# Automatic design of digital circuits using grammatical evolution and SystemVerilog

# Automatic Design of Digital Circuits Using Grammatical Evolution and SystemVerilog


Michael Kwaku Tetteh

Lero | Science Foundation Ireland Research Centre for Software

Faculty of Science and Engineering

Department of Computer Science and Information Systems

University of Limerick

Submitted to the University of Limerick for the degree of

*PhilosophiæDoctor (PhD) 2024*

1. Supervisor: Prof. Conor Ryan
Department of Computer Science and Information Systems
University *of* Limerick
Ireland

2. Co-Supervisor: Dr. Douglas Mota Dias
Department of Computer Science and Information Systems
University *of* Limerick
Ireland

1. External Examiner: Dr. James McDermott
School of Computer Science
University *of* Galway
Ireland

2. Internal Examiner: Dr. Patrick Healy
Department of Computer Science and Information Systems
University *of* Limerick
Ireland

1. Chair: Dr. Meghana Kshirsagar
Department of Computer Science and Information Systems
University *of* Limerick
Ireland

Day of the defense: 22nd March 2024

Signature from head of PhD committee:

# Abstract

Digital circuit design is a very complex and time-consuming task which requires a great deal of skill. Evolutionary Algorithms have been applied to circuit design problems and have been demonstrated to be better and more creative at exploring the circuit design search space than humans.

In industry, digital circuit design is greatly aided by the use of Hardware Description Languages (HDLs) and powerful logic simulators. HDLs permit circuit designers to design circuits at a very abstract level. These are then tested and synthesised before being committed to hardware.

We present *Automatic Design of Digital Circuits* (ADDC), a system which uses Grammatical Evolution (GE) to design digital circuits using an HDL and a logic simulator. First, we investigate the applicability of ADDC to circuit design and how best to configure it, with a focus on grammar design and selection operators. Grammar design is a key aspect of GE since grammars define the boundaries, size and topology of the problem search space. We observed from the results that experimental setups with grammars designed with domain knowledge recorded a higher success rate compared to grammars designed with no domain knowledge.

Secondly, we investigate the capability of ADDC to evolve more complex circuits by exploiting the use of higher-level language constructs such as `if-else`, `always block`, `switch-case`, and `generate for-loop` in SystemVerilog (an HDL). With the incorporation of these higher-level constructs, to date, ADDC has evolved the most complex selective parity, adder, and multiplier circuits in the relevant literature.

Finally, there are many digital circuits characterised by multiple outputs. However, both single and multiple output problems are typically represented using a single genome. We propose a new multi-genome GE, together with associated genetic operators, to help mitigate the ripple effect GE suffers from. The ripple effect in GE is a phenomenon whereby a small change (such as a mutation event) in an individual's genotype, usually in the beginning regions, results in a drastic change in its phenotype after the mapping process. This issue can be very problematic when solving multi-output problems using standard GE, as there is no mechanism to stop searching for an output once it has been solved and, instead, direct the search operators to regions of the genome coding for the unsolved outputs. Hence, due to the destructive effect of GE's search operators, solving such problems is challenging. A genome per output representation for a multi-output problem ensures ripple effects are localised. Results demonstrate the multi-genome GE outperforms standard GE on multi-output circuit benchmark problems.

# Declaration

I herewith declare that I have produced this thesis without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This thesis has not previously been presented in an identical or similar form to any other Irish or foreign examination board.

The thesis work was conducted from 2018 to 2024 under the supervision of Prof. Conor Ryan and Dr. Douglas Mota Dias at University of Limerick.

Limerick, 2024

# Acknowledgements

Thank God for sustaining and granting me the gift of life throughout this journey.

I am extremely grateful to my supervisor, Prof. Conor Ryan, for the supervision, expertise, invaluable advice and direction throughout the PhD journey. His balance of guidance and the freedom he allowed me to explore ideas independently has been immensely valuable.

I want to express my deepest gratitude to my Co-Supervisor, Dr. Douglas Mota Dias for your expertise and constructive feedback which have been crucial in refining my research.

Special thanks to JJ Collins through whom I got to know of this PhD opportunity and had the privilege of working with.

I am also grateful to the examiners Dr. James McDermott and Dr. Patrick Healy for their comments and constructive suggestions. Your feedback and meticulous attention to detail have significantly enhanced the quality and clarity of my research.

My sincere thanks to the chair of the viva, Dr. Meghana Kshirsagar, whose adept coordination and facilitation ensured a smooth examination process.

I extend my heartfelt thanks to my colleagues at the BDS lab. Their collaboration and insightful discussions have greatly enriched my research experience.

Finally, to my supportive family, who consistently checked on my well-being and progress throughout my entire PhD journey, I am deeply appreciative.

To my family and friends, whose unwavering support has been my cornerstone. To my late aunt, Dr. Francisca, who was like a second mother to me – her kindness, selflessness, and enduring love profoundly shaped me and inspired my journey. I also dedicate this work to my late brother Collins, in memory of the cherished moments we shared growing up. This achievement is a tribute to their memories.

# Publications

1. Behavioural Modelling of Digital Circuits in System Verilog using Grammatical Evolution.
   Conor, Ryan; Michael Kwaku, Tetteh; Douglas Mota Dias.
   Proceedings of the 12th International Joint Conference on Computational Intelligence, IJCCI 2020.
   https://doi.org/10.5220/0010066600280039
2. Evolution of Complex Combinational Logic Circuits Using Grammatical Evolution with SystemVerilog.
   Michael, Tetteh; Conor, Ryan; Douglas, Mota Dias. EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming.
   https://doi.org/10.1007/978-3-030-72812-0_10
3. Grammatical Evolution of Complex Digital Circuits in SystemVerilog.
   SN Computer Science 3, 188 (2022).
   Michael, Tetteh; Conor, Ryan; Douglas, Mota Dias.
   https://doi.org/10.1007/s42979-022-01045-9
4. ADDC: Automatic Design of Digital Circuit.
   Conor, Ryan; Michael, Tetteh; Jack, McEllin; Douglas, Mota-Dias; Richard, Conway; Enrique, Naredo.
   IntechOpen, Genetic Algorithms (2022)
   https://doi.org/10.5772/intechopen.104410
5. Evolving Multi-Output Digital Circuits Using Multi-Genome Grammatical Evolution.
   Michael, Tetteh; Allan, de Lima; Jack, McEllin; Aidan, Murphy; Douglas, Mota Dias; Conor, Ryan

Algorithms 2023, 16, 365. https://doi.org/10.3390/a16080365

# Contents

# List of Tables

# List of Figures

# List of Listings

# 1

# Introduction

The central hypothesis of this thesis is that Grammatical Evolution (GE) can be used to evolve fully functional and complex digital circuits in SystemVerilog—a Hardware Description Language (HDL).

According to Moore's law, the number of transistors on a chip will double every 18 months. The exponential growth of the number of transistors on a chip directly increases the integrated circuit (IC) design complexity. The complexity increases even further in combination with other factors, such as customer needs and standards.

Circuit design is a highly skilled and demanding job, a slow and expensive process (Rabey et al., 2003), with minor design errors costing millions of dollars to fix. Errors in circuit design have serious consequences, not only financially in the case of Intel, where the bug known as FDIV in the Intel Pentium processors cost 475 million dollars in replacing the products, but also their reputation (Kai-hui et al., 2009). Also, Therac-25, the cancer treatment medical device, had a fatal design error resulting in the overdose of six patients with radiation, resulting in serious injuries and deaths between June 1985 and January 1987 (Kai-hui et al., 2009; Leveson and Turner, 1993).

Circuit Design has evolved from the use of vacuum tubes to the use of ICs. This advancement has made digital circuits ubiquitous in the modern world, but as their complexity increases, so does the difficulty associated with their design. In order to make the design task tractable, HDLs were designed. These

abstract-level circuit designs are heavily tested using logic simulators to ensure functional correctness. Functionally correct circuits are then advanced through logic synthesis before fabrication takes place on a chip.

Evolvable Hardware (EHW) is a field that applies Evolutionary Algorithms (EAs) to circuit design tasks, which was conceived in the early 1990s (Higuchi et al., 1996; Higuchi et al., 1997; Murakawa et al., 1996). The choice of EAs for circuit evolution is motivated by the fact that EAs were demonstrated to be better and more creative at exploring the circuit design search space than humans (Thompson, 1996). A primary objective of EHW is the development of next-generation autonomous re-configurable hardware by means of artificial evolution (Sekanina, 2006; Yao and Higuchi, 1997). Such hardware should be well-suited for use in environments characterised by extreme conditions which render it unsafe and inaccessible to humans. An example application area is space exploration (Keymeulen et al., 2006; Lohn et al., 2000; Zebulum et al., 2005). However, a major and basic requirement of an adaptive hardware system is the ability to evolve its circuitry.

Despite the immense progress made in the EHW field since its inception, scalability issues have long hampered its progress. This is because existing methods mostly design circuits at the gate level using EAs. A major benefit of gate-level evolution is its capability to evolve optimised novel circuit designs, that is, circuits using fewer gates compared to human-designed circuits. For example, a 3-bit $\times$ 3-bit and a 4-bit $\times$ 4-bit multiplier evolved in (Vassilev et al., 2000) were 23.3% and 10.9% more efficient in terms of the number of two-input logic gates used, respectively. However, the drawbacks of gate-level evolution are the inability to scale to evolve complex circuits (Ali et al., 2004) due to the fact that designs are less interpretable and that they are difficult and take more time to verify than gate-level designs. These drawbacks account for some of the major issues confronting EHW.

## 1.1 Objectives

How to efficiently evolve complex circuits is a question a number of EHW researchers are constantly seeking answers to. Existing approaches range from

the use of bigger functional blocks (also known as function level evolution) and novel decomposition techniques to the improvement of genetic operators such as crossover and mutation. However, the evolved circuits are relatively complex as they are evolved at the gate level, which is not easily scalable as noted above.

This thesis presents the use of GE to evolve digital circuits in HDL at a higher abstraction level other than the gate level as a plausible solution to EHW scalability issues. GE is a Genetic Programming (GP) variant capable of evolving programs in any arbitrary language that is Backus-Naur form (BNF) compliant (Ryan et al., 1998).

The first objective of this thesis is to design and implement *Automatic Design of Digital Circuits* (ADDC), a highly configurable evolutionary framework for the functional evolution of digital circuits in HDLs using GE.

The second objective is to further improve ADDC's robustness by investigating the impact of grammar design and the choice of initialisation routines on GE's evolutionary performance.

The third objective is to assess ADDC's ability to evolve complex circuits at a higher level of abstraction using expressive programming constructs in SystemVerilog ( and HDLs in general), as a viable solution to EHW scalability issues.

Up until this point, both single-output and multi-output circuits were evolved using the standard GE linear genome. The last but not the least objective is to design and implement a GE variant which uses a better encoding structure (multi-genome encoding) better suited for multi-output circuits, which are usually more challenging to deal with.

## 1.2 Research Contributions

This thesis attempts to address the scalability issues in EHW and, by so doing, makes the following primary research contributions:

- Firstly, ADDC is introduced, an evolutionary framework which uses GE, SystemVerilog and Icarus Verilog, a circuit simulator for the functional evolution of digital circuits. ADDC is technology agnostic and easily configurable as the choice of HDL and simulator are left to the user. Results

obtained from several experiments applying ADDC to digital circuit benchmark problems indicate it is a robust tool for circuit design tasks.

- Secondly, grammars designed with the incorporation of simple domain knowledge extracted from the respective circuit benchmark problems and the choice of initialisation routine significantly improve the evolutionary performance of ADDC. Experimental setups with grammars designed with simple domain knowledge outperformed their grammar counterparts with no domain knowledge.

- Thirdly, the exploitation of programming constructs such as `generate for` loops, `if else`, `always` blocks and several types of operators available in SystemVerilog facilitates the evolution of complex circuits. Circuits evolved are substantially larger than the current state of the art for evolutionary approaches.

- Finally, MGGE, the proposed GE-variant which uses multiple genomes to evolve a unique output for multi-output circuit problems, is shown to be better suited for evolving such problems compared to standard GE.

## 1.3 Thesis Outline

The subsequent chapters of this dissertation are organised as follows:

Chapter 2 provides an overview of EAs and a concise introduction to GE. The components of GE are described in detail as GE is the EA of choice used to conduct experiments in this dissertation.

Chapter 3 provides an introduction to the field of EHW. The chapter covers the progress made in the field, the challenges and proposed solutions to address them.

Chapter 4 presents a detailed description of the ADDC framework designed for the evolution of digital circuits using GE, SystemVerilog and a circuit simulator. The chapter also covers an introduction to SystemVerilog and the standard Computer-Aided Design (CAD) flow for digital circuit design. Finally, the ADDC

framework is tested on combinational circuit benchmark problems, and the results are presented.

Chapter 5 extends the work presented in Chapter 4 by investigating the effect of grammar design choices and the choice of GE initialisation routines on evolutionary performance using combinational and sequential digital circuit benchmark problems.

Chapter 6 focuses on exploiting high-level programming constructs in SystemVerilog to evolve fully functional and parameterised circuits at a more abstract level other than gate-level.

Chapter 7 presents an MGGE better suited for tackling multi-output problems, specifically digital circuits.

Chapter 8, the final chapter, summarises the research conducted in this dissertation, the key research contributions, and future directions.

# 2

# Grammatical Evolution

## 2.1 Introduction

This chapter presents a concise introduction to GE, the EA of choice used to conduct experiments in this work. GE is a bio-inspired algorithm capable of evolving programs in an arbitrary language. GE has been applied to several domains such as image processing, business modelling, bio-informatics, music, architecture, and business modelling (Ryan et al., 2018)

Additionally, EAs with a focus on the underlying generic evolutionary process of these algorithms are covered briefly to help form the base foundation upon which the research presented in this body of work is carried. Also covered in brief is Cartesian Genetic Programming (CGP), a widely used EA in the EHW.

## 2.2 Evolutionary Algorithms

EAs are heuristic algorithms which are inspired by and loosely mimic natural evolution and are used to solve numerous real-world problems, such as architectural designs and analogue and digital circuit designs. Evolutionary Strategies, Evolutionary Programming, Genetic Algorithms (GAs) and GP are considered sub-categories of EAs with numerous variant implementations of these algorithms. Generally, EAs have the following characteristics:

- *Population-based*: EAs maintain a group of individuals referred to as a population. Each individual is a chromosome (also known as genotype or genome) representing a candidate solution to the problem being solved. The population evolves over time as variation operators are used to create new individuals to replace less fit individuals—*survival of the fittest*.

- *Fitness Function*: This is a function that assigns a fitness value to each individual in the population, indicating how well it solves the problem at hand. Hence, the fitness function plays a key role in facilitating improvements within the population as it evolves in order to obtain a good candidate solution for the problem at hand.

- *Reproduction*: Recombination (or crossover) and mutation operators are mechanisms used to create new offspring or solutions which inherit traits from their parents to replace less fit individuals in the populations.

- *Selection*: Selection operators constitute the mechanism by which individuals within a population are deemed fit to reproduce. The selection process is driven by the quality of the fitness value. Fitter individuals have a higher probability of being selected for reproduction compared to less fit individuals.

- *Iterative*: EAs repetitively evolve the population infinitely and are only brought to a halt if a termination condition or stopping criteria are defined prior to the execution of the algorithm.

### 2.2.1 Evolutionary Process

A generic evolutionary process contains three phases: initialisation, fitness evaluation, and evolutionary phases, which are summarised below:

1. **Initialisation**: This phase deals with the creation of individuals to constitute the initial population. The algorithms used for the creation of these individuals are usually called *initialisation routines*.

2. **Fitness Evaluation**: In this phase, the quality of each individual or candidate solution is assessed. The assessment of the quality of a solution is done using the defined fitness function. A fitness function is problem-dependent, and its purpose is to evaluate the degree of accuracy of a candidate solution compared to the desired optimal solution. At the end of this phase, if an optimal solution has been obtained, the evolutionary process is terminated or subject to the specified termination criterion.

Following the initialisation phase is the evolutionary phase, which consists of the selection and reproduction processes to create a new population. The period for each evolutionary cycle marks the end of a generation. The generational cycle is made up of a number of steps, which are described below and illustrated in Figure 2.1.

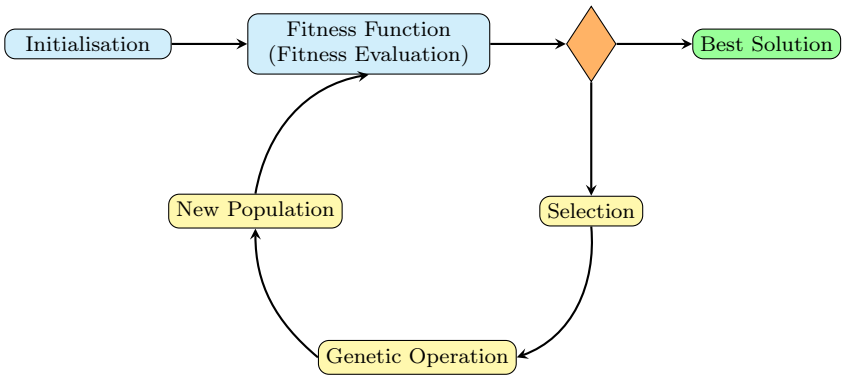


**Figure 2.1:** Standard evolutionary process for EAs.

3. **Selection**: The generational cycle starts off with the selection process. This involves a percentage or a selection of a few individuals from the current population to be chosen by a pre-defined selection algorithm as parents for reproduction. Most selection algorithms are fitness-based, i.e., the fitness of an individual is factored into the selection algorithm.

4. **Reproduction**: The reproduction phase involves a number of steps to create an offspring from the selected parent(s). During reproduction, two parents have portions of their genetic material combined to create offspring.

- **Crossover**: Crossover operators are regarded as global search operators.
- **Mutation**: Mutation algorithms are used to alter the genetic makeup of an offspring by some probability. Mutation operators are regarded as local search operators.

The evolutionary cycle is an infinite loop that keeps evolving generation after generation. The evolutionary cycle halts when a predefined termination condition or criteria has been met. Examples of termination conditions are fitness improvement stalls for a number of generations, the maximum number of fitness evaluations allowed has been reached, the maximum number of generations has elapsed and a drop in population diversity under a pre-specified threshold (Eiben and Smith, 2015).

## 2.3 Introduction to Grammatical Evolution

GE is a type of EA, more specifically, a GP variant used for the automatic generation of computer programs. GE evolves programs in any arbitrary language that is BNF compliant (Ryan et al., 1998). Standard GP uses an abstract syntax tree data structure for the representation of the individuals upon which genetic operators are applied directly. Unlike standard GP, GE uses a linear genome to represent individuals in the genotype space and uses a mapping mechanism and a Context-Free Grammar (CFG) to derive the phenotype in the syntax of the target language. GE's linear genome is made up of a sequence of integers or a bit string. Genetic operations are applied in the genotypic space (on the linear genomes). Illustrated in Figure 2.2 is a high-level overview of GE.

Standard GE consists of four main components: grammar, a mapper, a search engine, and a fitness function. The mapper requires a linear genome which consists of integers referred to as codons or a binary string with a sequence of bits representing a codon. The grammar is usually a subset of the target language designed and deemed sufficient to generate programs capable of producing a satisfactory solution to the problem at hand. The process of decoding the linear genome using the mapper and grammar is referred to as genotype-to-phenotype mapping.

The genotype-to-phenotype mapping process is described in Section 2.3.2. The search engine is an EA used to explore and guide evolution within the search space towards regions of the search space where potential solutions are likely to be found. GA is the most commonly used search engine in GE. The fitness function is used to evaluate and assign a fitness value to each individual, which serves as a measure of how well each individual solves the problem at hand. The individual with the best error or score is returned as the best solution found.

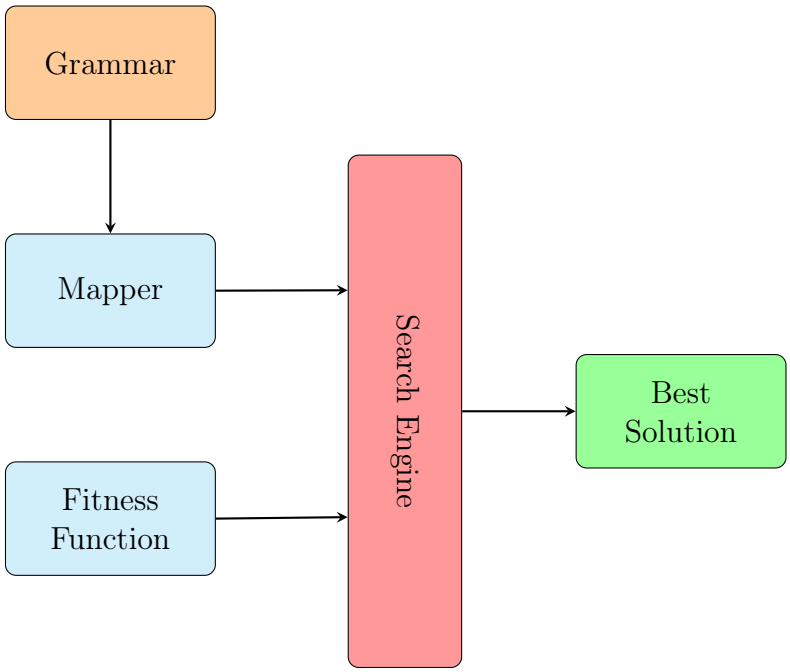


**Figure 2.2:** High-Level Overview of GE.

### 2.3.1 Backus-Naur Form

BNF is a meta-syntax notation used for expressing grammars of formal languages such as CFG. In turn, CFG is the type of grammar most commonly used with GE. A CFG denoted by $G$ is defined by a quadrupled tuple $\{N,T,P,S\}$, where:

1. $N$ represents the set of non-terminal symbols, which are enclosed within <>. For example, $\langle \text{expr} \rangle$ is a non-terminal symbol. Non-terminal symbols can be replaced by a symbol or a collection of symbols referred to as a production rule, which appear after the replacement symbol ::= on the right-hand side.

2. $T$ represents the set of terminal symbols, which are not enclosed within <>. For example, variable names and operators (e.g. +, -) of a programming language are considered terminals.

3. $P$ represents production rules, which are alternatives or choices to replace a non-terminal and are separated by a '|' symbol.

4. $S$ represents the start symbol, which is part of $N$ and serves as the initial symbol from which a legal sentence in the target language can be derived.

Shown in Figure 2.3 is an example of a grammar capable of generating mathematical expressions.

$$
\begin{aligned}
\langle \text{expr} \rangle &::= \langle \text{var} \rangle \mid \langle \text{var} \rangle \langle \text{op} \rangle \langle \text{var} \rangle \mid \langle \text{var} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
\langle \text{var} \rangle &::= \texttt{x} \mid \texttt{y} \mid \texttt{constant} \\
\langle \text{op} \rangle &::= \texttt{+} \mid \texttt{-} \mid \times \mid \texttt{/} \\
\langle \text{constant} \rangle &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

**Figure 2.3:** Example grammar capable of generating arithmetic expressions.

### 2.3.2 Genotype to Phenotype Mapping Mechanism

The GE mapping mechanism requires two main inputs: the linear genome (individual) and a grammar designed for the problem at hand. It starts from a start symbol. Usually, the first rule's left hand side is designated as the start symbol. For example, in Figure 2.3, ⟨expr⟩ is the start symbol. GE iteratively expands or rewrites all non-terminals by using a modulus rule to select a production each time a choice needs to be made. This process continues until a sentence devoid of non-terminal(s) is obtained. In Figure 2.4, using the sample genome provided, the sentence $x + 5$ is obtained.

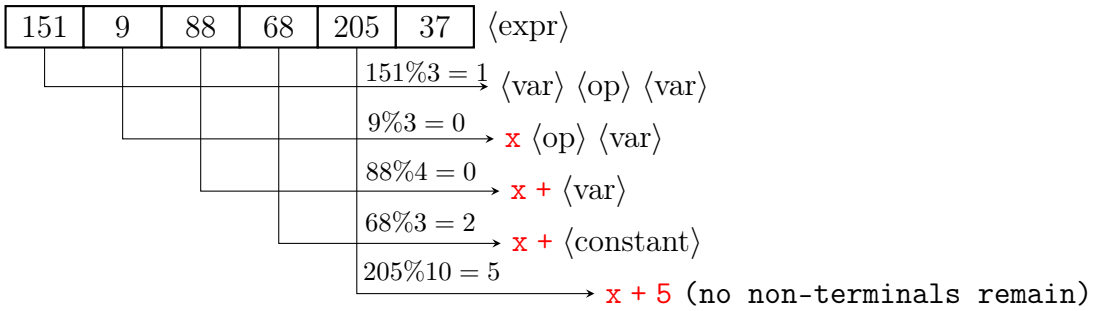


**Figure 2.4:** Grammatical Evolution Mapping Mechanism.

### 2.3.3 Search Engine: Genetic Algorithms

GAs which use either a binary string or a vector of integers as their problem representation are normally used as the search engine for GE. In the case of binary string representation, during the mapping process, 8-bit integer numbers are generated by converting consecutive 8-bit sequences. Other search engines, such as particle swarm optimisation, have been used instead of GE (O'Neill and Brabazon, 2004). A GA's evolutionary process is the same as described in Section 2.2.1. GAs use crossover and mutation operators for exploitation and exploration, respectively.

#### 2.3.3.1 Sensible Initialisation

Up until the introduction of Sensible Initialisation (SI), GE genomes were initialised randomly. That is, a minimum and maximum number of codons were specified, and a number (within the number of codons range specified) of codons were uniformly generated. However, random initialisation has a number of drawbacks (Ryan and Azad, 2003):

- A high percentage of individuals created are too short or too simple.
- A high percentage of individuals fail to map completely. In other words, most individuals do not produce valid sentences (containing non-terminal(s)) in the target language being used.
- Populations are evolved from a small percentage of valid individuals.

To address the above drawbacks, SI was proposed. SI is a GE implementation of the ramped half-and-half initialisation scheme for GP introduced by Koza (1992) for GP. Ramped-half-and-half uses two methods to create trees that form the population: *Grow* and *Full* methods. The Grow method creates trees of irregular shapes and sizes by selecting elements from the function and terminal set with equal probability until the specified maximum depth is reached. A branch terminates once a terminal is selected, even if the maximum depth hasn't been reached. The Full method, on the other hand, continuously selects elements from the function set until the maximum depth is reached before only selecting elements from the terminal set.

Unlike GP, recursive rules play the role of functions. First, SI requires the grammar to be labelled—whether a production rule is recursive or non-recursive and the minimum derivation tree depth to fully expand a production rule. A recursive rule has one or more of these characteristics: any of the choices on the right-hand side of a production rule contains the non-terminal on the left-hand side of the production rule; any of the choices of a production rule contains a non-terminal that leads back to the same non-terminal; any of the choices of a production rule contains a non-terminal that leads to a rule that is labelled recursive based on the two reasons earlier stated (Ryan and Azad, 2003). For example, the production rule ($\langle$expr$\rangle$) in Figure 2.3 is a recursive rule.

SI requires a maximum tree depth to be specified prior. Similarly to ramped half-and-half in GP, SI applies both the Grow and Full methods. When applying the Grow method, any production can be selected, while the full method chooses only recursive productions. Both Grow and Full methods are subject to the constraints of not exceeding the maximum specified tree depth and the availability of enough tree depth budget to expand all non-terminals to terminals fully.

#### 2.3.3.2 Effective Crossover

In GE, randomly selecting crossover points could potentially result in creating duplicate individuals as offspring i.e. same as a parent. This occurs when the chosen crossover points lie outside the effective length of the genome. The effective length refers to the number of codons that were consumed during the

mapping process to obtain a phenotype. Shown in Figure 2.5 are two parent genomes with a sequence of coloured codons highlighting the codons used during the mapping process (the effective length is the number of used codons). For example, in Figure 2.5, the effective length for parent 1 and parent 2 are 11 and 10, respectively. During crossover, if the crossover point randomly chosen falls within the non-coding regions (sequence of unused codons), the same phenotype will be obtained at the end of the mapping process, rendering the child a duplicate individual. The red vertical line through the parent genomes represents the crossover points.

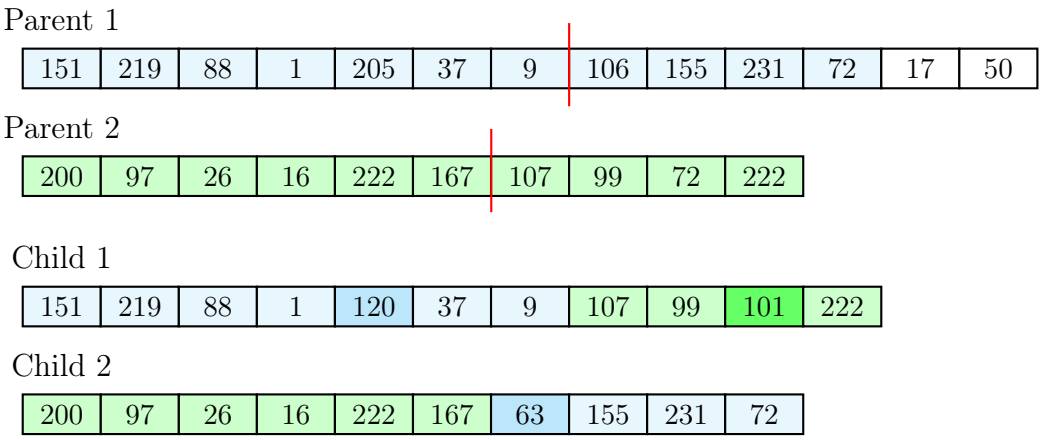


**Figure 2.5:** Illustration of effective crossover and mutation. The coloured sequence of codons represents the effective length.

