String such that the index of the population to load must be appended to the end of this file name so for population 0 the file name is params + "0". This case can be ignored with if !( params instanceof String ) with the effect that all islands will start from the initial population as defined by your GA process.

    2. At regular intervals when the EvolutionaryInteractiveTask is running on the client, the server will call this method to inject immigrants into its population. In this case the server calls this method with params equal to a Vector holding Individual objects to be inserted into each population. For a single population process then the vector will contain only one individual.

  - Object get( Object params ): This method is called regularly by the server to check on the progress of this evolutionary process and to retrieve emigrating individuals. This method should return a Vector holding the following in index order:

    0. A Vector holding the best individual of each population. One Individual object per population.

    1. A Vector holding the generation number for each population. One Integer object per population.

    2. A Vector holding the average fitness of each population. One Double object per population.
3. A Vector holding an individual selected for outbound migration for each population. One Individual object per population.

4. This is an optional element and is a Vector holding arrays of populations to be logged as a record of this island’s progress. For single population setups these arrays will be of size one. These populations will be logged to a file in the order they appear in the vector by calling their toString method. The most recent (last in vector) populations will be saved in special files.

Once you’ve made your evolutionary package implement these two interfaces you can start distributing it with IslandEv. The next section describes how.

A.2.2 Setting up your distributed islands evolutionary process

This section will assume that you know how to create an object of type EvolutionaryInteractiveTask. If you are using JaGa then this is the Monica object as described in the JaGa getting started guide jag (2004). If you are using another evolutionary package then this is the object implementing EvolutionaryInteractiveTask as mentioned in the previous section.

Creating the server object

The server can be given a queue of evolutionary tasks to go through in order. As soon as an individual is found superior or equal to the solution template, the server moves on to the next task and updates all clients. The server keeps log files showing when it starts a problem, when it finds a solution, when clients connect, etc. The migration rate between islands is configurable by setting the probability \( p \) that a migration event will succeed.

The following code snippet shows how to create a server object using the IslandsEvolutionServerWrapper helper class:

```java
import islandev.*;
import distrit.*;
import java.util.Vector;

public class IslandEvExample extends IslandsEvolutionServerWrapper {
    public IslandEvExample() {
        EvolutionaryInteractiveTask evoTask = getEvoTask();
        String evoTaskName = "MyEvoTask";
        Vector taskQ = new Vector(); // Queue of tasks
        Vector taskQNames = new Vector(); // Names of tasks in task queue
        taskQ.add( evoTask );
        taskQNames.add( evoTaskName );

        String logDir = "/home/username/log/";
        String logFileName = "evoLog.txt";
        double migrationProbability = 0.5; // This is p

        IsEv = new IslandsEvolutionServer( taskQ, taskQNames, logDir, logFileName, migrationProbability );
    }

    protected static EvolutionaryInteractiveTask getEvoTask() {
        // ... // code that creates an EvolutionaryInteractiveTask.
    }
}
```

Now that we have our server class IslandEvExample we can proceed to start the server.

Starting the server

IslandEv uses Remote Method Invocation (RMI). The first step is to configure the Java security policy. This is done with a policy file which will be assumed to be named java.policy and looks like:

```java
grant{
    permission java.net.SocketPermission "host:1024-65535", "connect,resolve";
    permission java.net.SocketPermission "host:80", "connect";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.lang.RuntimePermission "modifyThread";
};
```

This allows the server to use various sockets and to have file access to the /home/username/log directory.

To start the server:
1. Start the rmiregistry: type `rmiregistry` at the command line. **Important:** this must be done from a directory without direct access to your class files.

2. Start a webserver (any standard one will do) with its root at the root of your class file structure. Eg. if your webserver is running on host `host` on port `port` then `http://host:port/islandev/Individual.class` should point to the class file for the `Individual` interface.

3. Start your server. This can be done using the `SingleServerWrapper` class included with DistrIT by typing at the command line:

   ```java
   java -Djava.rmi.codebase=http://host:port/ -Djava.rmi.server.hostname=host -Djava.security.policy=java.policy distrit.server.SingleServerWrapper
   ``

   where `IES` is the RMI binding name of your choice for your server.

   Now the islands evolution server will be up and running waiting for clients to connect.

**Connecting the clients**

All the clients need to have is the generic 5Kb DistrIT client ¹ and a java.policy file granting connection to the server at host:

```java
grant{
    permission java.net.SocketPermission "host:1024-65535", "connect,resolve";
    permission java.net.SocketPermission "host:80", "connect";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.lang.RuntimePermission "modifyThread";
};
```

Now you can launch the client with: `java -Djava.security.policy=java.policy -jar ITClient.jar host IES clientId`

`clientId` is a unique name used by the server to refer to this client during evolution. If IslandEv is configured to log statistics into a MySQL database then this name will be used to refer to this client and two optional parameters can be appended to this command specifying the client’s email and the name of the team it belongs to.

Now this client should connect to your IslandEv server, download the evolutionary process and start running it.

**A.3 Output function generator synthesis using Sis**

**A.3.1 Combinational**

Sis E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and Sangiovanni-Vincentelli (1992) was used to synthesise and optimise the benchmarks from the MCNC’91 test suite into two input gate technology using all possible gates. The following procedure of general acceptance was used:

``````
full_simplify
print_stats -f
source script.rugged
```

These three commands were used in order and the last two were repeated until the number of factored literals went up. The circuit used was that resulting previously to the last application of `source script.rugged`. The same procedure was effected without the `full_simplify` at the start and the circuit with lowest factored literals was used for the second stage.

The second stage applied the following script:

``````
sweep
simplify
sweep
```

¹ it can be downloaded from http://prdownloads.sourceforge.net/distrit/ITClient.jar?download
simplify
xl_split -n 2
sweep
simplify
xl_split -n 2
sweep
xl_partition -n 2
sweep
sweep
xl_partition -n 2
sweep
xl_k_decomp -n 2
sweep
xl_cover -n 2 -h 3

The resulting circuit was then used as the output function generator to seed evolution with and is the one used to calculate duplication overhead. Unfortunately Gunther and Drechsler (1999) it is not possible calculate the theoretically minimum size of a circuit instantiation. Exhaustive search can be used for small circuits but is not practical for larger ones.

A.3.2 Sequential

For sequential circuits synthesis was started with:

state_minimize stamina -s 1
state_assign nova -e ih

and continued as above but replacing -n 2 for -n 4.
Bibliography


Brownsword, L. and Morris, E. (2003). The good news about COTS. news@sei interactive, 1Q03.