For example, in Figure 2.5, crossover points for parent 1 are chosen within the [1,11] range to reduce the probability of creating duplicate individuals. The most commonly used crossover operator in GE is the standard single-point crossover, which is depicted in Figure 2.5 and has been shown not to be adversely destructive and good at performing a global search over the course of an experimental run (O'Neill and Ryan, 2000).

#### 2.3.3.3 Wrapping

There are two possible states after the mapping process finishes: *complete* and *incomplete* mapping. A complete mapping state produces a legal sentence in

the target language being used, and therefore, the resultant program can be evaluated. In the case of incomplete mapping, all codons in the genome have been consumed and the result sentence still contains non-terminals, rendering it syntactically incorrect. In order to ensure that a high percentage of individuals are valid to help drive the evolution, a wrapping mechanism is used where the sequence of codons in the genome is reused based on a pre-specified number of times in an attempt to attain complete mapping (Ryan et al., 1998). If there still exists non-terminals in the resultant program, the mapping process is terminated. This is because in some cases the mapper is stuck in an infinite loop as it is continuously selecting the same production rules without reaching a terminal state.

## 2.4 Why is GE suited for Digital Circuit Design Tasks?

GE has been applied to many problems in different domains. However, despite its widespread adoption in many domains, digital circuit design hasn't received much attention from GE.

In industry, due to the complexity of circuits being designed, hardware engineers use HDLs such as Verilog/SystemVerilog and VHDL. HDL circuits designed at a high level of abstraction are interpretable and can be synthesised to obtain equivalent gate-level representations.

The very few existing works that applied GE to digital circuit design tasks merely investigated the feasibility of using GE to evolve complex circuits at higher levels of abstraction through the exploitation of the expressive programming constructs in HDLs (Karpuzcu, 2005) than at the gate level, which is not scalable (Cullen, 2008). However, the availability of good logic simulators makes GE an ideal tool for circuit design tasks.

## 2.5 Cartesian Genetic Programming

CGP is widely used in the evolutionary design of circuits. CGP is a GP variant which uses directed acyclic graphs to represent its programs instead of trees in conventional GP. CGP uses different types of genes in its genotype. A *function gene* encodes the address to a function in a lookup table a particular node must perform, *connection genes* encode addresses of the valid input sources for a node, usually in a feedforward manner (Miller and Thomson, 2000), and *output genes* hold addresses of nodes where program outputs are to be retrieved. CGP generally uses a $(1 + \lambda)$ evolutionary strategy, where $\lambda$ is usually small (often 1 to 4). Thus, one parent produces $\lambda$ offspring via mutation ( no crossover is normally used in standard CGP). The best-performing individual from parent and offspring is selected for the next generation. The mutation operator in standard CGP is usually used to alter a node's input connections, the node's function of a node or the active output of the program.

CGP's representation makes it suitable for the low-level evolution of circuits. Usually, the final output representation of CGP-evolved circuits is in the form of Boolean logic. CGP has been used to evolve sequential circuits (see Section 5.2) and combinational circuits, such as adders and multipliers of small and relatively complex input sizes (Sekanina et al., 2011), and seven-segment display decoders.

## 2.6 Summary

A brief introduction to EAs and a concise introduction to GE were presented in this chapter, covering the main components of GE, which are: grammar, mapping mechanism, initialisation (SI), wrapping (a mechanism to reduce the number of invalid individuals generated), effective mutation and crossover, and GAs, the GE's search engine. Also discussed were the reasons why GE may be a useful tool for circuit design.

# 3

# Evolution of Digital Circuits

## 3.1 Introduction

This chapter provides a background of EHW, a field under which this body of work falls. The importance of EHW, its challenges and the proposed solutions to some of these challenges from relevant literature are presented. Furthermore, the chapter discusses the limitations of current approaches and why evolving circuits at a higher level of abstraction other than gate-level using GE and an HDL is a plausible solution to evolving complex digital circuits.

## 3.2 Evolvable Hardware

Evolutionary design of circuits is a sub-category of EHW which deals with the use of EAs to evolve circuits—analogue and digital circuits. This study is restricted to digital circuits only as it is the focus of this work. EHW is an AI sub-field that deals with the application of EAs to autonomously adapt the behaviour or reconfigure the architecture of electronic hardware as it interacts with its environment (Adaptive Hardware or Evolving Hardware), the evolutionary design of conventional circuits (behavioural requirements are known upfront), and the evolutionary optimisation of circuits.

The idea of applying EAs (GA, GP, etc.) to evolve electronic circuits in a special hardware system called Darwin Machines (DMs) was conceived by de Garis

(1991). Two years later, de Garis observed software configurable hardware chips such as Field-Programmable Gate Arrays (FPGAs) were on the market. A DM was defined as a device capable of repeated self-reconfiguration using a configuration bit string. The configuration bit string maps to either the whole or a subpart of an FPGA device architecture.

The suggested DM architecture consisted of a single master circuit and a population of slave circuits. The master circuit behaves like a GA. The master circuit sends each slave circuit two bit-string instructions. The first is a GA chromosome consisting of the configuration of a circuit. The second bit-string instruction is used to configure another circuit, which measures the fitness of the designed circuit and reports the fitness value of the evolved circuits to the master. The master circuit uses this fitness information reported by the slave circuits to progress evolution into the next generation. Two approaches to DM design were proposed: the direct evolution of circuits in FPGA and the use of hardware accelerators. Hardware accelerators are specifically designed hardware for software simulation of circuits containing a maximum of several hundred thousand logic gates (de Garis, 1993).

Higuchi et al. reported the first experiment towards the goal of building a Darwin machine in (Higuchi et al., 1993). A 6-*bit* multiplexer was evolved using a fixed length GA (108 bits). The 108-bit chromosome was a representation that programmed a part of the CMOS-based programmable logic array (PLA), with a reprogramming time of 50 milliseconds, called GAL16Z8 chip (Higuchi et al., 1993). However, the simulation of the circuits was done on the GAL simulator because GAL did not guarantee an infinite number of reprogramming. Essentially, the entire experiment to evolve the multiplexer was purely software-based before the final design was committed to the GAL16Z8 chip.

Several other circuits have been evolved in the literature to date after the evolution of the 6-bit multiplexer by EHW pioneers. Other works have also focused on proposing new genetic representations, genetic operators, genetic algorithms, efficient simulation of circuits, etc., to increase evolvability, evolve compact circuit designs, and reduce evolution time. Despite these considerable feats, the EHW field is confronted with several challenges that remain a major focus of the EHW research domain.

## 3.3 Modes of Circuit Evaluation for Evolvable Hardware

The mode by which circuits are tested or evaluated is a very important component in the flow of the evolutionary design of digital circuits. As circuit complexity increases, the amount of testing required increases exponentially, and the population-based nature of EAs makes it very time-consuming.

At present, there exist three modes of circuit evolution in EHW. These are Intrinsic, Extrinsic, and Mixtrinsic evolution. In Intrinsic Evolution, candidate circuits are directly evaluated on target hardware such as FPGAs (Thompson, 1997). Extrinsic evolution is a software-based evaluation of circuits using simulators, such as Icarus Verilog (Williams and Baxter, 2002) and Quartus[1], before being committed to hardware. In Mixtrinsic Evolution, a population of candidate circuits is divided in two, with each evaluated using either a simulator or an FPGA, for example, but not both (Stoica et al., 2000).

## 3.4 Importance of Evolvable Hardware

The ultimate objective of EHW is developing next-generation hardware capable of self-reconfiguration through evolution (Sekanina, 2006; Yao and Higuchi, 1997). However, a major and basic requirement of an adaptive hardware system is the ability to evolve its circuitry. A major reason for the extreme importance of adaptive systems is their suitability for use in environments characterised by extreme conditions (such as extreme temperatures, high radiations, etc.), which are unsafe and inaccessible for humans to operate. A typical application area is space exploration (Keymeulen et al., 2006; Lohn et al., 2000; Zebulum et al., 2005). Furthermore, the choice of EAs for circuit evolution has been demonstrated to be better and more creative at exploring the circuit design search space than humans (Thompson, 1996). Also, the use of many-objective or multi-objective EA algorithms can be used to potentially evolve circuits with other desirable

[1] `https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html`

properties in addition to functionality, such as lower power consumption and small circuit area (Hrbacek et al., 2016).

## 3.5 Evolvable Hardware Challenges

In spite of the immense progress made in the EHW field since its inception, the discontinuation of FPGA bitstream documentation and issues of scalability have long hampered its progress. A bitstream is a file that contains all the configuration information for an FPGA. The Xilinx XC6200 (Hartenstein et al., 1998), which was the first commercially available FPGA with a well-documented bitstream, was a major boost for the EHW field at its early stages. This meant the bitstream of these FPGAs could be manipulated or evolved using bio-inspired algorithms such as GAs to design various circuits. Most notable in the field was the evolution of a tone discriminator using the beta-test version of Xilinx XC6216 via direct bitstream evolution (Thompson, 1997). However, FPGA manufacturers refrained from publicly making available the documentation of FPGAs due to a number of reasons, such as security concerns, intellectual property protection, and support and revenue business model.

With regards to the issue of scalability, the two major issues are Scalability of Representation and Scalability of Fitness Evaluation (Zdenek, 2018). As the number of circuit inputs increases, longer chromosomes are required to represent these circuits. This results in large search spaces, which are difficult, slow and expensive to search (Zdenek, 2018), reducing the chances of EAs finding novel circuit designs. Furthermore, due to the destructive nature of genetic operators, evolving circuits of such complexity become intractable. This is termed as *Scalability of Representation*. Also, increasing circuit inputs increases the number of test cases (Stomeo et al., 2006*a*). As a result, exhaustive testing is only feasible for modestly complex circuits —circuits with a small number of inputs and a reasonable number of gates that can be exhaustively tested. This phenomenon is termed Scalability of Fitness Evaluation.

## 3.6 Proposed Solutions to Evolvable Hardware Challenges

This section discusses solutions to EHW challenges discussed in Section 3.5. These are covered in two main sections: Project IceStorm, a solution to Intrinsic Evolution, and Tackling EHW Scalability Issues, which discusses proposed solutions to evolving more complex circuits.

### 3.6.1 Project IceStorm

Project IceStorm is an open-source project aimed at providing bitstream format documentation of iCE40 FPGAs as well as the requisite tools for the creation and analysis of the bitstream files (Wolf and Lasser, 2015). However, Project IceStorm, though not originally conceived as a solution to intrinsic evolution (direct evaluation of candidate circuits on the target hardware, such as an FPGA during evolution) in EHW, has become a solution. For example, Fritzsch et al. (2022) replicated the tone discriminator experiment (Thompson, 1997) using a Lattice iCE40 FPGA with an optimised evolution environment.

### 3.6.2 Tackling EHW Scalability Issues

Research conducted to address scalability issues in the evolutionary design of digital circuits has followed three main trajectories: Functional-Level Evolution, Increasing Evolvability Through the Improvement of Genetic Operators and Problem Decomposition. Research under these trajectories is discussed in separate sections below.

#### 3.6.2.1 Functional Level Evolution

Functional-Level Evolution (FLE) uses secondary logic functions such as adders, multiplexers and multipliers (usually in conjunction with primitive logic gates) other than only primitive logic gates in designing digital circuits (Murakawa et al., 1996).

In (Kalganova, 2000*b*), half-adder, full-adder and multiplier circuits were used as functions to design adders and multipliers. In comparison to gate-level evolution, function-level evolution obtained higher success rates as well as significant reductions in the number of generations in some instances (Kalganova, 2000*b*). Three-bit multipliers were evolved using binary multiplexers in (Vassilev and Miller, 2000), and a proposed Constrained CGP evolved higher-order approximate multipliers using imprecise lower-order multipliers (Senthilkumar et al., 2020). Approximate 9-input and 25-input median circuits (Vasicek and Sekanina, 2015), and an image filter (Slaný and Sekanina, 2007) have also been evolved at the functional level.

#### 3.6.2.2 Increasing Evolvability Through The Improvement of Genetic Operators

Evolutionary performance is largely dependent on the ability of genetic operators to facilitate the evolution of the population through the creation of fitter offspring as evolution progresses. The traditional CGP setup uses point mutation as its variation operator. This incurs wasted evaluations and stalls evolution when point mutation operations affect only inactive genes (genes which do not code any part of the phenotype). Single Active Mutation (SAM), unlike point mutation, ensures an active gene is mutated whenever mutation takes place (Manfrini et al., 2016).

Biased SAM is an improvement upon SAM, which increases its robustness when tackling combinational circuit designs. Biased SAM requires a transition probability matrix to work with. This is constructed for each problem by conducting preliminary experiments on that problem. During these initial experiments, functional mutations that increase the overall fitness of an individual are recorded. Subsequently, during the actual experiments, roulette wheel selection is applied on the transition probability matrix to determine the function the chosen functional node should be mutated to (Manfrini et al., 2016). On selected circuit design benchmarks, Biased SAM outperformed SAM.

Guided Active Mutation (GAM), another CGP mutation operator, was designed to mutate active node(s) in a subgraph corresponding to a single output (da Silva et al., 2019). The objective was to increase the overall fitness score of this output when compared to the truth table of the circuit. However, though

GAM recorded fewer fitness evaluations to find feasible solutions, it recorded lower success rates. As a result, SAM and GAM were merged in order to harness the strengths of both mutation operators. Their combination resulted in high success rates and best results on the selected benchmark problems (da Silva et al., 2019).

Semantically-oriented mutation operator (SOMO) performs mutation in the phenotypic space, after which the resultant phenotype is encoded to its corresponding genotype (Hodan et al., 2020). SOMO randomly selects an active node for mutation. A mutation operation may affect a function node or an input node connection. During a node input connection mutation, the best connection point that results in an increase in an individual's fitness is determined through the application of semantics (Hodan et al., 2020). SOMO evolved combinational circuits significantly faster compared to the multi-threaded parallel CGP implementation in (Hrbacek and Sekanina, 2014).

#### 3.6.2.3 Circuit Decomposition Methodologies

Several methodologies have been proposed to decompose complex circuit problems into subcircuits of evolvable complexity. These fall into two main categories: *inputs* and *outputs* decomposition. The former refers to decomposition methods that break down circuits into subcircuits using only the circuit's inputs, while the latter uses the circuit's outputs.

Increased complexity evolution (ICE), also referred to as Divide and Conquer methodology, breaks a complex system into sub-systems of evolvable complexity before the evolution is performed on each sub-system in a bottom-up manner (Torresen, 1998, 2001). Sub-systems can be evolved in parallel (if no dependency exists between them) or sequentially. These evolved sub-systems serve as building blocks in a more complex sub-system(s) in an upper layer. One challenge with ICE is that it requires the fitness functions for the respective sub-systems to be manually defined. However, for certain problems such as circuit design, a solution to this challenge is the partitioning of training vectors based on the circuit outputs (Torresen, 1998, 2001).

In contrast to ICE, Bidirectional Incremental Evolution (BIE) performs the automatic decomposition of a complex circuit into subsystems using either a suitable standard decomposition technique (such as Shannon's theorem) or an EHW-oriented decomposition technique or a combination of both (Kalganova, 2000*a*). BIE progressively decomposes complex circuits into subcircuits in a top-down manner while evolving each subsystem. Subsystem(s) undergo further decomposition only if evolution is unable to obtain optimal solutions. During the second phase of BIE, evolved subsystems are merged and optimised in a bottom-up fashion. In comparison to direct evolution, BIE performed best on 7-input and 10-output circuit benchmark problems.

On more complex circuits, BIE performed poorly (Stomeo et al., 2006*a*). As a result, instead of output decomposition in the case of BIE, an input decomposition technique termed Generalised Disjunction Decomposition (GDD) was proposed in (Stomeo et al., 2006*a*). GDD automatically decomposes a complex circuit into two sub-circuits: the evolvable and multiplexer subcircuits (Stomeo et al., 2006*a*). GDD requires the user to specify the number of inputs for the evolvable circuit, subject to the constraint that it must be less than the number of inputs of the original circuit. GDD then applies an algorithm to generate the evolvable sub-circuit's outputs using the actual circuit's truth table. The evolvable sub-circuit can be evolved using any EHW method (such as IEC or BIE) without any restrictions. The multiplexer sub-circuit takes the outputs of the evolvable sub-circuit as inputs and uses the remaining inputs of the actual circuit not used in the evolvable circuit as select lines for the multiplexer. The output of the multiplexer circuit is the same as that of the complete (original) circuit. Using a selection of circuit benchmark problems from the Microelectronics Center of North Carolina (MCNC) benchmark suite and randomly generated circuit problems, compared to BIE, GDD required fewer generations and less time and achieved better fitness scores (Stomeo et al., 2006*a*).

Stepwise Dimension Reduction (SDR) is a layered and output decomposition methodology that decomposes a circuit into two sub-circuits. One of these is evolved to handle input combinations with an output value of 1, while the second is evolved to handle those with 0 as output (Li et al., 2008). Internal intermediate mappings are devised if sub-circuits are not of evolvable complexity. SDR evolved

a 19-bit circuit, which GDD was unable to evolve. SDR obtained comparable results for a few of the benchmark problems in less time compared to GDD, but not on more complex multiple output circuit benchmarks (Li et al., 2008). SDR is effective on single-output circuit problems such as the parity circuit.

Common to all these approaches is that they are capable of evolving relatively complex circuits as it is feasible to first evolve sub-circuits before merging them into a complete circuit with a high-dimensional input.

#### 3.6.2.4 Other Scalability Approaches

In other works such as (Hrbacek and Sekanina, 2014), multi-threaded parallel implementations of evolutionary algorithms, specifically CGP, have been developed to exploit modern processor architectures to allow for more fitness evaluations. However, just a single run was performed to evolve a 5-bit $\times$ 5-bit multiplier due to the associated high computational effort. In (Vasicek and Sekanina, 2014), CGP candidate circuits are transformed into a Binary Decision Diagram (BDD). Similarly, the circuit is also specified in BDD format. A BDD evaluation is performed, where functional similarity is performed between the candidate circuit's BDD representation and the circuit's specification also in BDD format. The fitness of the candidate circuit is the hamming distance between the output vectors of the BDD representations of the candidate circuit and the circuit specification. The results reveal that BDD-based evaluation is faster than exhaustive testing.

## 3.7 Evolving Digital Circuits at a High Level of Abstraction

This is not the first work proposing the evolution of digital circuits in an HDL using an EA. One of the earliest works, if not the first, that explored the evolution of circuits in HDL is (Mizoguchi et al., 1994). There, a GA augmented with a rewriting system which uses a grammar to design digital circuits called Production Genetic Algorithm was used to evolve a circuit for an ant to solve the John Muir Trail problem. However, this work was aimed at demonstrating an alternative approach to designing circuits behaviourally. (Karpuzcu, 2005)

used GE to evolve a 1-bit adder in Verilog. However, a 1-bit adder is a trivial problem and did not require the use of a lot of Verilog's expressive programming constructs. Another work also evolved circuits using GE at the gate-level (Cullen, 2008). A lot of current approaches evolve circuits at the gate level. The issue with gate-level evolution is its inability to scale to complex circuits, especially from scratch (Vassilev and Miller, 2000).

At the early stages of EHW, the commercialisation and public provision of the detailed documentation of the Xilinx XC6200 family of FPGAs was a major boost. In this modern era, FPGA use cases require more powerful FPGAs than the early FPGAs that consisted of tens of thousands or hundreds of thousands of logic cells such as Xilinx XC6200. The discontinuation of the XC6200 family of FPGAs stalled the progress of EHW, especially EHW research that employed intrinsic evolution of circuits (Thompson, 1997). Given that FPGA complexity has greatly increased over the decades to meet the needs of modern real-world applications, direct bitstream evolution of FPGAs such as iCE40 FGPAs for such use cases will be very challenging as the chromosomes used to represent the bitstreams will be too long to evolve any satisfactory solution (also known as scalability of representation).

Furthermore, there is also the issue of portability of FPGA bitstream configuration across different FPGA families or manufacturers. This is because there exists no standard for FPGA architecture that FPGA manufacturers must adhere to, coupled with security reasons and intellectual property protection. Hence, an evolved bitstream solution may not be directly and easily ported to a different FPGA other than the one used for evolution. Last but not least, HDL-evolved solutions can easily be synthesised for different FPGA families if needed.

## 3.8 Summary

Presented in this chapter was a background to EHW, the importance of EHW, its challenges and existing solutions. The chapter also provided evidence as to why the use of GE and HDL will facilitate the evolution of complex digital circuits and serve as a plausible solution to EHW scalability challenges.

In the next chapter, the evolutionary framework for the functional evolution of circuits called ADDC is presented.

# 4

# Automatic Design of Digital Circuits

## 4.1 Introduction

In Chapter 2, a brief introduction to EAs, a concise overview of GE, and why it is an ideal tool for circuit design tasks were discussed. In this chapter, the evolutionary framework for the functional evolution of digital circuits using GE, SystemVerilog and a circuit simulator are presented. Also briefly covered is the standard CAD flow for digital circuits, highlighting the phase of the CAD flow this research fits in.

The results in this chapter were published as a book chapter titled "ADDC: Automatic Design of Digital Circuits" (Ryan et al., 2022).

The aims of this chapter are:

- Provide an introduction to HDLs (specifically SystemVerilog) and why they are the de facto standard for digital circuit design in the industry;
- Provide a brief overview of standard CAD flow for digital circuit design and where the body of work presented in this thesis fits in the flow;
- Present a detailed description of the proposed evolutionary framework called ADDC for the evolution of digital circuits;

- Test and assess the performance of ADDC on combinational circuit benchmark problems.

## 4.2 Overview

There are two main types of digital circuits, namely, combinational and sequential. Combinational circuits are a class of circuits whose output(s) depend(s) solely on current inputs. Sequential circuits, on the other hand, are a class of circuits whose output(s) depend(s) on both current and past input(s). Sequential circuits consist of memory elements such as flip-flops that store combinational logic's output, which are then fed back as input.

The use of schematic diagrams is an alternative approach to circuit design, alongside the use of HDLs. Schematic diagrams involve the use of graphical representations of circuit components to design circuits. Schematic diagrams reveal the functionality and connections between the various components used to realise the circuit functionality.

HDLs are well suited and ideal for large and complex circuit designs compared to schematic diagrams (Coffman, 1999; LaMeres, 2019). The use of HDLs is the *de facto* standard in the design of digital circuits in industry due to a number of reasons: first, real-world digital circuit designs are usually intractable to design at the gate level; second, more abstract designs are more interpretable, require relatively less time to vary/adapt and are easier to verify (Sutherland, 2017); third, they enable design module reuse through parameterisation and support for mixed-style design in a single module; fourth, HDL designs are portable across different hardware because they can be synthesised for several vendor-specific FPGAs as discussed in Section 3.7.

## 4.3 SystemVerilog

SystemVerilog is one of the major HDLs used in the industry for the design of digital circuits. SystemVerilog is an extension of Verilog, essentially a superset of Verilog. The extension was based on two objectives: efficient and more accurate modelling of digital circuits functionality; writing of efficient and race-free

verification code for large and complex designs (Sutherland, 2017). Digital circuit design in SystemVerilog can be performed at four main abstraction levels, in decreasing order of abstraction: *behavioural level*, *register-transfer level* (RTL), *gate level* and *digital switch level* (Sutherland, 2017). The higher the abstraction level, the less detail it reveals about the actual circuit representation.

Digital switch level deals with the design of circuits at the transistor level, while gate-level modelling involves the use of only primitive logic gates such as: `and`, `or`, `nor`, `xor`, `not`, etc., to design circuits. RTL designs make use of two primary constructs: `continuous assignments` and `always` procedural blocks (Sutherland, 2017). A statement preceded by the `assign` keyword is a continuous assignment. For example, `assign out = inA & inB;`. The output signal (`out`) is continuously driven by the right-hand expression whenever any of the operands or signals change, triggering a re-evaluation of the right-hand expression. RTL designs obscure circuit functionality realisation in silicon, which is only revealed after synthesis (Sutherland, 2017). In comparison to gate-level designs, RTL designs are interpretable, more capable of dealing with huge input sizes, better suited for verification of large and complex designs (Sutherland, 2017), etc. Behavioural level models circuits at a higher abstraction level than RTL, utilising all constructs in SystemVerilog, and therefore may not be synthesisable (constructs that cannot be translated to a gate-level description by the synthesis tool).

### 4.3.1 SystemVerilog Instructions

An `always` procedural block can be used to model combinational and sequential logic (Stuart Sutherland, 2006*a*). An `always` block behaves like an infinite loop that continuously executes statements within its block, except it is triggered either by a time or event control. With time control, the `always` block repeatedly executes each time the specified delay time elapses. For example, an `always` block that toggles a clock (clk) signal after every ten-time units will be described in SystemVerilog as `always #10 clk = ~clk;`.

A clock signal is an oscillating signal between a high and low state, which is used in digital circuits to ensure that the execution of functions/operations

is synchronised. There are two ways a digital circuit can be edge-triggered by a clock: positive and negative edge triggering. A digital circuit is positive-edge triggered when the clock transitions from a low state to a high state. On the other hand, a digital circuit is negative-edge triggered when the clock transitions from a high to a low state. In Verilog/SystemVerilog, positive and negative edge triggering of the clock are specified using `posedge clk` and `negedge clk`, respectively. Note, the time delay syntax is `#`⟨time-unit⟩. An always block with event control requires a sensitivity list. A sensitivity list is a list of signals that trigger the execution of the `always` block whenever any of the signal changes, for example, `always @(signalA, signalB) begin` ⟨stmts⟩ `end` and `always @(posedge clk) begin` ⟨stmts⟩ `end` for combinational and sequential logic modelling respectively. The syntax for sensitivity list specification is: `@(`⟨sensitivity-list⟩`)`. SystemVerilog supports another specialisation of the `always` block for specific use cases (Stuart Sutherland, 2006*a*).

SystemVerilog has various categories of operators: bitwise, reduction, conditional, arithmetic, logical operators, etc. Another very useful SystemVerilog programming construct that we make use of in this work is `generate for-loop` (synthesisable `for-loop`), which is useful for the description of circuits with a fixed and repetitive structure. The `generate for-loop` instruction creates $n$ module instances, which can be specified through SystemVerilog parameters. SystemVerilog also has `switch-case` and `if-else` constructs, which are either synthesised to multiplexers or priority encoders by synthesis compilers based on mutual exclusivity of the selection items (Stuart Sutherland, 2006*a*). However, there exist modifiers to use in conjunction with `switch-case` and `if-else` statements to reduce ambiguities and ease the task of synthesis compilers (Stuart Sutherland, 2006*b*).

Another very useful and powerful feature of SystemVerilog is circuit module parameterisation. Parameterisation facilitates the definition of circuit parameters that can be customised at circuit instantiation time. In other words, a parameterised circuit module can be instantiated with different parameter configurations to satisfy changing requirements.

## 4.4 Computer Aided Design for Circuits

Most modern complex digital circuits are constructed using some form of HDL. They are then subject to numerous simulation and testing steps to ensure that not only is the circuit functionally correct but also that it is indeed synthesisable. There are many powerful tools, such as Quartus (Intel/Altera) and Vivado (Xilinx), that are used by design experts to perform various tests on their programs before committing them to hardware.

CAD for circuits employs these sorts of tools in an iterative way, as shown in Figure 4.1, where designers repeatedly perform functional simulations to ensure that their circuit works as intended before moving to the synthesis phase, in which they ensure that the circuit is fully synthesisable. The reasons why a circuit may not be synthesisable vary from timing issues (components may simply be too far from each other and thus introduce some unexpected delays) to fitting issues (it may not be possible to create this particular circuit given the real estate available), amongst others.

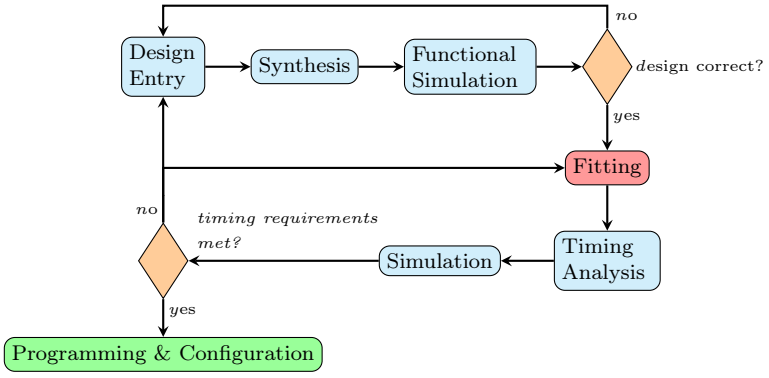


**Figure 4.1:** The standard CAD flow for digital circuit design.

## 4.5 ADDC Framework

ADDC is an evolutionary HDL circuit design framework mainly driven by GE. ADDC requires a grammar and a testbench as inputs for circuit evolution and verification respectively. The designed grammar must be BNF compliant and satisfy the grammar sufficiency property. Thus, the grammar must contain all the

necessary building blocks required to potentially evolve an optimal circuit. A testbench is analogous to regression testing in software programming. A testbench is a verification description written in the same HDL as the circuit that ensures the device under test (DUT) or evolved circuit meets functional and timing requirements. A typical testbench uses a test vector(s) containing circuit inputs and their respective expected outputs. For example, test vector(s) for combinational circuits are usually created from their truth table. ADDC is technology agnostic and easily configurable as the choice of HDL and simulator is left to the user. Illustrated in Figure 4.2 is ADDC's design flow for the functional evolution of circuits.

During the initial phase of the circuit design process, ADDC creates an initial population of circuit designs using a suitable GE initialisation routine, such as sensible initialisation. These individuals then undergo fitness evaluation. The fitness evaluation phase entails a number of steps. First, the genotype (genome) of each individual is translated to an HDL (SystemVerilog in this work) circuit design (phenotype) by the GE mapper, using the grammar designed for the circuit. The functional simulation of each circuit takes place, assuming all circuit designs are valid.

For these experiments, Icarus Verilog is used as the simulator. The simulator uses a testbench, which must be provided by the user for circuit verification. The simulator drives the input(s) specified in the testbench through the DUT and verifies if the circuit output is the desired output. The sum of the number of passed cases out of the total number of cases represents the fitness value of the circuit. After all circuits have been evaluated, ADDC makes the best circuit design available if the specified termination criterion is satisfied; otherwise, the circuit design evolution continues.

The next phase is reproduction, where usually individuals with either good overall fitness scores or individuals that perform best in certain cases are selected for crossover and mutation to create a new population of circuit designs. Lexicase selection performs well on circuit design benchmarks (Ryan. et al., 2020; Tetteh et al., 2021), hence selected as the choice of selection routine. Also, depending on the GA of choice, for example, steady state, generational GAs, etc., events like replacement or elitism may take place in creating the new population. The

new population undergoes fitness evaluation in a similar manner as described in Chapter 2. The process continues until the termination criterion is satisfied and the best circuit design is returned as the solution.

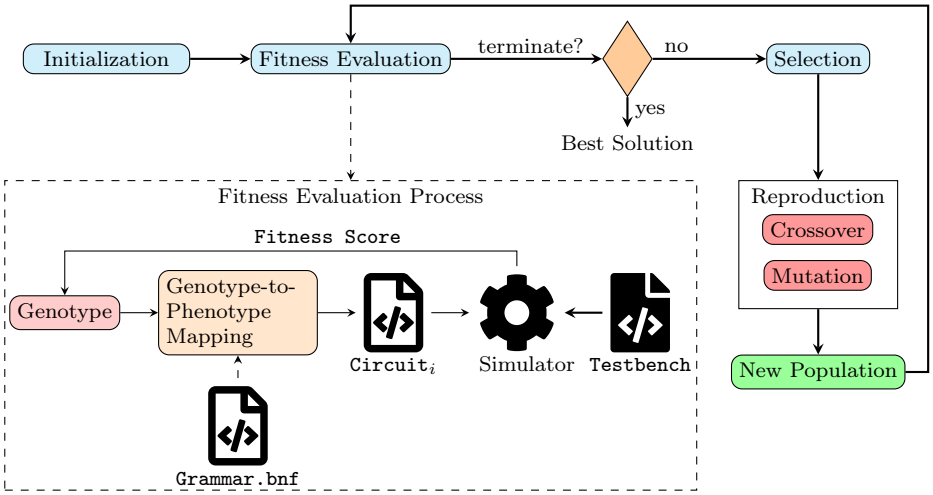


**Figure 4.2:** ADDC Functional Circuit Evolution Overview.

### 4.5.1 Success

It is important to define a metric by which the success of ADDC can be assessed. ADDC is driven by GE. GE's success can be quantified by its ability to find representative solutions to the problem at hand. Beyond circuit functionality, other desirable factors may include circuit size or area, propagation delay and power consumption. However, these qualities are important after obtaining a fully functioning circuit. Hence, the success of ADDC is restricted to its ability to find a viable solution out of a set of independent runs. For complex and time-consuming circuits, attaining any success rate greater than zero may suffice. In extreme cases or in industrial settings where evolving digital circuits may require a lot of resources due to their complexity, obtaining a single solution after a number of independent runs is good enough.

## 4.6 Experiments

Three circuit problems are considered. Two of these are standard benchmark problems: Seven Segment Display (SSD) and 16-to-1 Multiplexer. The third problem is the Hamming Code (7,4) Encoder. All three circuits are used in real-world applications. For example, Hamming Codes, SSDs and Multiplexers are used in satellite communication, digital calculators and telephone networks respectively.

### 4.6.1 Benchmark Problems

#### 4.6.1.1 Hamming Code (7,4) Encoder

Hamming Codes are linear error-correcting codes capable of detecting a single error and, at most, two errors but are only capable of correcting a single error. They belong to a category of codes referred to as Linear Block Codes. A Hamming Code (7,4) Encoder encodes a 4-bit data word into a 7-bit code word prior to data transmission by generating and adding three parity bits to the data word.

The structure of the code words generated by Hamming Codes can be classified into two categories: systematic and non-systematic encodings. The structure of systematic encoding separates the data word and code word, while the data word and parity bits are interspersed in non-systematic encoding. We adopt systematic encoding as it is easier to separate the data word from the parity bits.

The grammar designed for evolving the Hamming Code (7,4) Encoder is shown in Listing 1. The circuit's interface is defined using the ⟨begin-module⟩ rule. The grammar uses four bitwise operators and a bitwise negation operator in Verilog defined using ⟨bitwise-op⟩ and ⟨bitwise-neg⟩ rules, respectively. Hamming Code (7,4) Encoder generates three redundant bits, as stated earlier, and these have been defined using ⟨parity-bit-1⟩, ⟨pa-bit-2⟩ and ⟨parity-bit-3⟩ rules. The ⟨expr⟩ rule is used by the ⟨parity-bit-*⟩ rules to evolve variable length expressions that generate the correct parity bit. The grammar also features an important Verilog construct, the `always` procedural block, defined using ⟨always-block⟩, which behaves similarly to an infinite loop and which is triggered by a change in any signal (indicated with $*$) to evaluate the statements within its scope.

```
 <design-module> ::= <begin-module> <code-block> <end-module>
    <code-block> ::= <declarations> <always-block>
  <declarations> ::= "logic [1:3] parity;" <nextline>
        <always> ::= always @(*) begin <nextline> <parity-bit-1>
                     <parity-bit-2> <parity-bit-3> <output-stmt> end
                     <nextline>
    <parity-bit-1> ::= parity[1] "=" <expr>; <nextline>
    <parity-bit-2> ::= parity[2] "=" <expr>; <nextline>
    <parity-bit-3> ::= parity[3] "=" <expr>; <nextline>
   <output-stmt> ::= codeword "=" {dataword,parity}; <nextline>
          <expr> ::= <input> | <bitwise-neg><input>
                   | <input> <bitwise-op> <input>
                   | <input> <bitwise-op> <expr>
                   | <bitwise-neg>(<expr>)
<dataword-index> ::= 1 | 2 | 3 | 4
    <bitwise-op> ::= & | "|" | ^ | ^~
   <bitwise-neg> ::= ~
         <input> ::= dataword[<dataword-index>]
  <codeword-bit> ::= parity[1] | parity[2] | parity[3]
-------------------------------------------------------------------
  <begin-module> ::= "module hmc_encoder(input [1:4] dataword,
                      output logic [1:7] codeword);" <nextline>
    <end-module> ::= endmodule
      <nextline> ::= "\n"
```

**Listing 1:** Hamming Code (7,4) Encoder Grammar (highlighted in blue is the start symbol).

#### 4.6.1.2 Seven Segment Display

An SSD is an electronic device used for the display of decimal numerals. It is also capable of displaying letters, though some letters, such as `K`, `X`, `Z`, etc., are difficult to recognise on the device. The specification for the SSD considered in this work supports decimal numerals (`0-9`) and `A-F` letter representations. These numbers and characters are encoded as a 4-bit binary number ($0000 - 1111$). Conventionally, 4-bit binary numbers from $0000 - 1001$ (0-9) are referred to as binary coded decimals (BCDs). However, in this work, the binary numbers from $1010 - 1111$ are also referred to as BCDs. The 4-bit binary numbers starting from 0000 to 1001 are used to encode decimal numbers 0 to 9 respectively, while 1010 to 1111 are used to encode letters `A-F`. These BCDs are sent as inputs to the SSD. Upon receipt, the SSD generates a 7-bit binary number (each bit corresponding to an ON/OFF state of a segment) to turn on the appropriate LEDs to display

the digit/letter. Listing 2 shows the grammar designed to evolve the SSD. The BCD and the 7-bit binary numbers are defined by the ⟨bcd⟩ and ⟨seven-segment⟩ rules, respectively. The grammar uses the `switch-case` construct defined using ⟨switch-case⟩ as well as the `always` procedural block (⟨always⟩). ⟨begin-module⟩ defines the circuit interface of the SSD. The `bcd` and seven-segment values were provided to reduce the complexity of the problem solely for testing the ADDC framework.

```
<design-module> ::= <begin-module><next-line><code-block><next-line>
                    <end-module>
   <code-block> ::= <always>
       <always> ::= always @(bcd) begin <next-line><switch-case>
                    <next-line> end <next-line>
  <switch-case> ::= case(bcd) <next-line> <case-stmt> <next-line>
                    endcase
    <case-stmt> ::= <bcd-value> ":" <output> "=" <seven-segment>";"
                  | <bcd-value> ":" <output> "=" <seven-segment>";"
                    <next-line> <case-stmt>
 <default-case> ::= default ":" <output> "=" <seven-segment>";"
    <bcd-value> ::= 4'b0000 | 4'b0001 | 4'b0010 | 4'b0011 | 4'b0100
                  | 4'b0101 | 4'b0110 | 4'b0111 | 4'b1000 | 4'b1001
                  | 4'b1010 | 4'b1011 | 4'b1100 | 4'b1101 | 4'b1110
                  | 4'b1111
<seven-segment> ::= 7'b1111110 | 7'b0110000 | 7'b1101101
                  | 7'b1111001 | 7'b0110011 | 7'b1011011
                  | 7'b1011111 | 7'b1110000 | 7'b1111111
                  | 7'b1111011 | 7'b1110111 | 7'b0011111
                  | 7'b1001110 | 7'b0111101 | 7'b1001111
                  | 7'b1000111
       <output> ::= segment
          <bit> ::= 0 | 1
-------------------------------------------------------------------------
 <begin-module> ::= module "seven_segment_display(output logic [6:0]
                    segment, input logic [3:0] bcd);"
   <end-module> ::= endmodule
    <next-line> ::= "\n"
```

**Listing 2:** Seven Segment Display Grammar (highlighted in blue is the start symbol).

#### 4.6.1.3 16-to-1 Multiplexer

The multiplexer problem is a classic GP benchmark problem. A multiplexer is a multiple-input single-output device that accepts data (data lines) and an address (select lines) as inputs and uses the address to select the corresponding data line to be transmitted. The 16-to-1 multiplexer has 16 data lines and 4 select lines.

Listing 3 shows the grammar designed to evolve the multiplexer. Similarly to the SSD Grammar, the 16-to-1 Multiplexer Grammar also uses the *always* procedural block. However, here, an `if-else` (⟨if-else⟩) construct is used, although the `switch-case` construct is also suitable in this context. The addresses used to select a data line as output are defined using the ⟨address⟩ rule. This rule is used to generate a conditional statement (⟨cond⟩) by the ⟨if-else⟩ rule to determine the data line to select. The ⟨data-index⟩ defines the indexes of the 16-bit data, which is used by the ⟨data-bit rule⟩ to select the data line. ⟨begin-module⟩ defines the circuit interface of the 16-to-1 Multiplexer.

```
<design-module> ::= <begin-module><nextline><code-block><end-module>
                    <nextline>
   <code-block> ::= <always> <stmt>
       <always> ::= "always@(*)" <nextline>
         <stmt> ::= <if-else> <nextline>
      <if-else> ::= if(<cond>) <expr><nextline> else <expr>
                  | if(<cond>) <expr><nextline><else-if> else <expr>
      <else-if> ::= "else if"(<cond>) <expr><nextline>
                  | "else if"(<cond>) <expr><nextline> <else-if>
         <cond> ::= addr "==" <address>
         <expr> ::= " out = " <data-bit>";"
      <address> ::= 4'b0000 | 4'b0001 | 4'b0010 | 4'b0011 | 4'b0100
                  | 4'b0101 | 4'b0110 | 4'b0111 | 4'b1000 | 4'b1001
                  | 4'b1010 | 4'b1011 | 4'b1100 | 4'b1101 | 4'b1110
                  | 4'b1111
     <data-bit> ::= data[<data-index>]
   <data-index> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
                  | 13 | 14 | 15 | 16
-------------------------------------------------------------------------
  <begin-module> ::= "module mux(output logic out, input logic [1:4]
                      addr, input logic [1:16] data);"
    <end-module> ::= "endmodule"
      <nextline> ::= "\n"
```

**Listing 3:** 16-to-1 Multiplexer Grammar (highlighted in blue is the start symbol).

### 4.6.2 Evolutionary Parameters

The experimental parameters used for running the experiments are shown in Table 4.1. These parameters were chosen based on preliminary experiments conducted using all three circuit benchmark problems. 50 generations were used for evolving the Hamming Code (7,4) Encoder, while 100 generations were used for each of the SSD and 16-to-1 Multiplexer designs as preliminary results revealed these problems were relatively challenging to evolve compared to the Hamming Code (7,4) Encoder. All other parameters remain the same for all benchmark problems.

**Table 4.1:** Experimental Run Parameters.

| Parameter | Value |
|---|---|
| Initialisation | Sensible Initialisation (detailed description in Section 2.3.3) |
| No of generations | 50 for Hamming Code (7,4) Encoder,<br>100 for Seven Segment Display &<br>16-to-1 Multiplexer |
| Mutation rate | 0.01 |
| Crossover rate | 0.8 |
| Replacement rate | 0.5 |
| No of independent runs | 50 |
| Population | 1,000 |
| Selection | Lexicase Parent Selection |

### 4.6.3 Training and Testing

The number of training and testing cases are tabulated in Table 4.2.

**Table 4.2:** Number of Training and Testing Cases.

| Benchmark Problem | No of Training Cases | No of Testing Cases |
|---|---|---|
| Hamming Code (7,4) Encoder | 112 | — |
| Seven Segment Display<br>(*with A-F letter representations*) | 16 | — |
| 16-to-1 Multiplexer | 4,100 | 5,000 |

Each of the Hamming Code (7,4) Encoder and SSD have only 16 cases. However, for the Hamming Code (7,4) Encoder, every correct bit in each bit position in the codeword is counted as part of the total fitness score for a candidate circuit, giving a total of 112 ($7 \times 16$) cases. Given that Hamming Code (7,4) and SSD have so few cases, it is feasible to perform exhaustive testing during training, leaving no cases for testing as shown in Table 4.2.

On the other hand, the 16-to-1 Multiplexer has $2^{20}$ cases making exhaustive testing infeasible. As a result, we uniformly sample 4,100 and 5,000 cases for training and testing, respectively, as shown in Table 4.2. $2^{12}$ (4,096) cases were uniformly sampled in addition to the 4 corner cases, making it a total of 4,100 cases. The total number of training cases was rounded up to the nearest thousand, which is 5,000 and was used as the total testing cases.

## 4.7 Results and Discussion

The evolutionary performance for the experiments conducted for Hamming Code (7,4) Encoder, SSD and 16-to-1 Multiplexer described in Section 4.6.1 are visualised in Figures 4.3, 4.4 and 4.5 respectively. The success rate per benchmark problem is tabulated in Table 4.3.

### 4.7.1 Success Rate

A successful run is a single independent evolutionary run that evolved a fully functional circuit for the target problem. Fifty independent runs were conducted for all three benchmark problems. The success rate is the number of successful runs divided by the total number of evolutionary runs (i.e. 50) as tabulated in Table 4.3. A 100% success rate was attained for the Hamming Code (7,4) Encoder. The SSD and 16-to-1 Multiplexer obtained 60% and 86% success rates, respectively.

### 4.7.2 Evolutionary Performance

Visualisations of the evolutionary performance, as evolution progressed for Hamming Code (7,4) Encoder, SSD and 16-to-1 Multiplexer, are shown in Figures 4.3, 4.4

**Table 4.3:** Success Rate for Benchmark Problems.

| Benchmark Problem | Success Rate |
|---|---|
| Hamming Code (7,4) Encoder | $^{50}/_{50}$ |
| Seven Segment Display (*with A-F letter representations*) | $^{30}/_{50}$ |
| 16-to-1 Multiplexer | $^{43}/_{50}$ |

and 4.5 respectively.

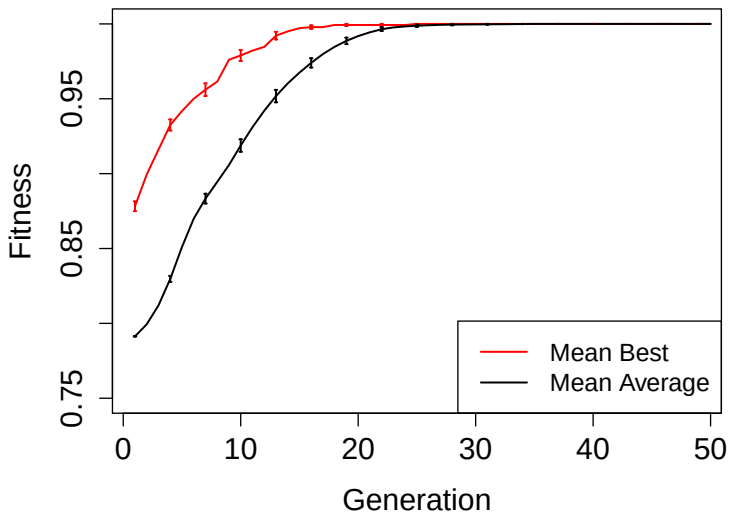


**Figure 4.3:** Mean best and mean average across runs for Hamming Code (7,4) Encoder.

The red line represents the mean best fitness per generation across the 50 independent runs conducted, while the black line represents the mean average fitness. Also plotted are error bars representing the standard error. The error bars are short to non-existent, indicating small variability between the fitnesses of individuals. Furthermore, all three plots reveal a steady and progressive increase in fitness as the evolution progressed, indicating the evolutionary search is continuously searching regions of the solutions where fitter individuals are located. As for the Hamming Code (7,4) Encoder and 16-to-1 Multiplexer problems, GE

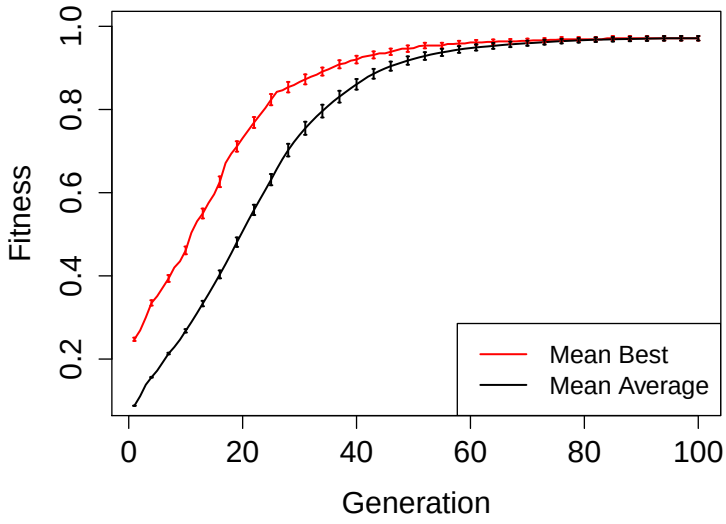


**Figure 4.4:** Mean best and mean average across runs for Seven Segment Display.

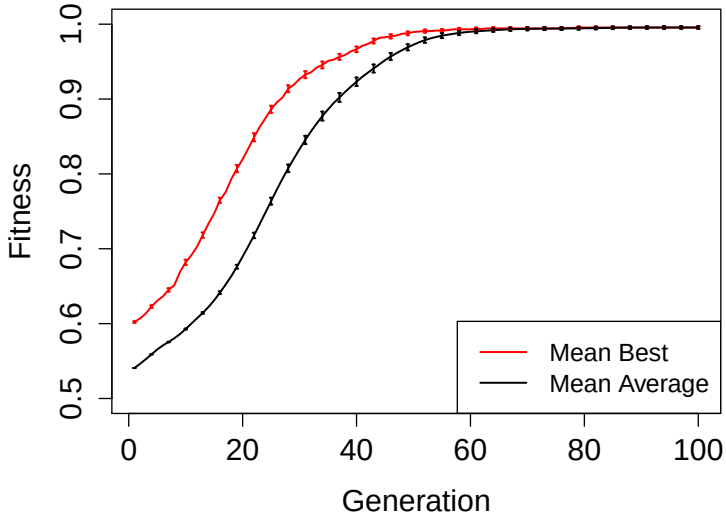


**Figure 4.5:** Mean best and mean average across runs for 16-to-1 Multiplexer.

discovers individual(s) that solve more than 50% of the test cases from the initial generations, while for the SSD problem, it evolves individual(s) that solve 25%

of the test cases on average.

## 4.8 Summary

This chapter presented ADDC—the evolutionary framework for the functional evolution of digital circuits. ADDC is highly configurable; it can support other HDLs and simulators, such as VHDL and ModelSim respectively. A brief overview of the standard CAD flow, an introduction to SystemVerilog and the components and design flow of ADDC were presented. ADDC has been demonstrated on three difficult, real-world problems and was successful on all three of them, including one with $2^{20}$ possible inputs.

# 5

# Behavioural Modelling of Digital Circuits in SystemVerilog Using Grammatical Evolution

## 5.1 Introduction

In the previous chapter, a detailed description of the ADDC framework for the evolution of digital circuits using GE, an HDL and a logic simulator was presented. ADDC was tested and attained good performance on a selection of combinational circuit benchmark problems.

This chapter extends the work done from the previous chapter by testing ADDC on another set of combinational circuit benchmark problems with a focus on grammar design and initialisation routines, as these are important for defining the boundaries of the search space and performing an effective search, respectively.

The results of experiments presented in this chapter were published in the proceedings of ECTA 2020 (Ryan. et al., 2020). In addition, the results of the extension of this work are presented. The initial work explored the benefit of evolving **combinational** circuits at a higher level abstraction through the use

of GE with SystemVerilog. It also investigated the impact of introducing simple domain knowledge into grammars and their impact on the evolutionary performance. The initial work is extended in two ways: inclusion of two **sequential** circuit benchmark problems and determining a better GE initialisation routine for this set of circuit design benchmark problems. All circuits are behaviourally modelled using only synthesizable SystemVerilog constructs.

The aims of this chapter are summarised as follows:

- Investigate the applicability of GE to circuit design and how best to set it up;
- Evolve behavioural models of combinational and sequential circuits;
- Investigate the effect of exploiting information known about a problem to obtain a grammar variant for each of the combinational circuit benchmarks;
- Identify the most suitable GE initialisation scheme for digital circuit benchmarks.

## 5.2 Overview

Recall digital circuits fall under two main categories: combinational and sequential circuits. The output(s) of a combinational circuit depend(s) solely on its current input. Sequential circuits, in addition to combinational logic, use memory elements such as flip flops, and their output depends on both the current and previous states of the circuits.

CGP has been used to evolve combinational circuits such as adders and multipliers of small input sizes (Sekanina et al., 2011), SSD, etc. Circuits such as the 1-*bit* adder, SR-latch and gated D-latch (sequential circuits) have been evolved at the gate level (Cullen, 2008), although SR-latch results were not shown due to space limitations. In (Vassilev and Miller, 2000), a 3-bit multiplier using binary multiplexers 9- and 25-median approximate circuits (Vasicek and Sekanina, 2015) were evolved at FLE.

In (Henson et al., 2018), CGP was used to evolve sequential circuits. CGP was extended to incorporate forward levels, allowing feedback required for sequential circuits. A D Flip-flop was used as a function in addition to logic gates primitive functions for the evolution of sequential circuits. A custom-built translation system is then used to convert each evolved candidate sequential circuit into a VHDL circuit specification, which is then simulated using Xilinx software. All candidate circuits are subjected to timing analysis, an exercise which greatly increases the evaluation time. For example, it took approximately 2 days and 20 hours to evolve a 4-*bit* up-counter (Henson et al., 2018). Perhaps a compromise, such as applying timing analysis to circuits only after they have either attained approximate functional correctness or fully functional correctness, may reduce experimental time.

As discussed in Chapter 3, existing systems evolve circuits at the gate level, which does not scale quite well with increasing circuit complexity. With the use of HDLs, evolving circuits at the higher abstraction levels is possible: RTL and behavioural level.

## 5.3 Experimental Design

ADDC's circuit design flow was shown in Figure 4.2. It captures only functional simulation, which is the current focus, without considering other circuit qualities such as size, propagation delay and power consumption. The selected benchmark problems are briefly described in Section 5.3.1. For each benchmark problem, a testbench is designed, which is used by the fitness evaluation function to determine the fitness score to assign to a candidate circuit.

The parameters used for the experimental setup are shown in Table 5.1. The parameter values were determined based on results obtained from initial experiments. Two different set-ups based on two initialisation routines are used to run each experiment: SI and Probabilistic Tree Creation 2 (PTC2). This is to enable us to ascertain the most suitable GE initialisation routine for these digital circuit benchmark problems. The selection of the PTC2 initialisation routine is based on evidence in (Nicolau, 2017), where PTC2 adapted for use with GE consistently performed better compared to other initialisation schemes on several symbolic

regression and classification problems. It produced the best performing solutions both with and without access to domain knowledge. This particular PTC2 implementation was based on the versions introduced in (Harper, 2010; Luke, 2000). It uses a pre-specified expansion budget to randomly select a non-terminal to expand while favouring recursive productions with minimum expansion requirements less or equal to the remaining expansions. Additionally, the number of expansions required to fully expand the outstanding non-terminals is budgeted.

**Table 5.1:** Experimental Run Parameters.

| **Parameter Type** | **Parameter Value** | |
|---|---|---|
| | ***Set-up 1*** | ***Set-up 2*** |
| Initialization | Sensible Initialization | PTC2 |
| Selection | Lexicase Selection | |
| No of generations | 200 | |
| Mutation rate | 0.01 | |
| Crossover rate | 0.8 | |
| Replacement rate | 0.5 | |
| No of runs | 30 | |
| Population size | 2000 | |

### 5.3.1 Benchmark Problems

Three combinational and two sequential circuit benchmarks are used, as Table 5.4 shows. Each combinational circuit has a grammar variant version designed to investigate the effect of the introduction of simple domain knowledge into grammar designs. The three combinational circuits benchmark considered are the 11-bit multiplexer, the SSD, and the Hamming Code (7,4) Decoder. The sequential circuits used are the JK flip-flop and the Mod-N Up-Down counter. Existing works mostly target the evolution of up-counters (Harper, 2010; Soleimani et al., 2011; Soliman and Abbas, 2004). In (Henson et al., 2018), 2-, 3- and 4- bit up-counters are evolved; 3-bit up-counters were also evolved in (Soleimani et al., 2011; Soliman and Abbas, 2004). The selection of the JK flip-flop as one of the sequential benchmark problems was partly motivated by its use as a memory element in the evolution of the 3-bit up-counter in (Soliman and Abbas, 2004).

The grammars shown in the listings in the subsequent sections are divided into two sections using a dashed line. The upper section is composed primarily of rules that require expansion and/or form an important part of the evolved program. The lower section contains rules that do not require expansion.

#### 5.3.1.1 11-bit Multiplexer

The 11-bit multiplexer has 8 data and 3 address/select input lines; it has been evolved through the use of both GP (Koza, 1992) and a classifier system (Fredivianus et al., 2010). Two grammars are employed. Multiplexer Grammar A, in Listing 4, uses the following operators as its function set: bitwise-`or`, bitwise-`and`, ternary operator (`if-else`), and logical negation. Multiplexer Grammar B, on the other hand, uses the `ternary operator` as the only function, as shown in Listing 5, which we speculated to be adequate to solve the 11-bit multiplexer problem. The data and address bits of the multiplexer are distinguished from each other using two separate rules, ⟨data-bit⟩ and ⟨address-bit⟩, respectively. These sorts of constraints are easily imposable in GE, as it is simply a matter of grouping the two types of terminals under different rules. Achieving the same in standard GP would be a more complex exercise as it requires the introduction of a new type that satisfies GP's closure principle. Both grammars use the `always` block. These grammars are explained in detail in Section 5.4.2.

#### 5.3.1.2 Seven Segment Display

Unlike the SSD evolved in Chapter 4, which was capable of displaying decimal numerals and A-F letters, the SSD specification considered in this chapter is limited to only digits. The device receives a BCD (0000 - 1001) as input, which gets decoded into a 7-bit binary number. Each bit of the 7-bit binary number corresponds to a segment/LED of the device; a bit value of 1 and 0 indicates the corresponding segment's LED to be turned ON or OFF, respectively, displaying the integer representation of the 4-bit binary number input. Both SSD Grammars in Listings 6 and 7 use the `switch-case` construct as the only functional construct. Similar to the Multiplexer Grammars, the BCD and seven-segment values are distinguished from each other using ⟨bcd-value⟩ and ⟨seven-segment⟩

```
<design-module> ::= <begin-module><nextline><code-block><end-module>
                    <nextline>
   <code-block> ::= <always> <statement>
       <always> ::= "always@(*)" <nextline>
    <statement> ::= <output> "=" <expr>;
                    <nextline>
         <expr> ::= (<expr> & <expr>) |(<expr> "|" <expr>)
                  |!(<expr>)|(<address-bit> ? <expr> ":" <expr>)
                  |<data-bit>
  <address-bit> ::= a0 | a1 | a2
     <data-bit> ::= d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7
-----------------------------------------------------------------------
        <output> ::= out
 <begin-module> ::= "module mux(output logic out, input logic a0,
                    a1, a2, d0, d1, d2, d3, d4, d5, d6, d7);"
   <end-module> ::= "endmodule"
     <nextline> ::= "\n"
```

**Listing 4:** 11-bit Multiplexer Grammar A (The start symbol is highlighted in blue).

```
<design-module> ::= <begin-module><nextline><code-block><end-module>
                    <nextline>
   <code-block> ::= <always> <statement>
       <always> ::= "always@(*)" <nextline>
    <statement> ::= <output> "=" <expr>; <nextline>
         <expr> ::= (<address-bit> ? <expr> ":" <expr>) | <data-bit>
  <address-bit> ::= a0 | a1 | a2
     <data-bit> ::= d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7
-----------------------------------------------------------------------
        <output> ::= out
<begin-module> ::= "module mux(output logic out, input logic a0,
                    a1, a2, d0, d1, d2, d3, d4, d5, d6, d7);"
   <end-module> ::= "endmodule"
     <nextline> ::= "\n"
```

**Listing 5:** 11-bit Multiplexer Grammar B (The start symbol is highlighted in blue).

rules, respectively. To investigate the effect of domain knowledge/grammar design choices, SSD Grammar A in Listing 6 assumes no knowledge about the problem and, therefore, the valid bcd and seven-segment binary numbers are left to GE to deduce. This is a non-trivial task as not all the values are required. However, SSD Grammar B in Listing 7 explicitly provides the BCD and seven-segment values, making it the least difficult and should record the best success rate.

```
<design-module> ::= <begin-module><nextline><code-block><next-line>
                    <end-module>
   <code-block> ::= <always><stmt>
       <always> ::= always@(bcd) <next-line>
         <stmt> ::= begin <nextline><switch-case><next-line> end
  <switch-case> ::= case(bcd) <nextline> <case-stmt> <next-line>
                    <default-case><nextline> endcase
    <case-stmt> ::= <bcd-value> ":" <output> "=" <seven-segment>;
                  | <bcd-value> ":" <output> "=" <seven-segment>;
                    <nextline> <case-stmt>
 <default-case> ::= default ":" <output> "=" <seven-segment>;
    <bcd-value> ::= 4'b<bit><bit><bit><bit>
<seven-segment> ::= 7'b<bit><bit><bit><bit><bit><bit><bit>
          <bit> ::= 0 | 1
-------------------------------------------------------------------------
       <output> ::= segment
 <begin-module> ::= "module ssd(output logic [6:0] segment, input
                    logic[3:0] bcd);"
   <end-module> ::= endmodule
     <nextline> ::= "\n"
```

**Listing 6:** Seven Segment Display Grammar A (The start symbol is highlighted in blue).

#### 5.3.1.3 Hamming Code (7,4) Decoder

Linear block codes are a category of codes capable of double error detection and single error correction. Thus, they have a minimum distance of three. A Hamming Codes belong to this category of codes (Miller et al., 2009). Hamming Code (7,4) Encoder encodes a 4-bit data word by generating three parity check bits. The 3 parity check bits and 4-bit data word are merged into a 7-bit binary string termed as codeword either in a systematic or non-systematic encoding style before transmission. At the reception point, the Hamming Code (7,4) Decoder retrieves the transmitted data word by generating a 3-bit binary number from the received codeword `referred to as the syndrome`. Each bit in the syndrome serves as a parity check for the 4-bit data word. Hamming Code (7,4) Decoder Grammars A and B are shown in Listings 8 and 9 respectively.

Both grammars are designed to define a `function` that encapsulates instructions responsible for the generation of the syndrome from the received codeword. For Hamming Code Decoders, if the syndrome is non-zero, the codeword has been corrupted during transmission. However, the integer equivalence of the non-zero

```
<design-module> ::= <begin-module><nextline><code-block><next-line>
                    <end-module>
   <code-block> ::= <always><stmt>
       <always> ::= always@(bcd) <next-line>
         <stmt> ::= begin <nextline><switch-case><next-line> end
  <switch-case> ::= case(bcd) <nextline> <case-stmt> <next-line>
                    <default-case><nextline> endcase
    <case-stmt> ::= <bcd-value> ":" <output> "=" <seven-segment>;
                  | <bcd-value> ":" <output>  "=" <seven-segment>;
                    <nextline> <case-stmt>
 <default-case> ::= default ":" <output> "="
                    <seven-segment>;
    <bcd-value> ::= 4'b0000 | 4'b0001 | 4'b0010 | 4'b0011 | 4'b0100
                  | 4'b0101 | 4'b0110 | 4'b0111 | 4'b1000 | 4'b1001
<seven-segment> ::= 7'b1111110 | 7'b0110000 | 7'b1101101
                  | 7'b1111001 | 7'b0110011 | 7'b1011011
                  | 7'b1011111 | 7'b1110000 | 7'b1111111
                  | 7'b1111011 | 7'b0000000
          <bit> ::= 0 | 1
-------------------------------------------------------------------------
       <output> ::= segment
 <begin-module> ::= "module ssd(output logic [6:0] segment, input
                    logic[3:0] bcd);"
   <end-module> ::= endmodule
     <nextline> ::= "\n"
```

**Listing 7:** Seven Segment Display Grammar B (The start symbol is highlighted in blue).

syndrome is assumed to be the bit position in the codeword where the possible error occurred. The bit in that position is flipped in an attempt to correct the codeword. The main difference between Hamming Code (7,4) Decoder Grammar A and B is the ⟨expr⟩ rule. Hamming Code (7,4) Decoder Grammar A contains no domain knowledge and is required to evolve expressions that generate the parity bits that form the syndrome.

For Grammar B, some domain knowledge is introduced by restricting the generated expressions to use exactly four bits out of the 7-bit codeword, which is known from the problem. As a result, speculatively, Hamming Code (7,4) Decoder Grammar B should perform better than Grammar A. Both grammars also use the `always` procedural block.

```
<design-module> ::= <begin-module><nextline><declarations><func>
                    <nextline><always-block><nextline><end-module>
 <declarations> ::= <d-var-type> <d-var-name>; <nextline>
 <always-block> ::= always@(<input-codeword>) begin <nextline>
                    <output><nextline><syndrome><nextline><if-else>
                    <nextline> end
      <if-else> ::= if(<condition>) begin end <nextline> else
                    begin <nextline><output-codeword>[<d-var-name>]
                    "=" ~<input-codeword> [<d-var-name>]; end
    <condition> ::= <d-var-name> "==" "3'b0"
       <output> ::= <output-codeword> "=" <input-codeword>;
     <syndrome> ::= <d-var-name> "=" <func-call>
    <func-call> ::= <func-name> (<input-codeword>);
         <func> ::= function <func-return-type><func-name>
                    (<func-input>); <nextline> begin <nextline>
                    <func-stmts> end <end-func>
 <func-stmts>   ::= syndrome[0] "=" <expr>; <nextline>
                    syndrome[1] "=" <expr>;<nextline>
                    syndrome[2] "=" <expr>; <nextline>
          <expr> ::= <sel-var> | <sel-var> <op> <expr>
       <sel-var> ::= <i-var>[<index>]
   <func-input> ::= input <range> <i-var>
         <index> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7
            <op> ::= ^ | & | "|"
-------------------------------------------------------------------------
<func-return-type> ::= "[2:0]"
       <func-name> ::= syndrome
           <range> ::= "[1:7]"
      <i-variable> ::= cw
      <d-var-type> ::= integer
      <d-var-name> ::= syndrome_value
  <input-codeword> ::= i_codeword
 <output-codeword> ::= o_codeword
        <end-func> ::= endfunction
    <begin-module> ::= "module hc_decoder(input [1:7] i_codeword,
                        output logic [1:7] o_codeword);"
     <end-module>  ::= endmodule
       <nextline>  ::= "\n"
```

**Listing 8:** Hamming Code (7,4) Decoder Grammar A (The start symbol is highlighted in blue).

#### 5.3.1.4 JK Flip-flop

Flip-flops are bistable circuits used for the storage of information. A JK flip-flop has four possible states with no invalid states. No output change occurs when both inputs J and K have an input value of 0. However, when both J and K

```
<design-module> ::= <begin-module> <nextline><declarations><func>
                    <nextline><always-block><nextline><end-module>
 <declarations> ::= <d-var-type> <d-var-name>; <nextline>
 <always-block> ::= always@(<input-codeword>) begin <nextline>
                    <output><nextline><syndrome><new-line><if-else>
                    <nextline> end
      <if-else> ::= if(<condition>) begin end <nextline> else begin
                    <nextline><output-codeword>[<d-var-name>] "="
                    ~<input-codeword>[<d-var-name>]; end
    <condition> ::= <d-var-name> "==" "3'b0"
       <output> ::= <output-codeword> "=" <input-codeword>;
     <syndrome> ::= <d-var-name> "=" <function-call>
    <func-call> ::= <funct-name> (<input-codeword>);
         <func> ::= function <func-return-type><func-name>
                    (<func-input>); <nextline> begin <nextline>
                    <func-stmts> end <end-func>
   <func-stmts> ::= syndrome[0] "=" <i-var>[<idx>] <op> <i-var>
                    [<idx>] <op> <i-var>[<idx>] <op> <i-var>[<idx>];
                    <nextline> syndrome[1] "=" <i-var>[<idx>] <op>
                    <i-var>[<idx>] <op><i-var>[<idx>] <op><i-var>
                    [<idx>]; <nextline> syndrome[2] "=" <i-var>
                    [<idx>] <op> <i-var>[<idx>]<op> <i-var>[<idx>]
                    <op><i-var>[<idx>]; <nextline>
   <func-input> ::= input <range> <i-var>
          <idx> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7
           <op> ::= ^ | & | "|"
-------------------------------------------------------------------------
<func-return-type> ::= "[2:0]"
       <func-name> ::= syndrome
           <range> ::= "[1:7]"
           <i-var> ::= cw
      <d-var-type> ::= integer
      <d-var-name> ::= syndrome_value
  <input-codeword> ::= i_codeword
 <output-codeword> ::= o_codeword
    <end-function> ::= endfunction
    <begin-module> ::= "module hc_decoder(input [1:7] i_codeword,
                        output logic [1:7] o_codeword);"
     <end-module>  ::= endmodule
       <new-line>  ::= "\n"
```

**Listing 9:** Hamming Code (7,4) Decoder Grammar B (The start symbol is highlighted in blue).

input values are 1, the current output value toggles. When $J = 0$, $K = 1$, the output is 0 and when $J = 1$, $K = 0$, the output is 1 at the clock transition. The JK flip-flop specification is captured in Table 5.2. Shown in Listing 10 is the JK

flip-flop grammar designed and used for evolution. The grammar makes use of `always` block and `switch-case` construct. The grammar uses the ⟨case⟩ rule to evolve the flip-flops input (J and K) and the ⟨statment⟩ rule to evolve the correct output depending on the J and K input combinations.

```
<design-module> ::= <begin-module><new-line><code-block><new-line>
                    <end-module>
    <code-block> ::= <always-block><new-line><switch-case>
  <always-block> ::= "always @(posedge clk)"
   <switch-case> ::= "case ({j,k})" <new-line><case-stmt><new-line>
                     endcase
     <case-stmt> ::= <case> ":" <stmt><new-line><case-default>
                   | <case> ":" <stmt><new-line><case-stmt>
  <case-default> ::= default ":" <stmt>
          <case> ::= "2'b"<bit><bit>
          <stmt> ::= q "<=" <terminal>;
           <bit> ::= 0 | 1
      <terminal> ::= <bit> | <lvalue> | "~"<lvalue>
--------------------------------------------------------------------------
        <lvalue> ::= q
  <begin-module> ::= "module jk_ff(input j, input k, input clk,
                     output logic q);"
    <end-module> ::= endmodule
      <new-line> ::= "\n"
```

**Listing 10:** JK Flipflop Grammar (The start symbol is highlighted in blue).

**Table 5.2:** JK Flip-flop Truth Table. ↑ represents the positive transition of the clock from 0 to 1; thus, the rising edge of the clock.

| CLK | J | K | Q |
|---|---|---|---|
| ↑ | 0 | 0 | Q (no change) |
| ↑ | 0 | 1 | 0 |
| ↑ | 1 | 0 | 1 |
| ↑ | 1 | 1 | ¬Q (toggle) |

#### 5.3.1.5 Modulus-16 (Mod-16) Up-Down Counter

A counter is a sequential logic device that increases or decreases its content by 1, or by some defined step value, when triggered by a clock signal. The mod-16

up-down counter is capable of counting up and down its content constrained to a range of 0 to 15. The specification for the Mod-16 Up-Down counter is captured in the transition table in Table 5.3. The counter has an active low reset signal. The Mod-16 Grammar shown in Listing 11 uses the `always` block, `switch-case`, and two arithmetic operators (−,+) for the counting operations. The ⟨case⟩ rule is used to evolve the state of the counter, and the ⟨stmt⟩ rule to evolve the appropriate output based on the state of the counter.

**Table 5.3:** Mod-16 Up-Down Counter Transition Table. '↑' and '↓' represent positive and negative transitions of the clock respectively. The '?' symbol represents don't care bits.

| CLK | RESET | EN_LOAD | EN_UP | LOAD | UP_DOWN | Q (OUTPUT) |
|---|---|---|---|---|---|---|
| ↑ | 0 | ? | ? | ? | ? | 0 |
| ↑ | 1 | 1 | ? | 4-BIT NUMBER | ? | LOAD |
| ↑ | 1 | 0 | 1 | ? | 1 | Q + 1 (count up) |
| ↑ | 1 | 0 | 1 | ? | 0 | Q - 1 (count down) |
| ↑ | 1 | 0 | 0 | ? | ? | Q (no change) |
| ↓ | ? | ? | ? | ? | ? | ? |

### 5.3.2 Testing

The number of cases used to evolve each of these circuits are shown in Table 5.4. The 11-bit Multiplexer ($2^{11}$ cases), SSD ($2^4$ cases) and Hamming Code (7,4) Decoder ($2^7$ cases) are exhaustively tested.

A total of 8 cases were used to evolve the JK Flip-flop, with multiple states tested more than once. The no-change state is tested twice when the clock is high and once when the clock is low. Q = 0 state is tested once. The Q=1 and toggle states are each tested twice in succession to ensure that the behaviours of the no-change and toggle states are properly evolved.

The total number of cases used to test the Mod-16 Up-Down Counter is 27. All counter states are tested several times successively or at different times in order to ensure each state is properly tested. The counter is reset 8 times. 10

```
<design-module> ::= <begin-module><always><endmodule>
       <always> ::= "always @(posedge clk)" <nextline><switch-case>
         <expr> ::= <terminal> | <terminal><op><terminal>
                  | <terminal><op><expr>
  <switch-case> ::= "casex ({reset, en_load, enable_ud, up_ndown})"
                    <nextline> <case-stmt> endcase <nextline>
    <case-stmt> ::= <case> ":" <stmt> | <case> ":" <stmt><case-stmt>
         <stmt> ::= "q<=" <expr>; <nextline>
         <case> ::= 4'b<bit><bit><bit><bit>
     <terminal> ::= 0 | 1 | q | load
          <bit> ::= 0 | 1 | ?
           <op> ::=  "-" | "+"
-------------------------------------------------------------------------
 <begin-module> ::= "module udcounter(input logic clk, up_ndown,
                     en_load, enable_ud, reset, input logic [3:0]
                     load, output logic [3:0] q);" <nextline>
    <endmodule> ::= endmodule
     <nextline> ::= "\n"
```

**Listing 11:** Mod-16 Up-Down Counter (The start symbol is highlighted in blue).

**Table 5.4:** Number of cases used for each problem. Sequential circuit benchmarks are marked with *.

| Problem | Number of Testcases. |
|---|---|
| 11-Multiplexer | 2,048 |
| Seven Segment Display | 16 |
| Hamming Code (7,4) Decoder | 128 |
| JK Flip-flop* | 8 |
| Mod-16 Up-Down Counter* | 27 |

cases are used to test the load state. 3 cases each are used to test the no change, count up and count down states.

## 5.4 Results and Discussion

In this section, we show results obtained from all the experiments conducted together with some discussions. The average mean best per generation across the 30 independent runs conducted for each of the 5 selected benchmark problems with error bars are plotted. Each graph shows the results for the two experimental

configurations: SI and PTC2. These are shown in Figures 5.1 to 5.8.

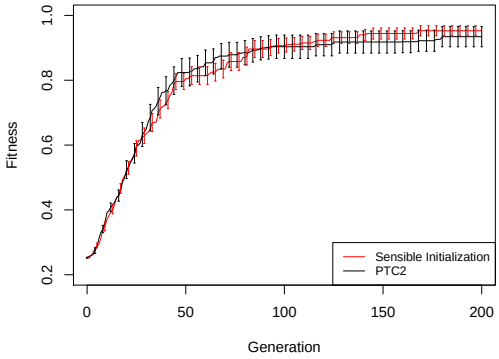


**Figure 5.1:** Mean best across 30 independent runs using Hamming Code (7,4) Decoder Grammar A with error bars.

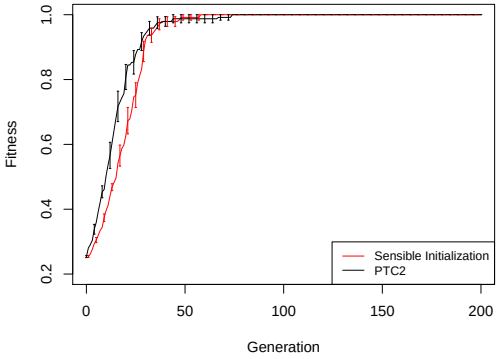


**Figure 5.2:** Mean best across 30 independent runs using Hamming Code (7,4) Decoder Grammar B with error bars.

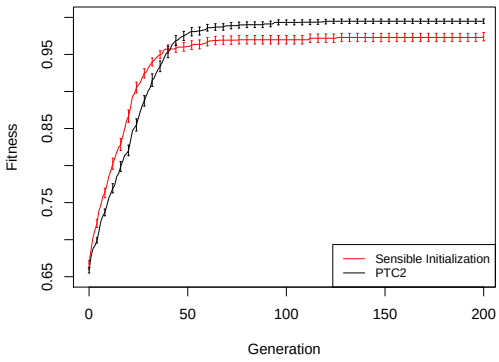


**Figure 5.3:** Mean best across 30 independent runs using 11-bit Multiplexer Grammar A with error bars.

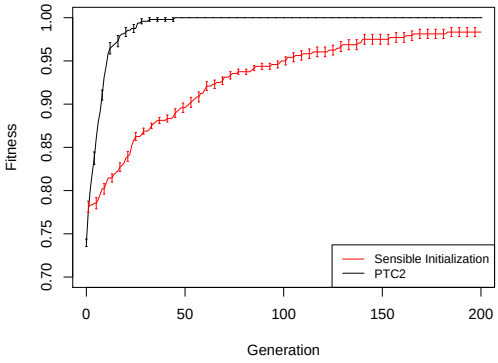


**Figure 5.4:** Mean best across 30 independent runs using 11-bit Multiplexer Grammar B with error bars.

### 5.4.1 Success Rate

The success rates for all experiments are tabulated in Table 5.5, showing both the impact of grammar design and the effect of initialisation routines on evolutionary performance.

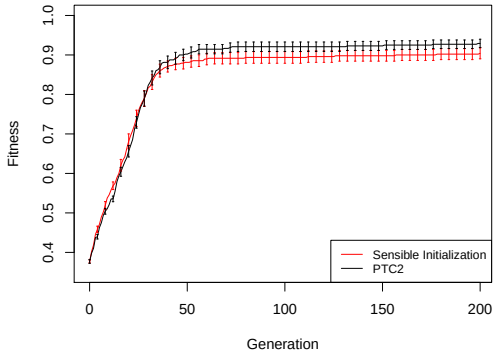


**Figure 5.5:** Mean best across 30 independent runs using SSD Grammar A.

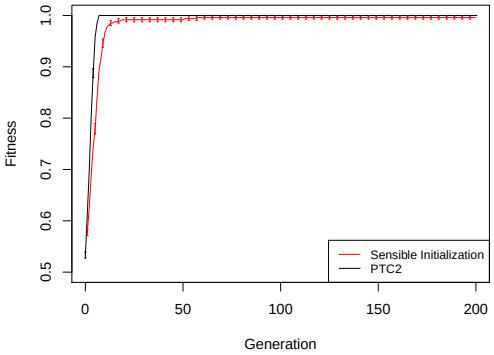


**Figure 5.6:** Mean best across 30 independent runs using SSD Grammar B.

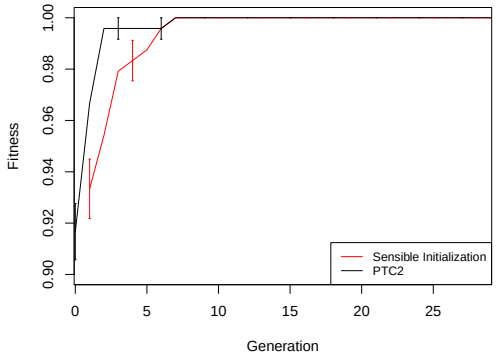


**Figure 5.7:** Mean best across 30 independent runs for JK Flip-flop.

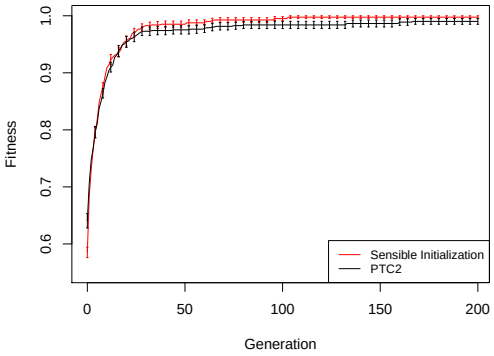


**Figure 5.8:** Mean best across 30 independent runs for *Mod*-16 up-down counter.

Candidate circuits of the combinational circuit benchmark problems that took more than 10 seconds during simulation were assigned a fitness score of zero to reduce their probability of being selected as parents for reproduction, thereby improving the evolutionary performance. This is because it was observed that during the initial phase of running the experiments, some candidate circuits contained very long expressions, which also created cyclic dependencies, causing the circuit simulator to hang. As a result, any circuit simulation that exceeds 10 seconds was terminated and awarded a fitness value of zero to allow evolution to progress. The 10 seconds were more than sufficient as the circuits required at

**Table 5.5:** Success Rate for All Experimental Configurations.

| **Problem Grammar** | **Successful Runs using Sensible Initialisation** (out of 30) | **Successful Runs using *PTC2*** (out of 30) |
|---|---|---|
| 11–bit Multiplexer Grammar A | 17 | 27 |
| 11–bit Multiplexer Grammar B | 22 | 30 |
| Seven Segment Display Grammar A | 8 | 8 |
| Seven Segment Display Grammar B | 28 | 30 |
| Hamming Code (7,4) Decoder Grammar A | 22 | 26 |
| Hamming Code (7,4) Decoder Grammar B | 30 | 30 |
| JK Flip-flop | 30 | 30 |
| Mod-16 Up-Down Counter | 29 | 26 |

most 5 seconds to evaluate. As a result, 17 and 8 optimal solutions for 11-bit Multiplexer Grammar A and SSD Grammar A were obtained, respectively.

Using PTC2, 27 and 8 optimal solutions using 11-bit Multiplexer Grammar A and SSD Grammar A were obtained, respectively. Across all problems, PTC2 obtained higher success rates except on the Seven Segment Grammar A, Hamming Code (7,4) Decoder Grammar B, and JK Flip-flop, where the tally was equal, and on the Mod-16 up-down counter, where SI recorded three more successful runs. All Grammar B versions for all three combinational circuit benchmark problems performed better than their Grammar A counterparts because some simple domain knowledge was infused into these grammars.

### 5.4.2 Grammar Design

In order to assess the impact of grammar design choices and the introduction of simple domain knowledge into grammars, two grammar variants are designed for each of the three combinational circuit benchmarks used: Hamming Code (7,4) Decoder, 11-bit multiplexer, and SSD. As speculated, the use of simple domain knowledge in some grammars had the intended influence.

The 11-bit Multiplexer Grammar A contained no domain knowledge and therefore contained more functions in its function set (ternary operator and bit-

wise operators). The 11-bit Multiplexer Grammar B used only a ternary operator, which, based on the behavioural description of a multiplexer, was speculated to be sufficient for modelling the behaviour of a multiplexer. From Figure 5.3 and Figure 5.4, it can be observed that Multiplexer Grammar B outperformed Grammar A irrespective of the initialisation routine used. The choice of highly expressive constructs/functions such as `if-else` at the behavioural level had the intended effect.

Recall that the SSD Grammar A version had to construct the BCD and segment values from bits. The SSD Grammar B version had these values explicitly provided for in the grammar. As speculated, the Grammar B version performed better than Grammar A because Grammar A had the additional task of evolving the appropriate BCD and seven-segment values available in Grammar B.

The Hamming Code (7,4) Decoder Grammars differ only in the ⟨expr⟩ rule. The ⟨expr⟩ rule for Grammar A has recursive productions. As a result, evolution has to evolve arbitrary length expressions that generate the correct parity bit. In the case of Grammar B, the ⟨expr⟩ length of its final expression was constrained by limiting it to always select exactly 4 bits out of the 7-bit codeword, and the information is derived from the Hamming Code (7,4) Decoder problem. This makes the Hamming Code (7,4) Decoder Grammar B more likely to solve the problem faster and with a guarantee of no bloated expressions. From Figure 5.1 and Figure 5.2, it is observed that the Hamming Code (7,4) Decoder Grammar B outperforms Grammar A even with the use of different initialisation schemes. This shows that, despite the importance of good initialisation schemes, they do benefit from concise grammars with simple domain knowledge infused into them, as it is possible to impose simple but generic constraints on the search space through the introduction of simple domain knowledge into grammars.

Evolved representation solutions for each grammar for each of the five benchmark problems are shown in the Appendix A.

### 5.4.3 Higher Abstraction Levels

Modelling circuits at a higher abstraction level, such as RTL or behavioural, is better suited for evolving complex circuits but may not always guarantee op-

timised designs (circuits with good power consumption, delay and size) after synthesis. This is due to the fact that circuit optimisation is left to the mercy of synthesis tools, though modern synthesis tools are very good at optimisation. A 1-bit full adder was evolved in RTL using GE with a population size of 200 and a 100,000 fitness evaluations budget (Karpuzcu, 2005). Out of 35 independent experimental runs, only two optimal solutions were obtained. Using ADDC with the same grammar and population size of (Karpuzcu, 2005), but opting for 200 generations (40,000 maximum fitness evaluations), 8 out of 30 optimal solutions were obtained. These solutions were found on average after 4,600 fitness evaluations. Detailed results of this experiment are not provided in this thesis, as a 1-bit full adder is trivial to evolve, and Verilog/SystemVerilog's behavioural constructs were not exploited enough in the grammar design, greatly due to the triviality of the chosen problem (1-bit adder).

To investigate the benefit of using the behavioural constructs in Verilog/SystemVerilog, two grammar variants per combinational circuit design problem were used, as described in Section 5.3.1, with a focus on grammar design choices and the effect of introducing simple domain knowledge about problems into grammar. Results, as discussed in Section 5.4.2, revealed that introducing simple information known about problems greatly improves the evolutionary search, thereby increasing the success rate. In Chapter 6, the parameterisation construct in SystemVerilog will be exploited to evolve very complex parameterised selective parity, adder, and multiplier combinational circuits using GE.

### 5.4.4 Impact of Initialisation Schemes on Circuit Design Benchmark Problems

PTC2 produces the best performance on almost all experiments using both grammars with and without domain knowledge introduced into them. However, experiments using SI obtained slightly better mean best fitnesses on the Hamming Code (7,4) Decoder grammars in Figures 5.1 and 5.2 approximately after 80 and 65 generations. SI outperformed PTC2 marginally and quite significantly on Hamming Code (7,4) Decoder Grammars A and B, respectively. Similarly, SI slightly outperforms PTC2 on *mod*-16 up-down counter. PTC2 obtained the

worst starting mean best fitness scores in the initial generations for the 11-bit multiplexer (notably), SSD (slightly) and JK flip-flop problems but converged quicker to fitter and optimal solutions in general. This implies good fitness scores in initial generations do not necessarily guarantee a good overall performance, which is consistent with findings in (Nicolau, 2017).

## 5.5 Conclusions

In this chapter, ADDC was used to evolve three combinational and two sequential benchmark problems. Two grammar variants per combinational circuit were designed – one version without domain knowledge and the other with simple domain knowledge infused into it. Results obtained revealed how grammars can be leveraged to dramatically reduce the search space, such as separating out clearly distinct wires, for example, input wires from output wires, and the inclusion of relevant functions only, such as in the case of the Multiplexer Grammar A.

Results obtained from the experiments reveal the impact of high-level constructs, such as `always` procedural block, `if-else`, `switch-case`, etc., on conventional circuit evolution. Due to the expressiveness of these programming constructs, they can replace many gates that would otherwise need to be evolved in gate-level evolution. This allows evolution to focus on evolving complex behaviours. All cases where these features could be tested benefited from the introduction of simple domain knowledge.

Furthermore, experimental setups using PTC2, overall, had consistent results across all problems, even on grammars without any domain knowledge infused into them. This makes PTC2 more suitable for circuit design problems. The use of Lexicase selection made it easier to target poorly covered parts of the solution space.

This is because lexicase selects specialists (individuals that perform well on a specific case(s)) and not individuals that have relatively low aggregated error. As a result, lexicase selection can better explore and drive evolution towards solutions (Helmuth et al., 2019, 2020; Jundt and Helmuth, 2019). Preliminary experiments showed that lexicase selection performed better than tournament

selection. However, conducting and reporting an experimental investigation comparing the performance of these selection methods for the problems addressed here is beyond the scope of this thesis.

# 6

# Evolution of Complex Parameterised Digital Circuits in SystemVerilog

## 6.1 Introduction

In Chapter 5, an empirical investigation was carried out to ascertain the impact of grammar design choices and the choice of initialisation routines on evolutionary performance ADDC.

This chapter focuses on exploiting the programming constructs in SystemVerilog, our HDL of choice, for use in grammars to facilitate the evolution of complex circuits. Additionally, corner case testing was employed to significantly reduce the amount of testing since exhaustive testing is not feasible for complex circuits.

We also further examine the use of the parameterisation feature in SystemVerilog to evolve complex and parameterised combinational circuits in SystemVerilog using GE. Some of the results presented in this chapter were previously published in (Tetteh et al., 2021), and an extension of the work also published (Tetteh et al., 2022).

In summary, this chapter tackles the issues of scalability through the use of SystemVerilog with GE to design fully functional and complex combinational

circuits. This is achieved through the use of the parameterisation feature in SystemVerilog without resorting to decomposition techniques.

The aims of this chapter are:

- Evolving a parameterised sequential circuit —Mod-N Up-Down Counter;
- Synthesising and comparing evolved parameterised circuits to state-of-the-art works;
- Investigating the effect of varying population sizes on evolutionary performance;
- Performing statistical analysis on the effect of input bit-width sizes on simulation time to determine appropriate input bit-width sizes for evolving parameterised circuits.

## 6.2 Overview

In Chapter 5, the two major issues confronting the EHW field, specifically, scaling to evolve complex circuits, were discussed. These challenges make the search space large and difficult to search, and only modestly complex circuits (circuits with a small number of inputs and a reasonable number of gates) can be exhaustively tested. To date, the most complex fully functional circuits evolved are a 10-bit + 10-bit adder (Hodan et al., 2020), a 6-bit $\times$ 6-bit multiplier (Stomeo et al., 2006*a*), a 19-bit parity circuit (Li et al., 2008) and a 28-input frg1 circuit (Vasicek and Sekanina, 2014). This chapter aims to answer the following research questions:

1. Can the use of HDLs (SystemVerilog in this case) evolve complex combinational and sequential circuits that compare favourably to state-of-the-art methods in terms of the size of inputs and gate count?
2. What are the drawbacks of evolving circuits using GE and HDLs?

## 6.3 Experimental Design

Four complex circuits are evolved: three combinational and one sequential circuit. The combinational circuits are multiplier, adder and selective parity generator. The chosen sequential circuit is an Up-Down Counter. These problems are representative of evolutionary circuit design benchmark problems in literature (Henson et al., 2018; Hodan et al., 2020; Hrbacek and Sekanina, 2014; Li et al., 2008). All circuits are evolved as parameterised design modules to accommodate all possible ranges of values of the parameters. Thus, evolved parameterised circuit modules retain their functionality for all possible ranges of values of the specific parameter(s). For example, an evolved optimal parameterised adder (N-bit + N-bit adder) must function perfectly as an 8-bit + 8-bit adder, 32-bit + 32-bit adder, etc. Prior to simulation or synthesis, parameter values of evolved parameterised circuits can be specified without requiring a re-run of evolutionary experiments, as is the case for existing EHW methodologies.

Parameterised circuits are designed using the keyword `parameter` in SystemVerilog. The usage of the `parameter` keyword is highlighted in yellow in the corresponding production of ⟨design-module⟩ rule for all benchmark circuit grammars. The grammars are divided into two segments separated by a dashed line. The top segment consists of rules that require expansion, in other words, the evolvable part of the grammar. The bottom segment (fixed grammar part) consists of rules that describe the corresponding circuit's interface, variable/register declaration and initialisation, required programming constructs (e.g. `always` block, `generate for-loop`), etc. The bottom segment can be left out of the grammar as all the rules consist of single productions. Alternatively, a circuit module template can be designed with a placeholder to represent the evolved segment.

### 6.3.1 Benchmark Problems

#### 6.3.1.1 Selective Parity Circuit

An odd parity circuit generates a bit value of 1 when the data to be transmitted contains an even number of 1s, and 0 when it contains an odd number of 1s.

```
<design-module> ::= <begin-module><declarations><initialize>
                    <for-loop><newline><output><end-module>
 <declarations> ::= "logic [width:0] intmd_parity;" <newline>
                    genvar i; <newline>
     <for-loop> ::= "for(i=0; i<width; i=i+1)" begin <newline><stmt>
                    end <newline>
         <stmt> ::= assign intmd_parity[i+1]"="<expr>;<newline>
       <output> ::= assign parity "=" intmd_parity[width]; <newline>
   <initialize> ::= assign parity "=" even_odd;<newline>
         <expr> ::= <select-bit> | <select-bit><bitwise-op><expr>
                  | ~(<expr>)
   <select-bit> ::= intmd_parity[i] | data[i]
   <bitwise-op> ::= ^ | & | "|" | ~^
-------------------------------------------------------------------------
 <begin-module> ::= "module parity #(parameter width=128) (input
                    logic [width-1:0] data, input logic even_odd,
                    output logic parity);" <newline>
   <end-module> ::= endmodule
      <newline> ::= "\n"
```

**Listing 12:** N-bit Selective Parity Grammar A (default input bit-width is 128). This uses `generate loop` and `bitwise operators`. Highlighted in blue and yellow are the start symbol and the use of parameterisation, respectively .

Conversely, an even parity circuit generates a bit value of 0 when the data contains an even number of 1s, and 1 when it contains an odd number of 1s.

We design two different parity generator circuit grammars: Selective Parity Grammar A and Selective Parity Grammar B, shown in Listings 12 and 13 respectively. The purpose of designing two separate grammars using different operators is to investigate the benefit of using tuned operators (operators deemed to be relevant to evolving the circuit based on its description or specification) in grammars. Selective Parity Grammar A uses generate loops (synthesisable ⟨for-loop⟩) and bitwise operators (⟨bitwise-op⟩) to loop through the data bits while performing bitwise operations until a parity bit is obtained. Selective Parity Grammar B uses both reduction operators (⟨reduction-op⟩) and bitwise operators (⟨bitwise-op⟩). Reduction operators apply bitwise operations on the bits of an n-bit operand (a vector) recursively to produce a scalar output.

```
    <design-module> ::= <begin-module> <code-block> <end-module>
       <code-block> ::= <always-block><newline>
     <always-block> ::= always@(data or even_odd) <newline><stmt>
            <stmt> ::= parity "=" <expr>;
            <expr> ::= <reduct-expr> | (<reduct-expr><bitwise-op>
                       <reduct-expr>)| (<reduct-expr><bitwise-op>
                       <expr>)
      <reduct-expr> ::= <reduction-op><input>
                      | <reduction-op>(<reduct-expr>)
                      | (<reduction-op>{<multi-concat-expr>})
<multi-concat-expr> ::= <input> | <reduct-expr>
                      | <multi-concat-expr>,<multi-concat-expr>
     <reduction-op> ::= & | ~& | "|" | ~"|" | ^ | ~^
       <bitwise-op> ::= ^ | & | "|" |  ~^
            <input> ::= data | even_odd
-------------------------------------------------------------------------
     <begin-module> ::= "module parity #(parameter width=128) (input
                       logic[width-1:0] data, input logic even_odd,
                       output logic parity);"<newline>
       <end-module> ::= endmodule
```

**Listing 13:** N-bit Selective Parity Grammar B (default input bit-width is 128). This uses `reduction` and `bitwise` operators. Highlighted in blue and yellow are the start symbol and the use of parameterisation respectively.

#### 6.3.1.2 N-bit + N-bit Adder

An adder circuit performs addition in digital electronic devices. The adder grammar is shown in Listing 14 and uses the `always` (⟨always-block⟩) procedural block, which takes a sensitivity list as arguments and executes the statements within its code block whenever a signal within the list changes. The operators used are a binary arithmetic operator (+) and bitwise operators (`&`, `|`, ^). The default input bit-width of the addends (numbers to be summed up) is 128, specified using the *parameter* keyword in the ⟨begin-module⟩ rule.

#### 6.3.1.3 N-bit × N-bit Multiplier

A multiplier performs multiplication in digital electronic devices such as computers, calculators, etc. The particular multiplier type considered here is the Add-Shift Multiplier. The Add-Shift Multiplier's operation is based on longhand multiplication. Each digit of the multiplier multiplies the multiplicand to obtain

```
<design-module> ::= <begin-module><nextline><always-block>
                    <end-module>
 <always-block> ::= always @(a, b or carry in)begin<nextline><stmt>
                    <end><nextline>
   <sum-output> ::= {carry out, sum} "=" <expr>;<nextline>
           <io> ::= a | b | carry in
         <expr> ::= <io> | <io><op>(<expr>) | ~(<expr>)
           <op> ::= + | & | "|" | ^
-------------------------------------------------------------------------
 <begin-module> ::= "module adder #(parameter width=128) (input
                    logic [width-1:0] a, b, input logic carry_in,
                    output logic carry_out, output logic
                    [width-1:0] sum);"
   <end-module> ::= endmodule
     <nextline> ::= "\n"
```

**Listing 14:** N-bit + N-bit Adder Grammar. Highlighted in blue and yellow are the start symbol and the use of parameterisation, respectively. The expression `{carry out, sum}` represents concatenation in Systemverilog. All three variables or multiple singles are combined to form a single vector to which the result of `expr` is assigned.

an intermediate product shifted a digit to the left of the preceding intermediate product. All intermediate products are then summed up to obtain the product of the multiplication. The Multiplier Grammar is shown in Listing 15. The grammar makes use of `always` (⟨always-block⟩), `for-loop` (⟨for-loop⟩) and `if-else` (⟨if-else⟩) programming constructs in SystemVerilog. Three different operators (⟨op⟩) are used: a binary arithmetic operator (+), shift operators (`«`, `»`) and bitwise operators (`&`, `|`).

### 6.3.2 N-bit Up-Down Counter

Counters are sequential logic devices that store the number of occurrences of an event, usually from a pre-defined state. The counting operation is triggered by a clock signal. A counter that increases its content after every clock cycle is known as an up-counter, while one that decreases its content is referred to as a down-counter. A bi-directional counter functions as an up-and-down counter.

An N-bit Up-Down Counter is a counter that functions as a mod-$2^n$ Up-Down Counter. That is, it can function as a mod- 2, 4, 8, 16, 32, etc., Up-Down

```
<design-module> ::= <begin-module><nextline><declarations><always>
                    <end-module>
       <always> ::= always @(multiplicand or multiplier)<nextline>
                    begin <nextline>
     <for-loop> ::= "for(i=0; i<i_width; i=i+1)" begin <nextline>
                    <if-else><nextline> end <nextline>
    <condition> ::= <io>[i]"=="<bit>
 <declarations> ::= integer<loop-var>;<nextline>
      <if-else> ::= if(<condition>)<nextline> begin <nextline>
                    product "=" <expr>; <next-line> end
                  | if(<condition>)<nextline> begin <nextline>
                    product "=" <expr>; <next-line> end <nextline>
                    else <nextline> begin <nextline> product "="
                    <expr>; <nextline>
         <expr> ::= <io> | <io><op>(<expr>)
           <op> ::= + | << | >> | & | "|"
           <io> ::= multiplier | multiplicand | product | i
          <bit> ::= 1'b0 | 1'b1
-------------------------------------------------------------------------
 <begin-module> ::= module mult #(parameter i_width=64, o_width=128)
                    (input logic [i_width-1":"0] multiplier,
                    multiplicand, output logic [o_width-1":"0]
                    product);
   <end-module> ::= endmodule
     <nextline> ::= "\n"
```

**Listing 15:** N-bit × N-bit Multiplier Grammar. Highlighted in blue and yellow are the start symbol and the use of parameterisation, respectively.

Counters. The transition table for N-bit Up-Down Counter is shown in Table 6.1. The counter has an active low-reset signal, i.e., it resets the counter to State 0 when it is 0. The Enable_Load input triggers a specified input value to be loaded by the counter. The Enable_Up and Up_Down inputs control the type of counting operation to be performed by the counter. If both inputs are high, the counter behaves as an up-counter. If the Up_Down input is high or 1 and the Enable_Up input is low or 0, the counter behaves as a down counter. The grammar designed for the N-bit Up-Down Counter is shown in Listing 16.

**Table 6.1:** N-bit Up-Down Counter Transition Table. The symbols '↑' and '↓' represent positive and negative transitions of the clock, respectively. The '?' symbol represents don't care bits.

| Clk | Reset | Enable_Load | Enable_Up | Load | Up_Down | Q (Output) |
|---|---|---|---|---|---|---|
| ↑ | 0 | ? | ? | ? | ? | 0 |
| ↑ | 1 | 1 | ? | N-bit Number | ? | Load |
| ↑ | 1 | 0 | 1 | ? | 1 | Q + 1 *(count up)* |
| ↑ | 1 | 0 | 1 | ? | 0 | Q - 1 *(count down)* |
| ↑ | 1 | 0 | 0 | ? | ? | Q *(*no change) |
| ↓ | ? | ? | ? | ? | ? | ? |

```
<design-module> ::= <begin-module> <always> <endmodule>
       <always> ::= always @(posedge clk) <nextline> <switch-case>
  <switch-case> ::= casex ({reset, en_load, enable_ud, up_ndown})
                    <nextline><case-stmt> endcase <nextline>
 <case-default> ::= default ":" "q<=" <expr>;
         <stmt> ::= "q<=" <expr>; <nextline>
         <case> ::= 4'b<bit><bit><bit><bit>
         <expr> ::= <terminal> | <terminal><op><terminal>
                  | <terminal><op><expr>
    <case-stmt> ::= <case> ":" <stmt> | <case> ":" <stmt><case-stmt>
     <terminal> ::= 0 | 1 | q | load
          <bit> ::= 0 | 1 | ?
           <op> ::= - | +
-------------------------------------------------------------------------
 <begin-module> ::= "module nbit_udcounter #(parameter width=64)
                    (input logic clk, up_ndown, en_load, enable_ud,
                    reset, input logic [width-1:0] load, output
                    logic [width-1:0] q);" <nextline>
    <endmodule> ::= endmodule
     <nextline> ::= "\n"
```

**Listing 16:** N-bit Up-Down Counter. Highlighted in blue and yellow are the start symbol and the use of parameterisation, respectively.

**Table 6.2:** Training cases for benchmark problems. $a$ and $b$ refer to addends for the adder circuit. For the multiplier circuit, $a$ and $b$ are multiplier and multiplicand, respectively. $N$ = bit-width. The Up-Down counter states 1010 and 1011, which set the counter in down-counting and up-counting modes, respectively, are exempted from the corner cases. This is because these states are tested by the general cases.

| Circuit | General | Corner Cases | General Cases | Total No. of Cases |
|---|---|---|---|---|
| N-bit Selective Parity | N=32<br>N=1024 | 1. data = 0<br>2. data = $2^N - 1$ | 48 sampled test inputs 128-bit data | 50 (2+48) |
| N-bit + N-bit | N = 32<br>N = 256 | 1. a = 0, b = 0<br>2. a = $2^N - 1$, b = $2^N - 1$<br>3. a = 0, b = N-bit num | 47 sampled test inputs between 0 to $(2^N - 1)$ | 50 $(3 + 47)$ |
| N-bit x N-bit | N = 32<br>N = 256 | 1. a = 0, b = 0<br>2. a = $2^N - 1$, b = 0<br>3. a = 0, b = N-bit num<br>4. a = 1, b = N-bit num | 46 sampled test inputs between 0 to $(2^N - 1)$ | 50 $(4 + 46)$ |
| N-bit Up-Down Counter | N = 32<br><br><br>N = 64 | 1. All 4-bit numbers (14 states of the counter) except 1010 and 1011.<br>2. An additional load state testing with a load value of 0 to distinguish between resetting to 0 and loading a 0. | 1. Put the counter in an initial state (load state)<br>2. 10 cases to test counting up and down operations. | $26(15 + 11)$ |

### 6.3.3 Testing

The selected default bit-width sizes of the parameterised circuits are high, rendering exhaustive testing infeasible. Furthermore, lexicase selection, the preferred selection method for digital circuit evolution, becomes computationally expensive with the increasing number of test cases. As a result, the training and testing cases were sampled using two strategies as shown in Table 6.2.

First, the corner test cases for each circuit problem are identified, usually targeting the test vectors located at the boundaries of all the possible test vectors (truth table) (Williams, 2014). Second, given that the corner cases have been identified, only a few samples of the remaining cases are required. Initial experiments were conducted to determine the number of training and testing cases to use. For the combinational circuits, the total number of cases is 50. The number of the remaining training and testing cases were generated by uniformly random

sampling (using `$urandom()` function in SystemVerilog) the internal test vectors within the input range of each circuit. The total number of cases for the N-bit Up-Down Counter is 26 (15 and 11 corners and remaining cases, respectively). The remaining cases target the counting up and down operations (5 cases each); a single case sets the counter in an initial state by loading a pre-specified value prior to training and testing. The use of corner cases and sampled test cases ensure our testbenches have good coverage over all possible test vectors.

The total number of cases for the Selective Parity and Adder circuits is 100. The Selective Parity circuit can behave as an odd or even parity circuit based on the `even_odd input signal`. During training and testing, both behaviours are tested. The Adder's `carry_in input signal` can be either a 0 or 1. In a similar approach, the Adder circuit is trained and tested with the same cases when the `carry_in` input signal is 0 and 1.

All obtained solutions are tested for functional accuracy. Given that the evolved solutions for the parameterised circuits cannot be tested by instantiating them with every possible bit-width size, we chose input bit-width sizes smaller and larger than the input bit-width with which the circuits were evolved/trained. Test case generation is done using the same procedure outlined in Table 6.2 for generating the training cases. For example, the Adder was tested using 32-bit + 32-bit and 256-bit + 256-bit test vectors, as shown in Table 6.2.

### 6.3.4 Evolutionary Parameters

Preliminary experiments were conducted to determine reasonable parameter values for the population size and number of generations. Based on the preliminary results obtained, 30 generations and a population size of 500 are used for the parity and adder circuits; 50 generations and 2,000 population size are used for the multiplier and Up-Down Counter circuits. All other parameters remain the same across all benchmark problems. Table 6.3 details the operators and parameters used in the experimental setup.

**Table 6.3:** Evolutionary Run Parameters.

| Parameter | Value | |
|---|---|---|
| Initialization | Sensible Initialization | |
| Selection | Lexicase Parent Selection | |
| Crossover Rate | 0.8 | |
| Mutation Rate | 0.01 | |
| Replacement Rate | 0.05 | |
| Number of Runs | 50 | |
| Generations | Parity and Adder | 30 |
| | Multiplier and N-bit Counter | 50 |
| Population Size | Parity | 200 |
| | Adder | 500 |
| | Multiplier and N-bit Counter | 2000 |
| Termination Condition | When mean, minimum and maximum fitness equals the maximum fitness or generation number equals specified maximum generation number | |

## 6.4 Results and Discussion

Results obtained from our experiments are shown in this section. Additionally, the results are compared to those obtained by state-of-the-art approaches. In summary, based on the results obtained, the use of GE and an HDL (SystemVerilog) is capable of evolving complex combinational and sequential circuits, which answers the research questions set out in Section 6.1. However, it should be noted that when compared with state-of-the-art approaches, despite our evolved circuits requiring fewer gates to realise in silicon, the optimisation of circuits evolved at an abstract level like RTL is largely dependent on the robustness of the synthesis tool.

The rest of the section discusses the evolutionary performance of each benchmark circuit, and success rate, analyses the effect of input sizes on circuit evaluation and compares our results to state-of-the-art methods.

### 6.4.1 Evolutionary Performance of Each Benchmark Circuit

In Figures 6.1, 6.2, 6.3, 6.4 and 6.5 the mean best and mean average fitnesses across the 50 independent runs conducted for each benchmark problem are plotted, in order to visualise evolutionary performance across generations.

Figures 6.1 and 6.2 show the plot for the Selective Parity Grammar A and Grammar B, respectively. Both grammars attained maximum mean fitnesses from the initial generation, revealing the triviality of evolving parity generator circuits at this level of abstraction. Fully functional Selective Parity circuits were created via initialisation without resorting to any form of evolution. The performance difference between the two grammars from these plots is that the Selective Parity Grammar B obtained lower mean average fitnesses than the Selective Parity Grammar A.

Evolution attains maximum mean best fitness within the first five generations for the adder circuit as observed in Figure 6.3. The starting mean average fitness is approximately 18% of the maximum fitness and increases rapidly to attain maximum mean fitness from generation 9 onwards. Both the mean best and mean average fitnesses for the Multiplier and N-bit Up-Down Counter progress steadily but do not attain maximum mean fitnesses. Furthermore, there exist short to non-existent error bars for the Selective Parity and Adder circuits in Figures 6.1 to 6.2 and 6.3 respectively, indicating little variability. The Up-Down Counter in Figure 6.5 has very short to non-existent error bars as the evolution progressed. For the multiplier circuit in Figure 6.4, short error bars are observed in the earlier generations and widen gradually in subsequent generations, signifying increasing variability in individual fitness values compared to the mean fitness value.

**Table 6.4:** Success Rate (Out of 50 Independent Runs)

| **Problem** | **Population Size** | **Success Rate** |
|---|---|---|
| Selective Even Parity Grammar A<br>Selective Even Parity Grammar B | 200 | 50 |
| Adder | 500 | 50 |
| Multiplier<br>Mod-N Up-Down Counter | 2000 | 23<br>24 |

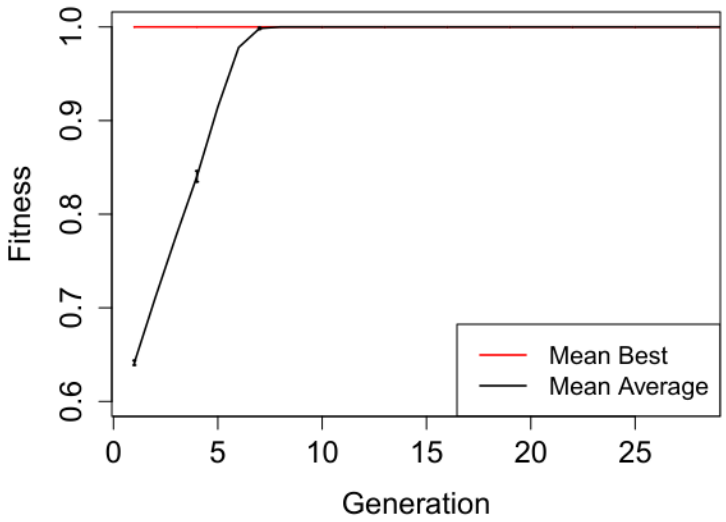


**Figure 6.1:** Mean best and mean average with error bars for N-bit Parity Generator Grammar A.

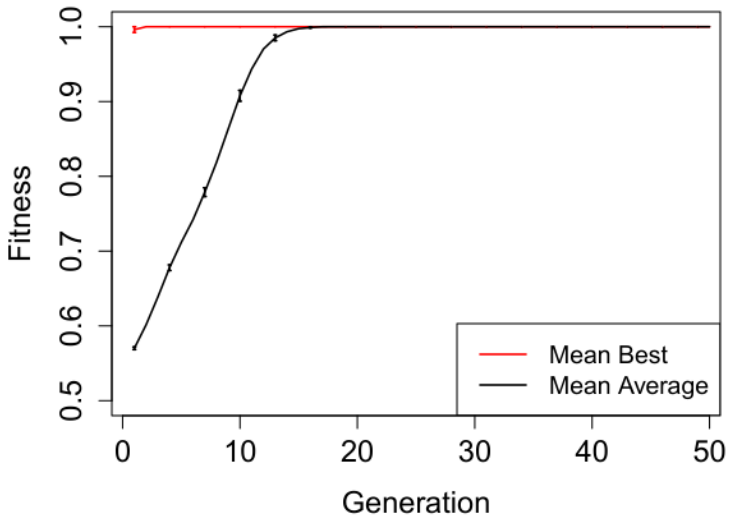


**Figure 6.2:** Mean best and mean average with error bars for N-bit Parity Generator Grammar B.

Given that it is not feasible to exhaustively test the correctness of the obtained candidate solutions, each candidate circuit was instantiated with a parameter

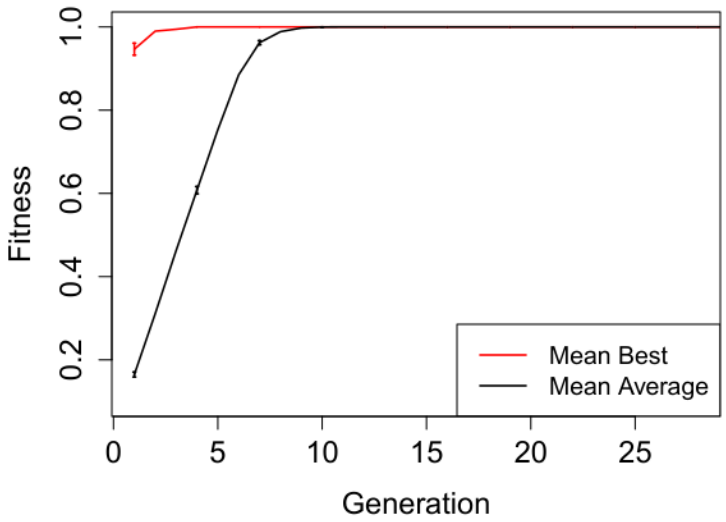


**Figure 6.3:** Mean best and mean average with error bars for N-bit + N-bit Adder.

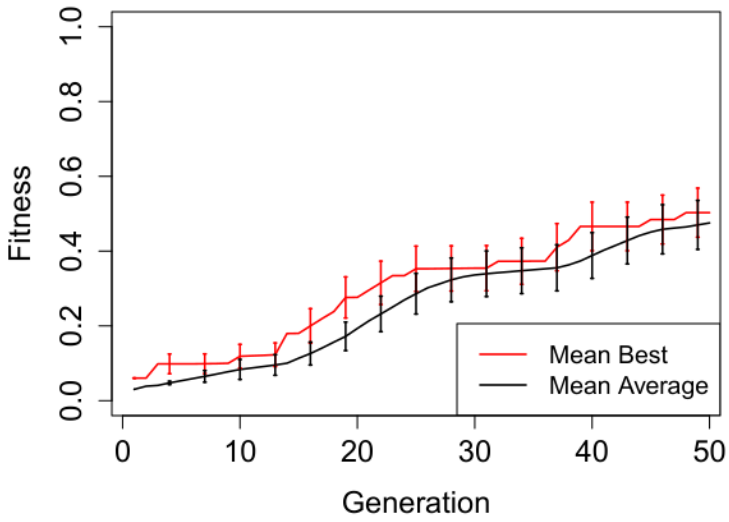


**Figure 6.4:** Mean best and mean average with error bars for N-bit × N-bit Multiplier.

value of 5, facilitating the exhaustive testing of these circuits.

To gain insight into the 23 non-optimal solutions obtained for the multiplier circuit benchmark problem, we conducted a post hoc test on each. In our post

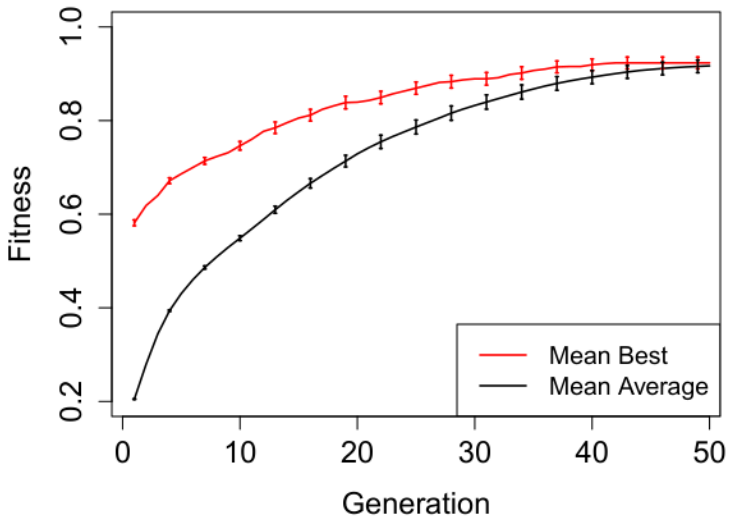


**Figure 6.5:** Mean best and mean average with error bars for N-bit Up-Down Counter.

hoc test, each non-optimal circuit was tested using the same testbench used in the evolutionary experiments. We then studied the test cases (test vectors) passed and failed by each circuit. Recall that we had two sets of test cases: corner cases and the sampled test cases within the circuit's input range.

We observed all 23 unsuccessful runs were only able to solve the 4 corner cases shown in the Table 6.4. This observation is of utmost significance as it proves sampling test vectors from the circuit's input range alone is insufficient to avoid exhaustive testing for complex combinational circuits. Corner case testing is equally important as it ensures valid inputs likely to cause circuit malfunction are explicitly tested.

### 6.4.2 Success Rate

The success rate for each problem is shown in Table 6.4. A successful run is defined as an independent evolutionary run that evolves an optimal solution.

The N-bit Selective Parity (both Grammar A and B) and N-bit + N-bit Adder circuits obtained 100% successful runs. The N-bit × N-bit Multiplier and N-bit

Up-Down Counter attained 46% and 48% successful runs respectively. Based on the population size and success rate of each circuit problem, the N-bit Selective Parity benchmark is the least difficult problem, followed by the N-bit + N-bit Adder. The N-bit × N-bit Multiplier and N-bit Up-Down Counter were the most challenging to evolve, requiring a higher population size.

### 6.4.3 Effect of Input Sizes on Experiment Duration

Equally important is the effect of increasing input sizes (bit-width) on evaluation time. In order to make this assessment, separate experiments were conducted using smaller input-width sizes: N = 32 for Adder, Multiplier and Up-Down Counter circuits; N = 64 for the Selective parity circuit as shown in Table 6.5. Wilcoxon tests were performed on each pair of simulation times. A p-value of 0.05 was chosen as the significance level, meaning any test returning a value below that level would be deemed significant and the difference runtime between each pair would be statistically significantly different.

Increasing bit-width sizes of Selective Parity Grammar A, N-bit + N-bit Adder and N-bit × N-bit Multiplier circuits has a significant effect on the evaluation time of circuits. In the case of Selective Parity Grammar A, recall bitwise operators together with a `generate for-loop` were used; as a result, it requires a minimum of *width* evaluation events each time an input changes to generate the parity. However, Selective Parity Grammar B uses reduction operators that take a vector (multi-bit input), apply the respective operation to each individual bit and return a single bit as output. Reduction operators execute in a single evaluation event. Therefore, the use of reduction operators takes significantly less time to simulate Selective Parity circuits created with Grammar B. The Adder performs more additions as the input bit-width sizes increase. Similarly, the multiplier performs several additions to sum up the partial products. As a consequence, there is a significant increase in evaluation time as the input bit-width size increases for these circuits. In the case of the N-bit Up-Down Counter, no significant difference is observed because we did not exhaustively test the counting up and down operations over the entire input range. However, we do observe a slight

difference in the mean experiment duration, signifying that the increase in input bit-width size does have some effect on the experiment duration.

Based on these observations, a careful selection of operators to be incorporated into grammars is suggested. Furthermore, selecting a reasonable default input bit-width size when evolving parameterised circuits can potentially reduce the experimental time. For example, an input bit-width size of 10 is reasonable for evolving a parameterised adder compared to a bit-width size of 1. A 1-bit + 1-bit Adder has only four test cases, which may not be sufficient to train a parameterised Adder properly.

**Table 6.5:** Effect of Input Size on Runtime. In bold are statistically significant.

| **Circuit Problem** | **Population Size** | **No of Runs** | **Input Size** | **Mean Duration per Run** (seconds) | **P-value** |
|---|---|---|---|---|---|
| *N-bit* Selective Parity (Grammar A) | 200 | 50 | 64<br>128 | 589.6141<br>8689.829 | **2.2e-16** |
| *N-bit* Selective Parity (Grammar B) | 200 | | 64<br>128 | 15.02446<br>15.16393 | 0.4841 |
| *N-bit* + *N-bit* Adder | 500 | | 32 + 32<br>64 + 64 | 95.41906<br>200.4862 | **2.2e-16** |
| *N-bit* x *N-bit* Multiplier | 2000 | | 32 x 32<br>64 x 64 | 844.9755<br>888.784 | **0.005294** |
| N-*bit* Up-Down Counter | 2000 | | 32<br>64 | 864.0152<br>875.6796 | 0.5418 |

### 6.4.4 Comparison With Literature

In Table 6.6, a representative solution for each benchmark problem was synthesised to obtain the respective number of gates required to realise each circuit functionality in silicon. Given that all evolved circuits are parameterised, different circuit instances with different bit-width sizes were created for each and synthesised as shown in Table 6.6. We use Yosys (Wolf, n.d.). Yosys is an Open Source framework for RTL synthesis which has an extensive support for Verilog-2005. It can be easily configurable with GE and has basic synthesis algorithms sufficient for our use case. These results (gate counts in Table 6.6 and mean execution time in Table 6.5) are compared to the most complex circuits evolved in the literature in Table 6.7.

**Table 6.6:** Number of logic gates required to realise a representative solution of each benchmark circuit evolved. D-FF represents D flip-flop. In bold are the circuit instances used for comparing the gate count of our evolved circuits to those evolved by state-of-the-art approaches in Table 6.7.

| **Circuit** | **Circuit Instance** | **Total Gates** | **Respective Gate Counts** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | **AND** | **OR** | **NOT** | **NAND** | **NOR** | XNOR | XOR | D-FF |
| *N-bit* Selective Parity | 4 | 4 | - | - | - | - | - | 2 | 2 | - |
| | 6 | 6 | - | - | - | - | - | 6 | - | - |
| | 8 | 8 | - | - | - | - | - | 6 | 2 | - |
| | 10 | 10 | - | - | - | - | - | 8 | 2 | - |
| | 12 | 12 | - | - | - | - | - | 10 | 2 | - |
| | 16 | 16 | - | - | - | - | - | 14 | 2 | - |
| | **19** | **19** | - | - | - | - | - | **13** | **6** | - |
| | 20 | 20 | - | - | - | - | - | 14 | 6 | - |
| | 32 | 32 | - | - | - | - | - | 24 | 8 | - |
| | 64 | 64 | - | - | - | - | - | 54 | 10 | - |
| | 128 | 128 | - | - | - | - | - | 100 | 28 | - |
| *N-bit* + *N-bit* Adder | $2+2$ | 10 | - | - | - | 6 | - | - | 4 | - |
| | $3+3$ | 15 | - | - | - | 9 | - | - | 6 | - |
| | $4+4$ | 21 | - | 1 | - | 13 | - | 1 | 6 | - |
| | $5+5$ | 26 | - | 1 | - | 16 | - | 1 | 8 | - |
| | $6+6$ | 31 | - | 1 | - | 19 | - | 1 | 10 | - |
| | $8+8$ | 41 | - | 1 | - | 25 | - | 1 | 14 | - |
| | **10+10** | **52** | **3** | **2** | - | **29** | - | **2** | **16** | - |
| | $16+16$ | 87 | 21 | 7 | - | 34 | - | 7 | 18 | - |
| | $32+32$ | 175 | 45 | 15 | - | 66 | - | 15 | 34 | - |
| | $64+64$ | 369 | 38 | 34 | 1 | 194 | 4 | 30 | 68 | - |
| *N-bit* × *N-bit* Multiplier | $2\times2$ | 8 | 4 | - | - | 1 | 2 | 1 | - | - |
| | $3\times3$ | 35 | 11 | 4 | 1 | 14 | - | 4 | 1 | - |
| | $4\times4$ | 86 | 18 | 13 | 3 | 36 | 1 | 12 | 3 | - |
| | **5 × 5** | **140** | **27** | **23** | **3** | **59** | **1** | **20** | **7** | - |
| | **6 × 6** | **220** | **39** | **38** | **3** | **95** | - | **35** | **10** | - |
| | $8\times8$ | 423 | 77 | 76 | 6 | 170 | 1 | 75 | 18 | - |
| | $10\times10$ | 702 | 133 | 134 | 9 | 266 | 3 | 137 | 20 | - |
| | $16\times16$ | 1,916 | 387 | 412 | 16 | 655 | 6 | 398 | 42 | - |
| | $32\times32$ | 8,101 | 1,600 | 1,794 | 46 | 2,751 | 12 | 1,771 | 127 | - |
| | $64\times64$ | 32,674 | 7,175 | 7,681 | 60 | 9,878 | 8 | 7,451 | 421 | - |
| Mod-*N* Up-Down Counter | Mod-4 | 66 | - | - | 11 | 34 | 17 | - | - | 4 |
| | Mod-8 | 144 | - | - | 18 | 56 | 62 | - | - | 8 |
| | Mod-16 | 280 | - | - | 32 | 115 | 117 | - | - | 16 |
| | Mod-32 | 550 | - | - | 57 | 224 | 237 | - | - | 32 |
| | Mod-64 | 1,104 | - | - | 112 | 458 | 470 | - | - | 64 |
| | Mod-128 | 2,264 | - | - | 271 | 745 | 1,120 | - | - | 128 |

Comparing the number of gates required to realise the respective circuits evolved by state-of-the-art approaches in Table 6.7 to our evolved circuits in Table 6.6, GE-evolved circuits require fewer gates. Similarly, less simulation time is required to evolve circuits (in Table 6.5) compared to state-of-art approaches in Table 6.7.

Since an example of an N-bit Up-Down Counter being evolved was not found in literature, in order to make a comparison to the 3-bit and 4-bit Up Counters

in Table 6.7, the evolved Up-Down Counter was modified to disable the count down operation before synthesis. The synthesised 3-bit requires 38 gates and 3 D-FFs; the 4-bit Up Counters require 46 gates and 4 D-FFs. Comparing these gate and D-FF counts to the number of gates and D-FFs to the Up Counter in Table 6.7, our resultant Up Counters use less number of D-FFs but require more gates. The number of D-FFs is consistent with the rule of thumb that an N-bit Counter requires an N number of flip-flops. However, our Up Counter comparison isn't entirely fair as no specification on the number of inputs or transition table is provided for the evolved 3-bit and 4-bit Up Counters in (Henson et al., 2018) since they affect gate count.

**Table 6.7:** Most Complex and Fully Evolved Functional Parity, Adder, Multiplier and Counter Circuits previous work.

| Circuit | | Inputs | Min Number of Gates | Mean Execution Time | Approach |
|---|---|---|---|---|---|
| Combinational | **Parity** | 19 | NA | 13,801s | CGP + Decomposition (SDR) (Li et al., 2008) |
| | **Adder** | 10 + 10 | 78 | 1,326s | CGP + semantic mutation (Hodan et al., 2020) |
| | **Multiplier** | $5 \times 5$ | 3,671 | 2,556 | CGP + semantic mutation (Hodan et al., 2020) |
| | | | NA | 548 core-hours single run (Hodan et al., 2020) | Parallel CGP Implementation (Hrbacek and Sekanina, 2014) |
| | | $6 \times 6$ | NA | 364,752s | GA + Decomposition (GDD + BIE) (Stomeo et al., 2006*b*) |
| Sequential | **Up Counter** | 3 | 39 (32 gates and 7 D flip-flops) | 5hr 36min 12s | CGP + VHDL Translation System + Timing Analysis (Single Run) (Henson et al., 2018) |
| | | 4 | 39 (26 gates and 13 D flip-flops) | 2days 20hr 13min 28s | |

## 6.5 Conclusion

In this chapter, we demonstrated that using RTL, a higher level of abstraction than the gate level, enables the design of much larger circuits compared to methods previously discussed in the literature. Fully functional N-bit selective parity, N-bit + N-bit adder, N-bit × N-bit multiplier and N-bit Up-Down counter circuits have been successfully evolved, representing the most complex circuits evolved in literature to date. The availability of expressive programming constructs and operators in SystemVerilog aided greatly in the evolution of these circuits. Furthermore, our approach recorded better mean execution time and required less number of gates after synthesis compared to state-of-the-art results.

The evolved circuits are parameterised, meaning once a circuit is evolved, different circuit instances of varying input sizes can be obtained simply by specifying the desired input sizes without requiring a re-run of experiments compared to existing methodologies. The work also established how an appropriate choice of operators can significantly reduce simulation time in the case of the selective parity benchmark. Also, using reasonable input bit-width sizes to evolve parameterised circuits generally reduces the evaluation time while the evolutionary performance remains the same.

# 6. EVOLUTION OF COMPLEX PARAMETERISED DIGITAL CIRCUITS IN SYSTEMVERILOG

# 7

# Evolving Multi-Output Digital Circuits Using Multi-Genome Grammatical Evolution

## 7.1 Introduction

ADDC has been used to evolve circuits of varying complexities in the preceding chapters. However, many circuits are characterised by different numbers of outputs (for example, an adder circuit), which need to be simultaneously evolved. In this work, a single genome has been the representation structure for all circuit benchmark problems that evolved up until this point. Speculatively, the use of a multi-genome representation for circuit problems (applicable to non-circuit problems such as multi-label classification problems) could significantly improve the performance of GE on multi-output problems.

The aims of this chapter are summarised as follows:

- Propose a multiple genome implementation of GE better suited for tackling multi-output problems called MGGE;

- Adapt genetic operators, initialisation routine and mapping mechanism, and implement a wrapping operator well-suited for MGGE;

- Introduce custom grammar syntax rules and a cyclic dependency-checking algorithm that facilitates the evolution of inter-output dependencies;

- Investigate the performance of MGGE with and without the use of our custom grammar syntax rules on multi-output combinational circuit problems.

## 7.2 Overview

A major benefit of GE's mapping process is the separation of the genotypic space from the phenotypic space, which facilitates an unconstrained evolutionary search (Ryan et al., 1998). While there has been some suggestion that the low locality, i.e., the measure of the correlation between neighbouring genotypes and their corresponding neighbouring phenotypes, brings about an issue (Rothlauf and Oetzel, 2006), other works (Castle and Johnson, 2010; Nicolau and Agapitos, 2018; O'Neill and Ryan, 2000; O'Neill et al., 2003) suggest genetic operations at the beginning regions of GE's genome are destructive but may serve as a good region for genetic operations.

Generally, a multi-output problem requires modelling complex input-output relationships and/or inter-output dependencies. Multi-output problems requiring the simultaneous modelling of these relationships are very challenging to deal with (Zhen et al., 2018). Hence, decomposing such problems into single-output problems and solving each using a single-output algorithm of choice may prove challenging as such approaches do not consider inter-output correlations (Zhen et al., 2018). This work proposes an MGGE for solving multi-output or multi-target problems, such as circuit problems, efficiently. First, each genome encodes a solution to a single and unique output. Second, custom grammar syntax rules are introduced to facilitate the evolution of inter-output correlations by evolution. Third, the use of a multi-genome should help reduce the destructive effect of genetic operators through the use of a genome per output variable while benefiting from their ability to escape local optima. Fourth, MGGE keeps track and stops the search for a solved output variable which is made available to all individuals in the population, thereby reducing the overall computational cost as evolution progresses. Finally, MGGE is better suited for multi-output circuit problems than

approaches that decompose such problems into single-output circuit problems using various decomposition techniques before merging the evolved sub-circuits into a complete circuit.

## 7.3 Multi-Output Problems

Multiple output or target variables characterise multi-output (also known as multi-target or multi-variate) problems. Multivariate regression and multi-label classification are categories of multi-output problems. Multi-variate regression deals with the estimation of a single regression model, which models how multiple independent and dependent variables are linearly related.

Multi-variate regression is used in different domains such as economics and finance, health care, social sciences, environmental studies and market research. For example, in economics and finance multi-variate regression is used to investigate the factors that affect inflation rates, stock prices, housing prices and GDP growth rate. In multi-label classification, zero or more labels are required as output for each input sample Osojnik et al. (2017). Gene classification in bio-informatics, text categorisation and image classification are real-world applications of multi-label classification. These problems are challenging to deal with, mainly due to their multi-variate nature and possible dependencies between target variables (Borchani et al., 2015).

Methodologies designed for tackling multi-output regression problems are categorised into two groups: *problem transformation* and *algorithm adaptation* methods (Borchani et al., 2015). Problem transformation methods transform multi-output problems into single-output problems, each uniquely solving one of the target variables using any single regression algorithm of choice (Borchani et al., 2015). Algorithm adaptation methods adapt existing single-target algorithms to be capable of handling multi-target problems (Borchani et al., 2015). Some advantages of the latter approach are better performance, better representation and interpretability and computationally more efficiency compared to single-target methods (Borchani et al., 2015).

### 7.3.1 Related Work

Several approaches have been proposed to deal with the issue of low locality inherent in standard GE as discussed in Section 7.2—these range from the implementation of new mappers to the use of different genome representation structures.

In (Akira et al., 2008), a Multi-chromosomal GE (MCGE) is presented, which employs multiple genomes as the number of non-terminals in a grammar to mitigate the low locality inherent in standard GE. To map a non-terminal, its corresponding genome of a non-terminal is used. Similarly, crossover operations are applied to genomes of matching non-terminals. As a result, this increases locality by retaining the sentences of non-terminals whose genomes were unaffected by genetic operations.

Similarly, Structured GE (SGE) employs a genotype of multiple chromosomes, each corresponding to a unique non-terminal. However, unlike MCGE, SGE does not support unconstrained recursive grammars; instead, intermediate symbols that mimic a pre-defined maximum level of recursions must be pre-defined. Each gene consists of values corresponding to the expansion choices of its corresponding non-terminal. The gene length is determined by computing the maximum number of expansions of a non-terminal, after which it is filled randomly with values in the interval $[0, c_N]$, where $c_N$ is the number of derivation choices for a non-terminal. As a result, SGE does not use the modulo rule as used in standard GE and MCGE.

In (Medvet, 2017), two novel mapping procedures are introduced: Hierarchical GE (HGE) and its variant, Weighted HGE (WHGE), which aim to address the issue of poor locality in standard GE. This paper conceptualises the phenotype as a hierarchy of non-terminal symbols. In HGE, the closer a non-terminal is to the root of the hierarchy, the larger the portion of the genotype it is allocated. After a non-terminal is expanded, its associated genotype substring is further subdivided among its descendant non-terminals. Weighted HGE differs from HGE in that the size of the genotype substring allocated to each non-terminal is proportional to its expressive power. Given that larger genotype substrings are allocated to non-terminals close to the root of the phenotype, it is less likely altering a single bit will alter the choice of expansion. In contrast, for nodes situated farther from

the root, which are assigned smaller genotype substrings, a single bit change has a greater likelihood of affecting the expansion decision.

Unlike the above GE multi-chromosomal approaches, which seek to address the issue of GE's low locality, our approach MGGE also adapts standard GE to use a multi-chromosomal representation better suited for multi-output or multi-target problems as it mitigates the issue of locality. In the above GE-variant approaches, where there is a gene or chromosome per non-terminal, MGGE uses a chromosome per output variable. Moreover, the above methods were not adapted to better handle multiple outputs, as is the case in MGGE (MGGE is described in detail in Section 7.4).

Digital circuit designs, such as adders and multipliers require multiple output signals to be correct for their output to be accurate. Consequently, these sort of problems can benefit from a GE variant tailored to better handle multi-output problems. Some of these output signals can be independent of one another, meaning that some output signals do not require other output signals as inputs (or as building blocks) to generate their respective output. In such a scenario, only a sub-section of an individual needs to be modified. These sub-modifications can be difficult for a classical GA with a single chromosome to perform as the genetic operators do not know how the modifications will affect the candidate design.

Multiple chromosomes have been used in various evolutionary hardware problems to tackle the evolution of circuits. CGP is often used with gate-level evolutionary hardware problems as its column and row structure is ideal for connecting Boolean logic gates. Multiple codons in the chromosome are consumed when choosing a gate and its connections, which can lead to issues when performing crossover, as different gates can have a different number of connections. (Slowik and Białko, 2006) address this by using a multi-chromosome genome that groups the logic gates with their input connections. This allows for the crossover method only select crossover points that would not produce an unconnected input and thus produce only valid individuals.

(Walker et al., 2009) introduced a method called Multi-Chromosome CGP (MC-CGP) for Evolutionary Hardware. This approach uses a multi-chromosome where each chromosome is used in a separate CGP evolution to evolve a partial

solution for a single output. When a solution is found for each output, they are then combined to produce the final solution. This approach allows the evolution to be performed in parallel but can produce redundant circuitry. To address this problem, Coimbra and Lamar (2016) used a hybrid approach of running an MC-CGP followed by a standard CGP where the multi-chromosome is treated as a single chromosome. This approach allowed them to reduce the gate count of the evolved individuals significantly.

CGP is generally only employed at the gate level and thus is unable to enjoy the benefit of HDLs. Work that has looked at higher-level problems includes a multi-objective approach that was taken to evolve a Proportional-Plus-Derivative fuzzy logic controller. In this work, Baine (2008) used four separate chromosomes to describe the input and output fuzzy sets for both the proportional and derivative parts of the controller. He found that the multi-chromosome approach converged around 20% faster compared to a single-chromosome approach.

In (Reyes et al., 2018), standard Gene Expression Programming (GEP) is adapted to directly handle multi-target regression problems without resorting to any transformation or decomposition approaches. A multi-gene chromosome representation is adopted. An $N$-target regression problem requires a chromosome with $N$ genes. Each gene codes a solution to a single and unique target variable. Hence, the number of genes contained in a chromosome is problem-dependent. Individual initialisation remains the same as in standard GEP but is subject to a metric that ensures an individual is only added to the initial population if its similarity in comparison to other individuals in the population is below a threshold. Three different transposition operators are used to transpose gene fragments within the same gene, between two genes (of the same chromosome) and move a gene to the beginning of the chromosome. Also, three recombination operators are used to create offspring. Tested on eight multi-target regression datasets in comparison to two other methods, Gene Expression Programming for Multi-target Regression (GEPMTR) significantly outperformed the two state-of-the-art methods in seven of the eight multi-target datasets employed.

These works show that using a multi-chromosome approach for multi-target problems can significantly help in finding solutions. In addition, it helps address the issue of locality that GE can experience.

## 7.4 Proposed Approach: Multi-Genome GE

### 7.4.1 Problem Description

As discussed in Section 2.3.2, the GE mapping process starts from a start symbol. For the purpose of explanation, Figure 2.4 is repeated in Figure 7.1. Usually, the first rule's non-terminal is designated as the start symbol; hence, in Figure 7.1, ⟨expr⟩ is the start symbol. GE iteratively expands or rewrites all non-terminals by using a modulus rule to select a production each time a choice needs to be made. This process continues until the sentence consists solely of terminals. In Figure 7.1, using the sample genome provided, the sentence $x + 5$ is obtained.

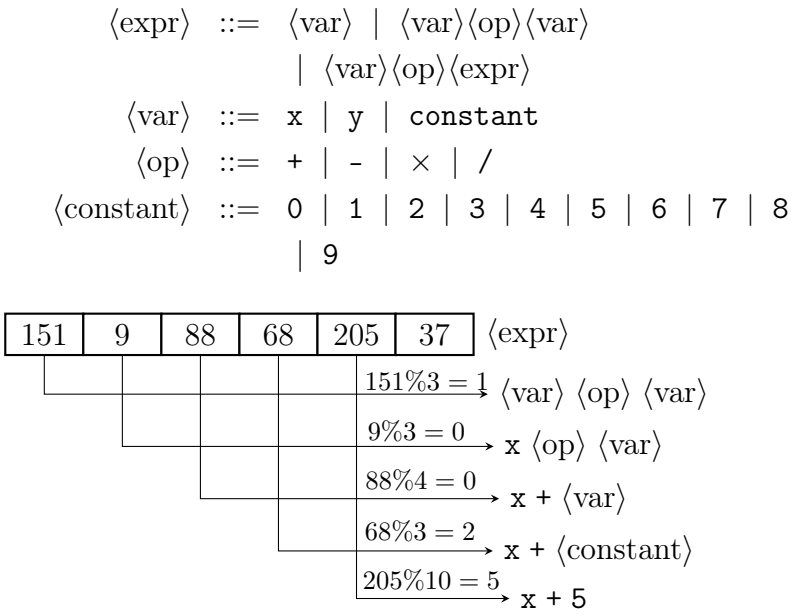


**Figure 7.1:** Grammatical Evolution Mapping Mechanism.

Assuming a mutation event affects the first codon, changing its value from 151 to 150, this means the first production ⟨var⟩ gets chosen after applying the mod rule ($150\%3 = 0$). As a result, the resultant sentence is now $x$ after the mutation event. As described, a totally different phenotype is obtained after a simple codon alteration in GE's genome, with all potentially evolved building

blocks lost after genetic operations. This phenomenon is referred to as the *ripple effect* (O'Neill et al., 2003).

As mentioned earlier, multi-output problems require all outputs to be simultaneously evolved. Assuming a standard GE individual (single genome representation) solves two of the three output variables, it may be challenging to solve the third due to the ripple effect caused by genetic operators. It is also computationally expensive as there is no mechanism in place to stop searching for output variables which other individuals in the population have successfully evolved. Thus, standard GE does not have the necessary functionality to evolve inter-target dependencies properly.

### 7.4.2 Problem Formulation

Assuming a combinational circuit denoted as $C$ has multiple inputs and outputs, the training/testing vectors for $C$ can be denoted as $\{X_1 \cdots X_m\}$ and $\{Y_1 \cdots Y_n\}$ representing inputs and expected outputs respectively. Each circuit output in $\{Y_1 \cdots Y_n\}$ can depend on one to $m$ input variables ($\{X_1 \cdots X_m\}$) as well as from zero to $n-1$ circuit outputs $\{Y_1 \cdots Y_n\}$; as far as no cyclic dependencies are formed between output variables in the case of combinational circuits.

Given that multi-output circuit problems (and multi-output problems in general) generally exhibit a more complex behaviour compared to a single-output circuit problem, evolving such circuits, especially at a low level of design—*gate-level*, using standard GE is intractable.

### 7.4.3 Multi-Genome GE

MGGE uses a multiple genome representation consisting of $n$ genomes, where $n$ represents the number of outputs or targets. Each genome evolves a circuit functionality satisfying a unique circuit output. An example of a multi-genome for a multi-output problem with three output variables or target variables is shown in Figure 7.2. An output or target variable is a variable to be predicted by other variables.

A new genome representation requires suitable initialisation routines, mapping mechanisms, selection, replacement, wrapping mechanisms and genetic operators.

Furthermore, new grammar design specifications are introduced, which must be adhered to for proper parsing of the grammar to ensure the adapted initialisation routine and mapper algorithms work properly. These adapted and new operators are described in detail in subsequent sections.

| | | | | |
|---|---|---|---|---|
| 234 | 32 | 112 | | |
| 37 | 65 | 43 | 189 | 222 |
| 17 | 105 | 243 | 149 | |

**Figure 7.2:** A multi-genome representation for a multi-output problem with three output variables.

### 7.4.4 Grammar Design

In MGGE, each genome is mapped to its output expression or subprogram to model and predict its corresponding output variable. MGGE requires a single grammar, just like standard GE. However, custom grammar format specifications have been introduced and must be adhered to. There are two main custom grammar specifications: *output rule* and *output variable rule*. These rules are illustrated by a sample grammar in Listing 17. These grammar format specifications allow for grammar rules to be annotated with required details, which are parsed and used by the mapper, wrapper, selection, initialisation and genetic operators.

An *output rule* (or *target rule*) is a rule that codes a solution to one of the output variables in a multi-output problem. To define an output rule, the name of the rule must be preceded by *tr* followed by an output group number. During the mapping process, whenever such a rule is encountered, a sub-mapping process is spawned. For example, in Listing 17, there are four output rules: ⟨tr1-output-1⟩, ⟨tr1-output-2⟩, ⟨tr1-output-3⟩ and ⟨tr1-output-4⟩. Also, in Listing 17, there are two groups of output variables: *1* and *2*. In other words, there should be an output variable rule for each output group number as illustrated in Listing 17.

The purpose of the group numbers is to facilitate modelling dependencies between output variables while avoiding cyclic dependencies, which is key for the evolution of combinational circuit designs. Recall in Section 7.4.2; as stated

```
       <stmts> ::= <tr1-output-1><tr1-output-2><tr1-output-3>
                   <tr1-output-4>
<tr1-output-1> ::= output_1 "=" <expr>;
<tr1-output-2> ::= output_2 "=" <expr>;
<tr2-output-3> ::= output_3 "=" <expr>;
<tr2-output-4> ::= output_4 "=" <expr>;
 <tv1-outputs> ::= output_1 | output_2
 <tv2-outputs> ::= output_3 | output_4
        <expr> ::= <terminal> | <terminal><op><expr>
    <terminal> ::= a | b | <tv1-ouputs> | <tv2-outputs>
          <op> ::= & | ^ | ~^
```

**Listing 17:** MGGE Sample Grammar. A rule preceded by tr is a target or output rule which codes a solution to one of the output variables of a multi-output problem. A rule preceded by tv is an output variable rule whose productions are output variables (terminals).

earlier, there might exist dependencies between outputs in some multi-output problems; hence, the group number allows outputs (represented by the output variable names as productions of output variable rules) to be grouped to allow evolution to evolve the potential dependencies which may exist between outputs belonging to the same group. For example, in combinational circuit problems, there may exist dependencies between outputs, but there is the need to ensure there exist no cyclic dependencies between these outputs. Failing to do so will result in circuit synthesis tools synthesising memory elements to store past/present output values, which render such a circuit a sequential circuit.

Illustrated in Listing 18 is an example of a program with an existing cyclic dependency. Assuming the program in Listing 18 is a combinational circuit design in SystemVerilog, both outputs depend on each other, which will cause memory elements to be synthesised by circuit synthesis tools, rendering the circuit a sequential circuit instead of a combinational circuit. The cyclic dependency checker in Algorithm 3 is used to ensure that during the initialisation and mapping processes, such cyclic dependencies do not occur.

An output variable rule is defined by preceding the outputs group name by *tv* followed by the output group number (e.g. ⟨tv1-outputs⟩ and ⟨tv2-outputs⟩). Essentially, an output variable rule's productions are output variables (e.g. `output_1` and `output_2`) as shown in Listing 17. Also, note that the exact order in which

```
...
    output_1 = a + output_2;
    output_2 = output_1 + b
...
```

**Listing 18:** A sample program segment with cyclic dependency.

output rules are defined for each corresponding output variable must be followed when defining an output variable rule as observed in Listing 17. An output variable rule is the means by which evolution evolves dependencies which may potentially exist between outputs. During the initialisation and mapping process of each output rule (in other words, when an output variable is being modelled), the cyclic dependency checker algorithm ensures only choices of output variables which do not cause cyclic dependency can be chosen whenever an output variable rule non-terminal is encountered. The cyclic dependency checker algorithm is described in Section 7.4.7.

### 7.4.5 Initialisation

The initialisation of the multi-genome representation is dependent on the type of initialisation scheme required. In the case of random initialisation, each genome is randomly initialised. In other words, $n$ number ($n$ equals the number of outputs) of random initialisation events. However, SI is preferred, as it creates diverse and valid individuals, as discussed in Section 2.3.3.1. To benefit from SI's desirable properties, SI is adapted for use with the multi-genome representation. First, SI requires the grammar to be labelled. Each production of a rule is required to be labelled with the minimum depth to expand all non-terminals to terminals fully and whether or not the production is recursive. Each rule is then labelled recursive if any of its productions is recursive, the rule's minimum depth equals the minimum of the minimum depth of its productions. During initialisation, SI applies grow and full methods when constructing the derivation tree. When applying the Grow method, any production with a minimum depth less than the remaining depth is eligible for selection. The remaining depth is obtained by subtracting the current depth from the specified maximum depth. The full method

behaves similarly, except recursive productions are preferred if the remaining depth can accommodate it.

During MGGE SI, $n$ genomes in the multi-genome will require $n$ SI events. The start symbol for each SI event is an output rule. The order of definition of the output rules in the grammar determines which genome to use during initialisation. The first genome is initialised by fully mapping the first output rule encountered, including any other rules occurring before the first output rule. The $(n-1)th$ genome is initialised when the $(n-1)th$ rule is encountered. The $nth$ genome is initialised with codons used to completely map the $nth$ output rule encountered and any remaining rules appearing after the $nth$ output rule. However, during the initialisation of an output rule, if an output variable rule is encountered its corresponding dependency graph must be checked using the cyclic dependency checking algorithm to ensure only production choices not resulting in cyclic dependency are valid for selection. Recall from Section 7.4.4 that productions of output variable rules are essentially output variable names. If no output variables are valid, input variables are selected.

### 7.4.6 Multi-Genome Mapping

A multi-genome consists of $n$ genomes, where $n$ equals the number of output variables that must be simultaneously predicted. Given that there are $n$ genomes, the mapping process involves $n$ sub-mapping processes. Each genome encodes a solution to a single and unique output or output variable. As mentioned in Section 7.4.4, the multi-genome mapper requires a rule to be defined for each output variable, termed an output rule. Output rule non-terminals serve as the start symbols for the sub-mapping processes. The order of output rule definitions dictates which genome to use during the sub-mapping process. The MGGE mapping algorithm is shown in Algorithm 1.

The algorithm uses the first genome in the multi-genome to map the first output rule encountered during the mapping process. In other words, the order of definition of the output rules dictates which genome in multi-genome to use. Thus, the order of rule definitions in the grammar is very important to aid the

Algorithm 1: Multi-Genome GE Mapper

```
1  Require genome, grammar, wrap_operator, max_wraps;
2  Ensure multi_genome.size() > 1;
3  Function Map (multi_genome, grammar, wrap_operator, max_wraps)
4      genome.phenotype_valid ← true ;
5      genome_index ← 0;
6      genome ← multi_genome.genomeAt(index) ;
7      if genome is empty then
8          genome.phenotype ← empty string;
9          genome.phenotype_valid ← false;
10         genome.effective_size ← 0;
11     mapped_output_rules ← declare set;
12     //stores target rules that have been mapped;
13     phenotype ← empty string;
14     start_symbol ← grammar.getStartSymbol();
15     // start symbol for SubMap dependency_manager ← instantiate
         dependency_manager object ;
16     SubMap(start_symbol, mapped_output_rules, multi_genome, genome_index,
         phenotype, dependency_manager) ;
17     multi_genome.phenotype ← phenotype;
```

mapper to generate valid phenotypes (programs or circuits in this context). Additionally, if any non-terminals are present before the first and last output rules during the mapping process, codons are consumed from the initial genome. Also, an output rule non-terminal must be used only once in the grammar.

MGGE mapping process starts off by using the first genome, as can be seen on line 6 in Algorithm 1. Lines 7-10 check if the genome contains any codons and terminates the mapping process if empty, as this indicates an invalid individual. Otherwise, if the genome contains codons, the mapping process continues as normal. The sub-mapping process is spawned as shown on line 16 in Algorithm 1, which maps the first output rule encountered, including all prior non-terminals, as stated in the previous paragraph. The sub-mapping process algorithm is shown in Algorithm 2. The sub-mapping algorithm is the same as the standard GE mapping process, with additional logic to accommodate the multi-genome representation. The algorithm is recursive as it spawns a sub-mapping process whenever it encounters an unmapped output rule non-terminal by checking the set of `mapped_output_rules`. Otherwise, it continues the mapping process consuming codons from the first genome (lines 12-17 in Algorithm 2). Lines 20-54 remain

the same as standard GE mapping where symbols belonging to selected rule production are placed on a stack for subsequent derivation steps. Wrapping events are checked and applied accordingly.

### 7.4.7 Cyclic Dependency Checking Algorithm

Algorithm 3 shows the pseudo-code for the cyclic dependency checking. A 2D matrix is used as a graph to model the dependencies between each group of output variables. For each position $i, j$ of matrix A, if $A[i, j] = 1$, it means that the output variable represented at position $i$ depends on the output variable at position $j$. The function `UpdateGraphMatrix` is a recursive function, which has the inputs `M`, a square matrix with shape $n \times n$, and `pos`, the respective position to be updated. This function assigns 0 to the position `pos` in the matrix `M`, but this assignment triggers subsequent updates in its consequence. Subsequent calls of this function are recursive to ensure the necessary updates are made to prevent cyclic dependencies between output variables. For example, if output variable `a` depends on output variable `b`, then output variable `b` cannot depend on output variable `a`. Subsequently, any output variable which depends on output variable `b` cannot depend on output variable `a`.

The main function `FillMatrix` initialises an empty matrix $M$ with shape $n \times n$. Since the variables cannot depend on themselves, the positions $i, j$, where $i = j$, are initialised with 0. The input `dependencies` is a list with the respective dependencies to be filled in the matrix. In the next step, the positions regarding each dependency are filled with 1 (if possible), and the updates in consequence of that are made by calling the function `UpdateGraphMatrix`.

### 7.4.8 Perfect Wrapping

Given that multiple output variables are being evolved simultaneously, a partially mapped output rule will render the entire individual invalid, negatively impacting the evolutionary performance. This means performing genetic operations in MGGE will produce higher invalid individuals than standard GE. In other words, the higher the number of output variables in a multi-output problem, the higher the number of invalid individuals created by genetic operations.

## Algorithm 2: Output Rule Sub-Mapping

```
1  Function SubMap (start_symbol, mapped_outputs_rules, multi_genome, genome_index, phenotype,
     dependency_manager)
2      genome ← multi_genome[index];
3      {derivation_stack, is_wrapping, codon_index, effective_size } ← {stack, false, 0 , 0} ;
4      if genome is empty then
5          return
6      derivation_stack ← push start_symbol ;
7      while derivation_stack not empty do
8          current_symbol ← derivation_stack.top();
9          derivation_stack.pop();
10         if current_symbol is TERMINAL then
11             phenotype ← append symbol ;
12         else if rule is OUTPUT_RULE and rule ∉ mapped_output_rules then
13             mapped_output_rules ← insert output_rule;
14             if mapped_output_rules.size() > 1 then
15                 SubMap(current_symbol, mapped_output_rules, genome, mapped_outputs.size()-1,
                     phenotype, dependency_manager) ;
16             else
17                 goto Continue_Mapping_Process from line 19;
18         else
19             Continue_Mapping_Process:
20             rule ← grammar.GetRule(current_symbol);
21             production_choice ← 0;
22             if rule.productions.size() > 1 then
23                 if codon_index == genome.length()-1 then
24                     codon_index ← 0;
25                     is_wrapping ← true
26                 eligible_prods ← create vector to hold eligible productions
27                 if not is_wrapping then
28                     if current_rule == OUTPUT_RULE_VAR then
29                         check if a dependency graph exists for this rule group else create one
30                         eligible_prods ← retrieve valid output vars from dm
31                         prod_choice ← genome[codon_index] % current_rule.prods.size();
32                         if prod_choice not in eligible_prods then
33                             randomly choose a prod from eligible_prods && modify codon value
                               accordingly
34                         update the dependency graph to reflect the chosen output variable
35                     else
36                         for prod ∈ rule.prods do
37                             if prod contains an output_var then
38                                 if output_var has prods valid for selection then
39                                     eligible_prods ← prod
40                             else
41                                 eligible_prods ← prod
42                         prod_choice ← codon % rule.prods.size() ;
43                         logic here same as from lines line 32 to line 33
44                 else
45                     if current_rule == OUTPUT_RULE_VAR then
46                         eligible_prods ← valid OUTPUT_RULE_VAR.prods
47                     else
48                         logic is the same as from line 36 to line 41
49                     prod_choice ← invoke perfect wrapping operator;
50                 if current_rule == OUTPUT_RULE_VAR then
51                     update the dependency graph with the chosen prod
52             selected_prod ← rule.prods[prod_choice];
53             for sym ∈ reverse(selected_prod.symbols) do
54                 push prod_symbol onto derivation_stack;
55     {genome.phenotype, genome.effective_size} ← {phenotype,effective_size} genome.phenotype_validity ←
         true;
```

**Algorithm 3:** Cyclic Dependency Checking

```
1  Function UpdateGraphMatrix (M, pos_{x,y})
2      M[pos_x, pos_y] ← 0;
3      while i < n do
4          if M[i, pos_y] = 1 then
5              M ← UpdateGraphMatrix(M, [pos_x, i]);
6          i++;
7      return M;
8  Function FillMatrix (n, dependencies)
9      //M ← Initialise matrix;
10     M[i,j] ← 0, where i = j;
11     M[i,j] ← ✖, where i ≠ j;
12     for pos in dependencies do
13         if M[pos_x, pos_y] = 0 then
14             raise error;
15         else
16             M[pos_x, pos_y] = 1;
17             M = UpdateGraphMatrix(M, [pos_y, pos_x]);
18             while i < n do
19                 if M[i, pos_y] = 0 then
20                     M = UpdateGraphMatrix(M, [i, pos_x]);
21                 i++;
22     return M;
```

To mitigate this effect, a new wrapping operator called *perfect wrapping* is introduced. Perfect wrapping was inspired by PTC2 as described in Section 5.3. Similarly, perfect wrapping requires the grammar rules and productions to be labelled with the minimal number of codons required to map fully. The perfect wrapping operator requires a list of eligible productions as input. If the rule being derived is an output variable rule, its corresponding dependency graph must be checked to determine its eligible productions. First, the list is populated with eligible productions which are non-recursive and require the minimum number of codons to expand fully. If the list is empty, then recursive productions requiring the minimum number of codons to expand are considered. Second, the mod rule is applied to determine if the chosen production is part of the list of eligible productions. If not, one of the productions in the list is uniformly selected at random, and Equation 7.1 is used to generate a new codon to replace the current

codon, which, when used with the mod rule, dictates the same choice. This process continues while codons are reused from the genome until a valid sentence is attained.

Perfect wrapping is similar to two strategies used to repair invalid individuals in PonyGE2 (Fenton et al., 2017). The first strategy requires a pre-specified default rule, which is used deterministically to fully map an incomplete individual. The second strategy is to only select non-recursive rules during the wrapping events though this may not always be the case.

Similarly, in (Akira et al., 2008), during wrapping, rules that increase the number of non-terminal symbols are not selected, thereby reducing the number of invalid individuals.

Perfect wrapping differs from these strategies as non-recursive productions are considered first, with priority given to production(s) and non-terminals requiring a minimal number of codons to expand fully. The same selection criteria are used when considering recursive productions.

$$new\_codon = random\_number\% \left( \frac{max\_codon\_value + 1}{choices} \right) \times choices + chosen\_prod \tag{7.1}$$

### 7.4.9 Selection

The selection operator implemented for MGGE does not rely on the total aggregated fitness of the genomes of an individual. This is because it is possible the worst-performing parent might have completely solved one of the output variables but will rarely or never be selected as a parent. Instead, a *pseudo-parent* is created by performing $n$ number of selection events, where the best-performing genome per output variable is selected from the pool of potential parents. Any selection operator, such as tournament, lexicase, and roulette-wheel selections, can be used to select each genome.

For example, in a standard crossover event, two pseudo-parents are created and used to create two offspring. The selection process is illustrated in Figure 7.3.

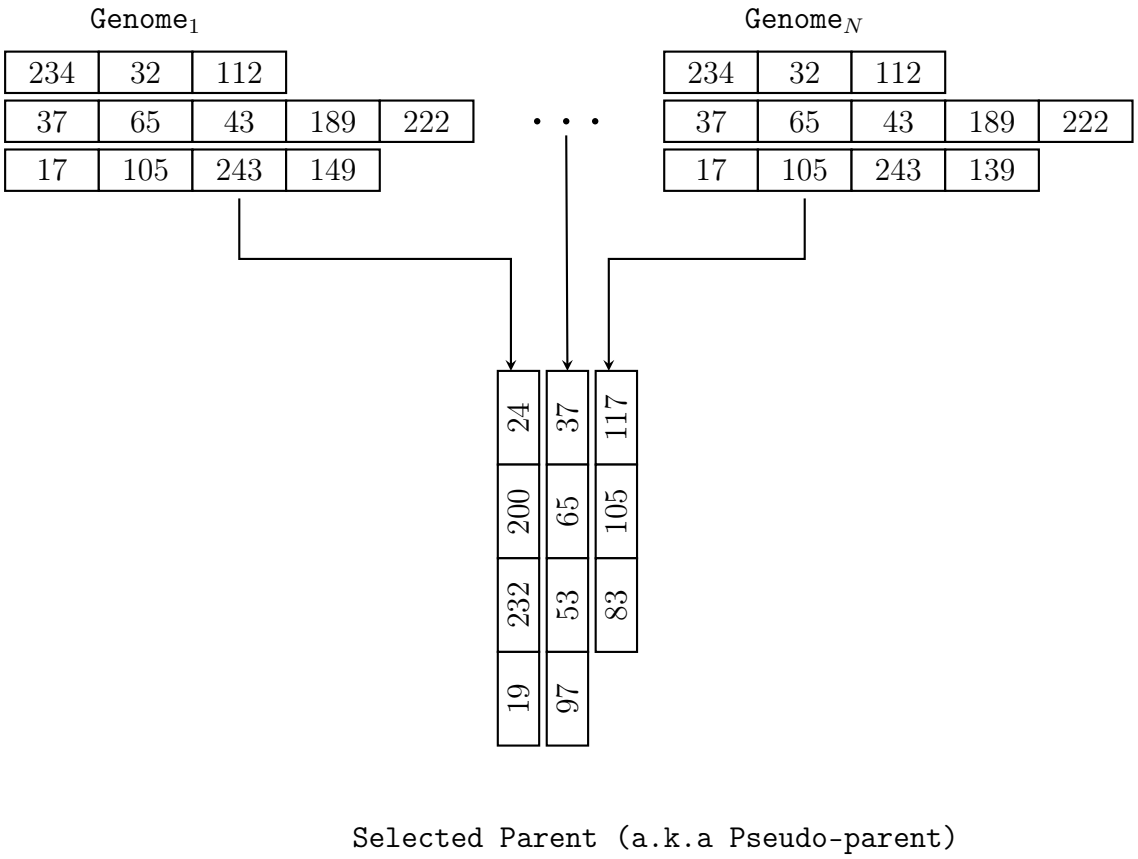


**Figure 7.3:** Multi-Genome Selection. Selection occurs between genomes that occupy the same index in the multi-genome.

### 7.4.10 Well-formed Crossover and Mutation

Each genome in a multi-genome evolves a solution to a single and unique target variable. As a result, crossover operations occur only between genomes that evolve the same output variable. Mutation operations are applied per genome.

These genetic operations can be performed in a number of ways. Three types of events are currently implemented for genetic operation purposes. These include: *random single-event*, *binary mask-event* and *all-event*. In a random single event, only a single genome is randomly chosen for crossover and mutation. For a random single event, a binary mask of bit size equal to the number of genomes is randomly generated. All genomes in multi-genome with bit positions in the binary mask whose bit values are one are chosen for mutation and crossover. In all

cases, all genomes undergo genetic operations subject to the specified probability of these events occurring by the user.

### 7.4.11 Substitution Operator

In solving multi-output problems, there may exist inter-output dependencies as noted in Section 7.2. To facilitate the evolution of these dependencies, we introduced custom grammar format specifications as described in Section 7.4.4. Also, recall each genome in MG is meant to evolve a solution to a single unique output. Evolution is creative and can potentially exploit the inter-output dependency feature by using other sub-optimal output variable expressions as building blocks to solve another output perfectly. However, before extracting the solved output from an individual, if there exists a dependency on solved output variables, it is necessary to rewrite/substitute such output variables with their corresponding expression to obtain the final perfect expression for the solved output.

From Listing 19, assuming output variable `output_1` has completely been solved. Recall the best-evolved solution is a combination of the best-performing genome per output variable. Given that `output_1` is dependent on `output_2` and `output_2` is, in turn, dependent on `output_3`, if we extract `output_1` without recursively substituting output variables with their corresponding expressions to obtain an expression devoid of output variables, `output_1` will no longer be regarded as solved, as `output_2` and `output_3` are very likely to be different in the created best individual. Applying the substitution operator, we will obtain the solved expression devoid of any output variables as shown in Listing 20. Hence, the importance of the cyclic dependency checking algorithm is to ensure the substitution operator does not get stuck in a non-terminating loop as a result of cyclic dependency existing between output variables.

```
output_1 = cos(x) + output_2 + output_2; (50/50)
output_2 = output_3 + sin(y); (5/50)
output_3 = 2^3 + 3; (1/50)
```

**Listing 19:** An individual that completely evolves the solution to *output_1 (50 out of 50 total cases).*

```
output_1 = cos(x) + output_2 + output_2;
output_1 = cos(x) + (output_3 + sin(y)) + (output_3 + sin(y));
output_1 = cos(x) + ((2^3 + 3) + sin(y)) + ((2^3 + 3) + sin(y));
```

**Listing 20:** Final perfect *output_1* statement after applying the substitution operator.

In Listing 20, a typical iteration of the substitution operator to rewrite the solved `output_1` variable before it is extracted is shown. As can be observed from Listing 20, output variable `output_1` used output variable `output_2` twice which were replaced by the same expression at the end of the substitution operation. That is, in cases where an output variable uses another output variable multiple times will result in an increase in the size of the phenotype. In the case of digital circuits, such a scenario will increase the total number of gates required to realise the gate-level design of the circuit. However, this is a post-processing issue and is not addressed in this work.

### 7.4.12 Evolutionary Cycle

A steady-state EA is used to run the benchmark problems in this work. Unlike standard GE, where the best-performing individual is returned as the best-evolved solution, MGGE creates its best individual by combining the best-performing genome for each output variable. However, when an output variable solution has been evolved, substitution operations are performed at the phenotype level described in Section 7.4.11. The function of the substitution operator is to rewrite an expression in terms of variables other than the output variables.

Once an output variable has been solved, its solution is made available to all individuals in the population, and evolution no longer searches for it. However, this functionality can be modified if required for evolution to continue the search but for solutions with additional desirable properties, such as shorter expressions.

## 7.5 Experiments

Two combinational benchmark problems are used in this work: ripple carry adder and hamming code encoder. Two instances of each problem are considered based on the number of inputs and outputs, Hamming Code (7,4) and (15,11) Encoders, and 5-bit + 5-bit and 10-bit + 10-bit Adders.

Experiments are conducted to investigate the performance of standard GE versus MGGE on these benchmark problems. Two grammars are for each experiment. The first grammar variant, which is called Grammar with Output Variable Sharing, allows output variables to use other output variables as building blocks as long as no cyclic dependency between output variables is formed. This is achieved by the definition of Output Variable Rules as described in Section 7.4.4. In other words, Output Variable Rules, together with the cyclic dependency algorithm, are the means by which MGGE evolves inter-output dependencies that may exist. This feature is modelled in the grammar by adding the Output Variable Rule non-terminal to the ⟨expr-i⟩ rules as the last production (highlighted in both grammars). The second version of the grammar, which is referred to as Grammar with No-Output Variable Sharing, follows the same structure as the previous grammar but simply omits the Output Variable Rule. Hence, with this grammar variant, if an inter-output dependency between two output variables exists, evolution will have to evolve the expression/functionality of the independent output variable independently.

We hypothesise that for problems where there exist inter-output dependencies, the Grammar with Output Variable Sharing will perform better than the Grammar with No-Output Variable Sharing and vice versa.

### 7.5.1 Ripple Carry Adder

There are several types of digital adders, such as ripple-carry, carry-save and carry-lookahead adders. In this work, the ripple-carry adder is used as its operation follows elementary addition, which is ideal for explaining the concept of output variable sharing.

Illustrated in Figure 7.4 is the elementary addition of two numbers. The addition operation starts from the last column (or the least significant numbers)

and propagates any carry to the next least significant numbers up until the most significant numbers (first column). As can be observed, the $(n)th$ column addition depends/requires the carry from the $(n-1)th$ column addition. This problem can be modelled as an 8-output problem: four carry and four sum digits. Each sum digit (in other words, output) will require/depend on the carry from the previous addition stage as input. Without the inter-output dependency feature, each column addition to obtaining a sum digit will additionally require evolving the expression to generate the appropriate carry, taking into account all the carry signals propagated starting from the addition of the least significant digits of the addends.

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1\ \ } \\ 8\ 2\ 3 \\ +\quad 9\ 8\ 9 \\ \hline 1\ 8\ 1\ 2 \end{array}$$

**Figure 7.4:** Example of elementary addition.

### 7.5.2 Hamming Code Encoder

As described in Section 4.6.1.1, hamming codes belong to the family of linear block codes and can be grouped into two categories: systematic and non-systematic encodings. Similarly, systematic encoding is adopted, as its structure is easier—data-word followed by the parity bits to obtain the code-word.

Hamming Code (7,4) Encoder encodes a 4-bit data word into a 7-bit code word before transmission. Therefore, the Hamming Code (7,4) Encoder can be modelled as a 3-output problem; each output represents the redundant bit generated and added to the data word to obtain the code word. Each redundant bit is generated by applying bit operations to a unique set of specific bits in the data word. Therefore, there exists no dependency between the outputs of the hamming code encoder. Hence, it is an ideal problem to explain the no-output variable sharing concept as well as the performance of MGGE on such problems.

### 7.5.3 Grammars

The grammars for the adder and hamming code encoder problems are shown in Listings 21 and 22, respectively.

```
          <stmts> ::= {<tr1-sum-i><tr1-cout-i>}_{n=5}
                      {<tr1-sum-i+1><tr1-cout-i+1>}
                       ...
                      {<tr1-sum-i=n><tr1-cout-i=n>}
 <tr1-sum-i>_{i=[1..n]} ::= sum[i] "=" <expr-i>; <nextline>
<tr1-cout-i>_{i=[1..n]} ::= in_cout[i] "=" <expr-i>; <nextline>
    <expr-i>_{i=[1..n]} ::= a[i] | b[i] | c_in
                        | (<expr-i><bitwise-op><expr-i>)
                        | <expr-i> <bitwise-op> <expr-i>
                        | <tv1-input>
    <bitwise-op> ::= "&" | "|" | "^" | "~^"
     <tv1-input> ::= {sum[i = 1] | in_cout[i = 1]}
                        | {sum[i = i + 1] | in_cout[i = i + 1]} |
                          ...
                        | {sum[i = n] | in_cout[i = n]}_{n=5}
-------------------------------------------------------------------
<output-stmt> ::= codeword " = {dataword,parity_bit};"
                  <nextline>
   <nextline> ::= "\n"
```

**Listing 21:** N-bit + N-bit Adder Grammar. Output variable sharing is enabled by making an output variable rule non-terminal a production of ⟨expr-i⟩ rules as highlighted in blue

Both grammars are shown in a compressed form to save space, with certain repetitive rules, production choices and symbols compressed. By repetitive, we mean rules with very similar, but not identical, definitions.

A repetitive rule with a similar definition has been compressed using the syntax $\langle\text{tr1-sum-i}\rangle_{i=[1..n]}$ as shown in Listing 21. The same compression syntax has been applied to repetitive production choices and symbols and enclosed with curly braces where appropriate.

Both grammars have been designed to use bitwise operators (i.e. gate-level design), which makes the circuit problems more challenging to evolve from scratch. Each grammar is divided into two: *upper* and *lower* sections, separated by dashed lines. The upper section of the grammar contains the rules that need to be expanded. The lower section contains rules that define the fixed part of the circuit

```
            <stmts> ::= <tr1-syndbit-i><tr1-syndbit-{i+1}>
                        ··· <tr1-syndbit-i = m>
<tr1-syndbit-i>_{i=[1..m]} ::= parity_bit[i] "=" <expr>";" <nextline>
        <tv1-input> ::= parity_bit[1] | parity_bit[2]
                      | parity_bit[3] | parity_bit[4]
             <expr> ::= <input> | <input><bitwise-op><expr>
                      | <bitwise-neg>(<expr>)
            <input> ::= dataword[ <index> ] | <tv1-input>
            <index> ::= i | i+1 | ··· | k-1 | k = 2^m - m - 1
       <bitwise-op> ::= & | "|" | ^
      <bitwise-neg> ::= ~
-------------------------------------------------------------------------
<output-stmt> ::= codeword " = {dataword,parity_bit};"<nextline>
   <nextline> ::= "\n"
```

**Listing 22:** Hamming Code (N,M) Encoder. Output variable sharing is enabled by making an output variable rule non-terminal a production of ⟨input⟩ rules as highlighted in blue

problem, such as the circuit interface. An expanded form of the grammar for the 5-bit + 5-bit Adder with Output and No-Output sharing are shown in Listings 23 and 24 respectively.

### 7.5.4 Experimental Parameters

The experimental parameters for conducting the experiment are shown in Table 7.1. Preliminary experiments were conducted to tune these parameters. A population size of 1,000 is used for all other experiments except the 10-bit + 10-bit Adder, which is a more challenging circuit to evolve. Down-sampled lexicase selection is used for each selection event. Down-sampled lexicase sub-samples training cases during the selection event, thereby reducing the total number of evaluations (Hernandez et al., 2019).

## 7.6 Results and Discussions

Figures 7.5, **??**, 7.7 and 7.8 show the evolutionary performance for Hamming Code (7,4) Encoder, Hamming Code (15,11) Encoder, 5-bit and 10-bit adders using both grammar versions, respectively. The solid red line, dashed red line, solid

```
      <stmts> ::= <tr1-sum-1> <tr1-cout-1><tr1-sum-2> <tr1-cout-2>
                  <tr1-sum-3><tr1-cout-3><tr1-sum-4> <tr1-cout-4>
                  <tr1-sum-5><tr1-cout-5>
  <tr1-sum-1> ::= sum[0] "=" <expr-1>; <nextline>
 <tr1-cout-1> ::= in_cout[0] "=" <expr-1>; <nextline>
  <tr1-sum-2> ::= sum[1] "=" <expr-2>; <nextline>
 <tr1-cout-2> ::= in_cout[1] "=" <expr-2>; <nextline>
  <tr1-sum-3> ::= sum[2] "=" <expr-3>; <nextline>
 <tr1-cout-3> ::= in_cout[2] "=" <expr-3>; <nextline>
  <tr1-sum-4> ::= sum[3] "=" <expr-4>; <nextline>
 <tr1-cout-4> ::= in_cout[3] "=" <expr-4>; <nextline>
  <tr1-sum-5> ::= sum[4] "=" <expr-5>; <nextline>
 <tr1-cout-5> ::= in_cout[4] "=" <expr-5>; <nextline>
     <expr-1> ::= a[0] | b[0] | c_in | <expr-1><bitwise-op><expr-1>
                | <expr-1><bitwise-op>(<expr-1>) | <tv1-input>
     <expr-2> ::= a[1] | b[1] | c_in | <expr-2><bitwise-op><expr-2>
                | <expr-2><bitwise-op>(<expr-2>) | <tv1-input>
     <expr-3> ::= a[2] | b[2] | c_in | <expr-3><bitwise-op><expr-3>
                | <expr-3><bitwise-op>(<expr-3>) | <tv1-input>
     <expr-4> ::= a[3] | b[3] | c_in | <expr-4><bitwise-op><expr-4>
                | <expr-4><bitwise-op>(<expr-4>) | <tv1-input>
     <expr-5> ::= a[4] | b[4] | c_in | <expr-5><bitwise-op><expr-5>
                | <expr-5><bitwise-op> (<expr-5>) | <tv1-input>
 <bitwise-op> ::= & | "/" | ^ | ~^
 <addend-idx> ::= 0 | 1 | 2 | 3 | 4
  <tv1-input> ::= sum[0] | in_cout[0] | sum[1] | in_cout[1] | sum[2]
               | in_cout[2] | sum[3] | in_cout[3] | sum[4]
               | in_cout[4]
-------------------------------------------------------------------------
<begin-module> ::= "module adder(input logic [4:0] a, b, input
                    logic c_in, output logic [4:0] in_cout, output
                    logic [4:0] sum);" <nextline>
                    always @(*) begin <nextline><stmts><nextline>
                    end <nextline> <endmodule>
  <endmodule> ::= endmodule
   <nextline> ::= "\n"
```

**Listing 23:** 5-bit + 5-bit Adder Grammar with Output Variable Sharing. Output variable sharing is enabled by making an output variable rule non-terminal a production of ⟨expr-*⟩ rules as highlighted in blue.

black line, and dashed black line are the mean best fitness across 30 independent runs for MGGE using the Grammar with Output Variable Sharing, MGGE using grammar with No-Output Variable Sharing, GE using Grammar with Output Variable Sharing, GE using grammar with No-Output Variable Sharing respec-

```
     <stmts> ::= <tr1-sum-1><tr1-cout-1><tr1-sum-2><tr1-cout-2>
                 <tr1-sum-3><tr1-cout-3><tr1-sum-4><tr1-cout-4>
                 <tr1-sum-5> <tr1-cout-5>
 <tr1-sum-1> ::= sum[0] "=" <expr-1>; <nextline>
<tr1-cout-1> ::= in_cout[0] "=" <expr-1>; <nextline>
 <tr1-sum-2> ::= sum[1] "=" <expr-2>; <nextline>
<tr1-cout-2> ::= in_cout[1] "=" <expr-2>; <nextline>
 <tr1-sum-3> ::= sum[2] "=" <expr-3>; <nextline>
<tr1-cout-3> ::= in_cout[2] "=" <expr-3>; <nextline>
 <tr1-sum-4> ::= sum[3] "=" <expr-4>; <nextline>
<tr1-cout-4> ::= in_cout[3] "=" <expr-4>; <nextline>
 <tr1-sum-5> ::= sum[4] "=" <expr-5>; <nextline>
<tr1-cout-5> ::= in_cout[4] "=" <expr-5>; <nextline>
    <expr-1> ::= a[0] | b[0] | c_in | <expr-1><bitwise-op><expr-1>
               | <expr-1><bitwise-op>(<expr-1>)
    <expr-2> ::= a[1] | b[1] | c_in | <expr-2><bitwise-op><expr-2>
               | <expr-2><bitwise-op>(<expr-2>)
    <expr-3> ::= a[2] | b[2] | c_in | <expr-3><bitwise-op><expr-3>
               | <expr-3><bitwise-op>(<expr-3>)
    <expr-4> ::= a[3] | b[3] | c_in | <expr-4><bitwise-op><expr-4>
               | <expr-4><bitwise-op>(<expr-4>)
    <expr-5> ::= a[4] | b[4] | c_in | <expr-5><bitwise-op><expr-5>
               | <expr-5><bitwise-op> (<expr-5>)
<bitwise-op> ::= & | "/" | ^ | ~^
<addend-idx> ::= 0 | 1 | 2 | 3 | 4
 <tv1-input> ::= sum[0] | in_cout[0] | sum[1] | in_cout[1] | sum[2]
               | in_cout[2] | sum[3] | in_cout[3] | sum[4]
               | in_cout[4]
-------------------------------------------------------------------
<begin-module> ::= "module adder(input logic [4:0] a, b, input
                    logic c_in, output logic [4:0] in_cout, output
                    logic [4:0] sum);" <nextline>
                    always @(*) begin <nextline><stmts><nextline>
                    end <nextline> <endmodule>
   <endmodule> ::= endmodule
    <nextline> ::= "\n"
```

**Listing 24:** 5-bit + 5-bit Adder Grammar without Output Variable Sharing

tively. In Table 7.2, we summarise the number of independent runs for each experimental setup.

The benefits of the new genome representation can be observed from Figures 7.5, **??**, 7.7, and 7.8. The experimental setups using MGGE outperform GE when the same grammar variants are used with the exception of the 5-bit + 5-bit adder in Figure 7.7, where both MGGE and GE using Grammar with No-Output

**Table 7.1:** Evolutionary Run Parameters. In bold are the default population sizes used for the actual experiments.

| Parameter | Value |
|---|---|
| Initialisation | SI |
| Selection | Down-sampled Lexicase Selection |
| Crossover Rate | 0.8 |
| Mutation Rate | 0.01 |
| Replacement Rate | 0.05 |
| Number of Runs | 30 |
| Generations | 100 |
| Mutation | All-events Well-formed Mutation |
| Crossover | All-events Well-formed Crossover |
| Wrapping Operator | Perfect Wrapping |
| Population Size | 1000 All other problems<br>2000 10-bit + 10-bit Adder |
| Termination Condition | When mean, minimum and maximum fitness equals the maximum fitness or generation number equal specified generation number |

**Table 7.2:** Success Rate On Benchmark Problems Out of 30 Independent Runs.

| Problem | Standard GE | | MGGE | |
|---|---|---|---|---|
| | *Output Variable Sharing* | *No-Output Variable Sharing* | *Output-Variable Sharing* | *No-Output Variable Sharing* |
| 5-bit + 5-bit Adder | 0 | 0 | 16 | 0 |
| 10-bit + 10-bit Adder | 0 | 0 | 21 | 0 |
| Hamming Code (7,4) Encoder | 18 | 24 | 30 | 30 |
| Hamming Code (15,11) Encoder | 0 | 0 | 0 | 27 |

Variable Sharing converge to similar final mean best fitness. However, MGGE Grammar with No-Output Variable Sharing attains the final mean best fitness after a few generations. Also observed is that both MGGE setups for the Hamming Code (7,4) Encoder in Figure 7.5 attain their final mean best fitness after a few generations. This suggests evolution can perform quite well on relatively easier problems with no inter-output dependencies when the least appropriate grammar is used (Grammar with Output Variable Sharing). Furthermore, MGGE outperforms GE on all problems when used with the most appropriate grammar

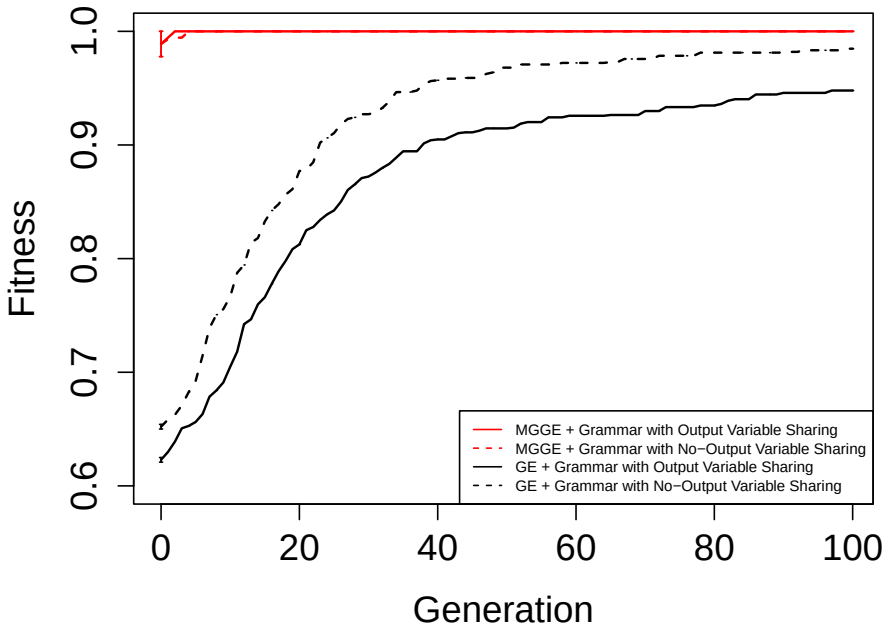


**Figure 7.5:** Mean average best across 30 independent runs for Hamming Code (7,4) using MGGE and standard GE.

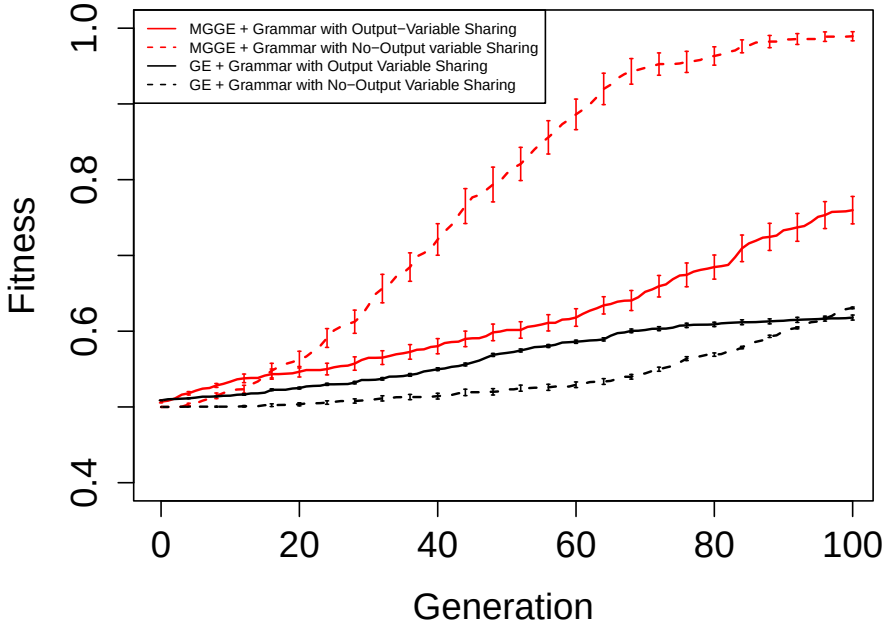


**Figure 7.6:** Mean average best across 30 independent runs for Hamming Code (15,11) using MGGE and standard GE.

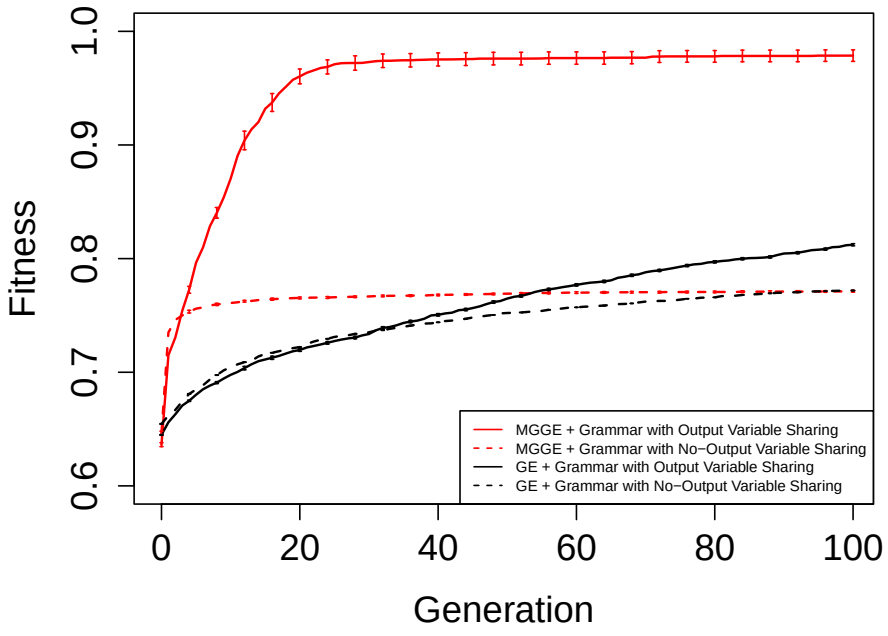


**Figure 7.7:** Mean average best across 30 independent runs for 5-bit + 5-bit Adder using MGGE and standard GE.

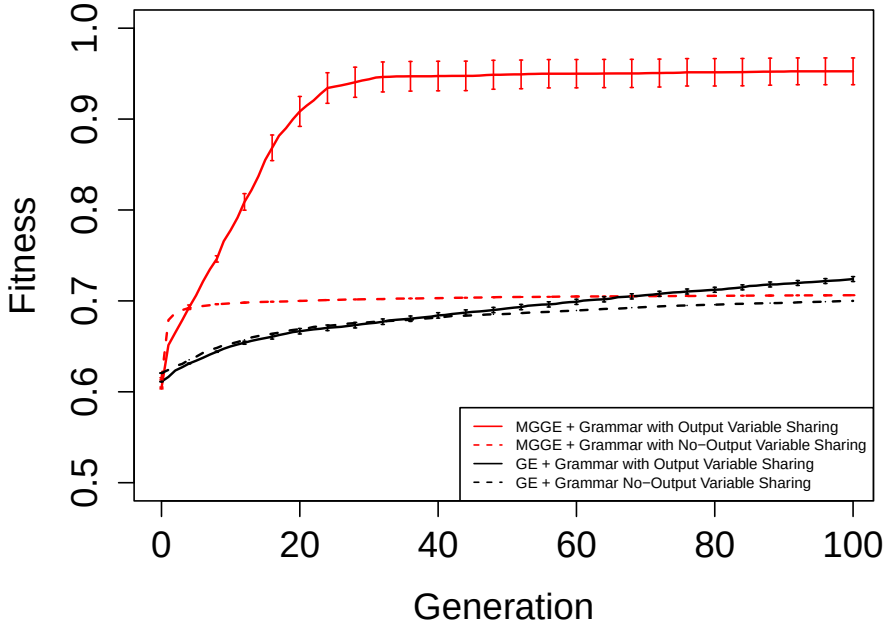


**Figure 7.8:** Mean average best across 30 independent runs for 10-bit + 10-bit Adder using MGGE and standard GE.

variant.

As can be observed, the experimental setup for MGGE using the Grammar with Output-Variable Sharing for the 5-bit + 5-bit and 10-bit + 10-bit adders recorded a success rate of 16/30 and 21/30, respectively, while all other setups recorded zero successful runs. This is because the ripple-carry adder, which operates like elementary addition, propagates the carry bits from the previous bit addition stages to the last stage of bit addition. Hence, designing the grammar to allow output variable sharing while ensuring no cyclic dependency between output variables is formed ensures correct carry outputs are available for the summation stages to use. As a result, the Grammars without Output Variable Sharing will require evolution to evolve the carry-out bit from the previous summation stage for the current summation stage, rendering the problem more challenging to deal with. Furthermore, the 5-bit + 5-bit adder recorded a lower success rate than the more challenging 10-bit + 10-bit adder; this was because the setup for the 10-bit + 10-bit adder used a population size of 2,000, which is twice the population size of 1,000 used for the 5-bit + 5-bit adder.

On the Hamming Code (7,4) and Hamming Code (15,11) Encoder problems, MGGE with the Grammar with No-Output Variable Sharing obtained a success rate of 30/30 and 27/30, respectively. Using the Grammar with Output Variable Sharing, MGGE obtained zero successful runs. This is due to the fact that for the hamming code encoders, there exist no dependencies between the output variables. Therefore, using the grammar with Output Variable Sharing for a problem with non-existing inter-output dependencies impedes evolutionary performance.

Finally, for both problems, the error bars for the MGGE setups are larger for the versions with higher input sizes (Hamming code (15,11) encoder and 10-bit + 10-bit adder). This indicates greater variability in individual fitness values compared to the mean fitness value for these problems. These problems involve more output variables, which makes evolving a solution more challenging, hence the higher variability in individual fitness values.

### 7.6.1 Analysis of Representative Solutions

```
parity_bit[1] = dataword[7]^(dataword[1]^(dataword[5]^dataword[9]~^(dataword[9]
                ~^dataword[4]~^dataword[11]^dataword[9]~^dataword[2])));
parity_bit[2] = dataword[6]~^(dataword[7]^dataword[3]^dataword[4]~^(dataword[10]
                 ~^dataword[1]~^dataword[11]));
parity_bit[3] = dataword[4]^(dataword[10]~^dataword[9]~^dataword[3]^dataword[2]~^(dataword[11]
                 ~^dataword[8]));
parity_bit[4] = dataword[9]~^(dataword[8]~^dataword[10]~^(dataword[11]~^(dataword[7]
                 ~^(dataword[5]~^dataword[6]))));
```

**Listing 25:** Representative Solution Obtained for Hamming Code (15, 11) Encoder using MGGE and Grammar (without Output variable sharing).

A representative solution for each problem type (with and without output dependencies) is shown in Listings Listing 25 and Listing 26. Listing 25, shows a representative solution of Hamming Code (15, 11) Encoder, a problem without output inter-dependencies. It can be observed from the solution that the expressions that evolve each parity bit solely rely on the data-word bits. As presented in Table 7.2, no representation solutions were attained for experimental setups using grammars with Output Variable Sharing. The final or best solutions for these setups had at least the expression for generating one of the bits relying on an output variable as an input. Hence, evolving a solution to a problem where output dependencies do not exist increases the difficulty of the problem as, in addition to the problem being solved, there is an additional constraint that has to be satisfied, and that is, evolved solutions must be devoid of output variables.

In Listing 26, the Adder problem, whose operations follow the elementary column addition method, involves propagating carry bits from the least to the most significant bits column additions. As can be observed, the sum bit expressions use the carry bit from the addition of the previous or less significant bit to obtain the sum and carry bits. However, the final solutions for the Adder experimental setups that use grammars without output variable sharing were unable to obtain a representative solution. This is because each subsequent column, other than the least significant column, has to be able to evolve an expression which generates the correct carry bit from the previous column addition to be able to generate its corresponding sum bit.

```
    sum[0] = b[0]^(c_in)^a[0]|(b[0]^(b[0]));
in_cout[0] = b[0]^(c_in~^a[0]&a[0])&(c_in^b[0]);
    sum[1] = in_cout[0]^b[1]^(a[1])&(a[1]|(a[1]));
in_cout[1] = a[1]&(in_cout[0])|(b[1]&(in_cout[0]))|(b[1]&(a[1]));
    sum[2] = a[2]&b[2]~^(a[2]~^(in_cout[1])~^(a[2]~^(b[2]|(a[2]))^(b[2]^(b[2]))));
in_cout[2] = a[2]&(in_cout[1])|(b[2]&(a[2]|in_cout[1]&(b[2]|(a[2]
             &c_in|(b[2]^(a[2]|in_cout[1])&(b[2])))));
    sum[3] = a[3]^b[3]~^(a[3])&b[3]~^(in_cout[2])^(b[3]&(a[3]))|(b[3]^b[3]);
in_cout[3] = b[3]&a[3]|(a[3])&(in_cout[2])|(b[3]&in_cout[2])&(b[3]^a[3]|c_in^(c_in));
    sum[4] = a[4]~^b[4]~^(in_cout[3]);
in_cout[4] = a[4]&(a[4]&(in_cout[3])&(a[4]))|(b[4]&(in_cout[3]^b[4]&a[4]));
    sum[5] = b[5]~^a[5]~^(in_cout[4]);
in_cout[5] = a[5]&(in_cout[4])|b[5]&a[5]|(b[5]&(in_cout[4]));
    sum[6] = a[6]^b[6]^(in_cout[5]);
in_cout[6] = b[6]&(a[6]|in_cout[5])|(a[6]~^(b[6]))~^b[6]~^(b[6]|b[6]~^(a[6])&(b[6]&(a[6]
             |in_cout[5])|(a[6]~^(b[6]))~^b[6]~^(b[6]|b[6]~^(a[6])&(b[6]|b[6]
             &(a[6]|in_cout[5])|(a[6]~^(b[6]))^(b[6])^(b[6]^in_cout[5])))));
    sum[7] = a[7]~^((b[7])~^(in_cout[6]));
in_cout[7] = b[7]~^(b[7]~^a[7]|b[7]~^a[7]|b[7]~^(in_cout[6])^(b[7]
             &(c_in&a[7]~^(c_in))~^c_in&c_in)&(b[7]));
    sum[8] = b[8]~^(in_cout[7])~^(a[8]);
    sum[9] = b[9]~^(b[9]~^(a[9]~^(b[9]~^a[9])~^(a[9]~^(in_cout[9]))));
in_cout[9] = b[9]^(a[9]^b[9])&(b[9]^(a[9]^b[9])&(in_cout[9]^c_in^(c_in)));
```

**Listing 26:** Representative solution obtained for 10-bit + 10-bit Adder using MGGE and grammar (with output variable sharing).

## 7.7 Conclusions

In this chapter, MGGE was presented, and it was demonstrated that it is better suited for solving multi-output (or multi-target) problems compared to standard GE. Genetic operators, mappers and initialisation routines have been adapted to work with the new genome representation. Custom grammar format specifications, together with a cyclic dependency-checking algorithm, have been developed to facilitate the evolution of inter-output dependencies. A new wrapping operator called perfect wrapping has been developed to ensure every single MGGE mapping event creates a valid executable object.

# 8

# Conclusions and Future Directions

## 8.1 Introduction

The central hypothesis of this thesis is that the use of GE to design digital circuits at a higher level of abstraction other than gate-level using HDLs, specifically SystemVerilog in this work, facilitates the evolution of fully functional and complex digital circuits. This research designed and developed an evolutionary framework for the functional evolution of circuits called ADDC. Several empirical experiments were conducted to: test the robustness of ADDC, investigate the impact of grammar design and choice of GE initialisation routine on evolutionary performance and the exploitation of expressive programming constructs in SystemVerilog to design complex circuits. Furthermore, it was observed that GE's performance on multi-output problems could be improved by using a more suitable representation structure to encode such problems, which led to the design and implementation of MGGE, a variant of GE well-suited for solving multi-output digital circuit benchmark problems. In summary, this thesis is the first to:

- Show that the use of GE and SystemVerilog (applicable to other HDLs) can evolve digital circuits at a higher level of abstraction other than gate level;

- Use GE and SystemVerilog to evolve fully functional and parameterised Adder, Multiplier, Selective Parity and Up-Down Counter circuits at a more abstract level other than gate level—RTL. Parameterised modules have the additional benefit of not requiring a re-run of evolutionary experiments if multiple instances with different input sizes are required. The Adder (6.4×), Multiplier (10.7×) and Selective Parity (6.7×) circuits are substantially larger than the current state of the art for evolutionary approaches;
- Design and implement a novel GE variant called MGGE better suited for evolving multi-output digital circuits but also applicable to multi-output problems other than digital circuits.

A summary of the results, the research outcomes and future directions for future research are briefly described in this chapter.

## 8.2 Research Outcomes

An overview of the outcomes of the research objectives from the experimental chapters, which are Chapters 4, 5, 6, and 7, are presented below.

### 8.2.1 ADDC Tool for Functional Evolution of Digital Circuits

ADDC, an evolutionary-based tool powered by GE for the evolution of a fully functional digital circuit using SystemVerilog and Icarus Verilog, a Verilog/SystemVerilog simulator, was designed and implemented. ADDC was tested using some combinational digital circuit benchmark problems. The results presented in Section 4.7 demonstrate that GE was capable of evolving digital circuits using SystemVerilog.

### 8.2.2 Improving Evolutionary Performance

Grammar design and initialisation routines are key components of GE that define the size and facilitate an effective search of the search space, respectively.

In order to investigate the impact of these two components, a series of experiments was conducted using different grammar versions (with and without domain knowledge) and two of the best GE initialisation routines (Sensible Initialisation and PTC2). The results presented in Section 5.4 revealed grammars designed with simple domain knowledge performed best and PTC2 initialisation routine overall, offered the best performance on all experiments using grammars with and without simple domain knowledge incorporated.

### 8.2.3 Evolution of Complex Digital Circuits

The evolution of complex circuits remains a challenge for the EHW field in spite of much effort. There are two major issues as discussed in Section 3.5: the amount of testing required and the low evolvability of representation structures to handle complex circuitry, at least partially due to the destructive effects of genetic operators. GE was used to evolve the most complex adder, multiplier, selective parity and up-down circuits successfully from scratch and without resorting to the use of any standard decomposition methods due to our ability to use programming constructs and operators available in SystemVerilog. Additionally, all evolved circuits are parameterised, meaning they have the additional benefit of not requiring a re-run of evolutionary experiments if multiple instances with different input sizes are required.

Furthermore, a well-known technique for reducing testing from Digital Circuit Design known as corner case testing, which is relied upon by skilled circuit designers to avoid time-consuming exhaustive testing, was employed to reduce the amount of testing. Also demonstrated was a simple way to identify and use corner cases for evolutionary testing and show that it enables the generation of massively complex circuits.

The results for these experiments were presented in Section 6.4

### 8.2.4 Evolving Multi-Output Circuit Benchmarks Using a Novel Multi-Genome GE

Digital circuits are characterised by single or multiple outputs. With the use of standard GE, the same linear genome representation is used to encode the

solutions. However, GE could perform even better on multi-output problems if a more suitable representation and adapted operators are used instead. To investigate this hypothesis, a multi-genome GE was implemented and tested on multi-output problems and results compared to that of standard GE. The results presented in Section 7.6 showed that the novel GE variant outperforms standard GE substantially and is more suitable for multi-output digital circuit problems.

## 8.3 Limitations

Despite results suggesting the use of GE and HDL is ideal for digital circuit design tasks, it is not without some limitations. The use of an HDL requires the user to have technical knowledge about the HDL of choice—how to design grammars free of syntax errors and modelling errors. Syntax errors are easier to find and fix as most simulators will report such errors at the functional simulation phase. Modelling errors, on the other hand, are more challenging to fix, as they are only noticeable during the synthesis phase (conversion of RTL or high-level designs to gate-level representation) of the circuit design when designed grammars do not adhere to the guidelines of the HDL of choice. For example, if an evolved circuit uses conditional statements such `case` statements and `if` conditions which are not mutually exclusive, these circuits may not be directly synthesisable without post-processing.

There are a number of possible ways to resolve this issue. One is through the definition of new rules or via the modification of existing rules. The second option is the use of Attribute Grammars to impose the necessary constraints. Third, modelling errors such as redundant logic can be gotten rid of by a number of approaches: manually by hardware designer, design of SystemVerilog/Verilog redundant logic removal algorithm, use `unique case` statements (for switch case constructs), etc.

## 8.4 Future Work

### 8.4.1 The Use of Attribute Grammars to Impose Circuit Design Constraints

Digital circuit design specifications may be accompanied by constraints that must be satisfied in cases where such constraints cannot be easily satisfied by the definition of grammar rules and may require the use of attribute grammars. Attribute grammars provide an easier approach to incorporating constraints or semantic information into GE grammars. As stated earlier in Section 8.3, the use of attribute grammars will help ensure only syntactically and semantically valid individuals are evaluated, thereby reducing the total time spent on evaluating the population circuits.

### 8.4.2 Scaling Beyond Higher Abstraction Level of Digital Circuit Evolution

Despite the availability of high-level programming constructs in HDLs, which helps to evolve complex circuits, we acknowledge certain circuit problems require decomposition. Decomposition is a standard practice in industry. HDLs support hierarchical modelling for the design of large and complex circuits. In hierarchical modelling, design modules in lower layers are included in design modules in higher layers. Communication between circuits is established through the input and output ports of the circuits. A very large complex circuit will be slow to test. Therefore, the conventional generate and test approach of standard evolutionary systems may not suffice. A more efficient approach that will exploit the power of modern processors will be of high priority when we begin to venture into industrial-scale circuit designs.

### 8.4.3 Improvement of Multi-Genome GE

Despite the success of MGGE on the benchmark circuit problems used, there are several open questions and further experiments that need to be conducted. For example, the best mutation and crossover operations for the genetic operators

adapted for Multi-Genome GE need to be investigated. We need to develop more intelligent algorithms to help MGGE evolve very complex output dependencies that may exist in other multi-output benchmark problems.

To some extent, the use of multiple genomes and output rules in MGGE can be considered output decomposition. This is because each output variable has a corresponding unique genome which encodes its solution.

Recall that MGGE grammar design requires the user to annotate rules which are to be treated as an output rule by both the parser and the mapper. This feature facilitates the decomposition of the problem. However, well-designed grammars improve evolutionary performance, as has been demonstrated in Chapter 5. As a result, if a user misannotates a rule, this may potentially increase the difficulty in searching regions of the search space where solutions may be perceived to be.

Moving forward, a more dynamic and automatic approach will be required to evolve rules that need to be labelled as output rules and the potential dependencies between them. For example, in (O'Neill and Ryan, 2004), a universal GE is used to co-evolve a solution grammar from a universal grammar and the solution. Such a methodology can be integrated into MGGE to facilitate automatic problem decomposition by co-evolving a solution grammar, which essentially entails labelling rules that should be treated as output rules and their interdependencies.

Furthermore, the current output variable dependency feature (also known as output variable sharing) is very strict, as output variables are only useful as dependencies to other output variables when they completely solve their respective objective. In other words, an output variable is not used as input by another output variable if it has not attained the maximum fitness value. This may not be so beneficial for symbolic regression problems. Moving forward, we need to improve this feature to allow output variables which, when used by other output variables, contribute to the increase of their fitness value.

Finally, MGGE has the potential to be applied to several application domains other than the digital circuit domain. First, MGGE can be used to evolve the single dependent variable and several independent variables before performing multiple regression analysis to combine these variables to predict the target variable. Second, MGGE can be applied to multi-output classification problems. Last

but not least, MGGE can be applied to single-output problems that can be decomposed. Third, MGGE could be adapted to simultaneously solve the problem at hand and tune the grammar by evolving the inter-dependency relationships which may exist between the output variables. In general, MGGE should be applicable to multi-output problems. However, certain application domains may require a number of features of MGGE to be customised, such as the substitution operator and mapping mechanism.

# Appendix A

```
module hc_decoder(input [1:7] i_codeword, output logic [1:7] o_codeword);
integer syndrome_value;
function [2:0] syndrome(input[1:7]cw);
begin
    syndrome[0]=cw[3]^cw[5]^cw[1]^cw[7];
    syndrome[1]=cw[3]&cw[3]^cw[7]^cw[2]^cw[6];
    syndrome[2]=cw[4]^cw[5]^cw[2]^cw[7]^cw[2]^cw[6];
end endfunction
always @(i_codeword)begin
    o_codeword=i_codeword;
    syndrome_value=syndrome(i_codeword);
    if(syndrome_value==3'b0) begin end
    else begin
        o_codeword[syndrome_value]=~i_codeword[syndrome_value]; end
    end
endmodule
```

**Listing 27:** Representative Solution obtained for Hamming Code (7,4) Grammar A.

Let's assume the data to be transmitted is `0101`, and the transmitted data or codeword is supposed to be `0100101`. The syndrome function in Listing 27 will generate the syndrome `100`. Within the `always` block, the `if` condition checks if the syndrome is 0 or 000; if not, there was an error during transmission and, therefore, flips the bit at position or index 4 (100). In effect, the third syndrome bit (syndrome[2]) gets flipped to a 0, resulting in the correct codeword being corrected. Hence, whenever syndrome[2] produces a 1 instead of a 0, it gets corrected, and the codeword is successfully decoded.

```
module hc_decoder(input [1:7] i_codeword, output logic [1:7] o_codeword);
integer syndrome_value;
function[2:0]syndrome(input[1:7]cw);
begin
    syndrome[0]=cw[1]^cw[7]^cw[3]^cw[5];
    syndrome[1]=cw[6]^cw[2]^cw[7]^cw[3];
    syndrome[2]=cw[6]^cw[7]^cw[5]^cw[4];
end endfunction
always @(i_codeword)begin
    o_codeword=i_codeword;
    syndrome_value=syndrome(i_codeword);
    if(syndrome_value==3'b0)begin end
    else begin
        o_codeword[syndrome_value]= ~i_codeword[syndrome_value]; end
end
endmodule
```

**Listing 28:** Representative Solution obtained for Hamming Code (7,4) Grammar B.

```
module mux(output logic out,
    input logic a0,a1, a2, d0,
    d1, d2, d3, d4, d5, d6, d7);

always@(*)
out=(a0?(a1?(a2?d0:(a2?d0:d1)):
    (a2?d2:d3)):(a0?(((a2?((d1&
    d6)|d2):(a2?((d1&d6)|d2):d3))
    |d6)&d6):(a1?(a0?d1:(a2?d4:
    d5)):(a0?!((a2?(d0&d0):
    d0)):(a2?d6:(a1?(a2?!(d6):
    d5):d7))))));

endmodule
```

**Listing 29:** Representation Solution obtained for 11-*bit* Multiplexer Grammar A.

```
module mux(output logic out,
    input logic a0, a1, a2, d0,
    d1, d2, d3, d4, d5, d6, d7);

always@(*)
out=(a2?(a1?(a0?d0:(a2?d4:(a2?
    d4:d6))):(a0?d2:d6)):(a2?(a1?
    d2:(a1?(a1?d4:d7):d2)):(a1?
    (a2?(a0?d6:(a1?(a1?d7:d4):
    d7)):(a0?d1:(a2?d4:d5))):
    (a2?(a0?(a2?d3:d2):d6):
    (a1?(a0?d1:d5):(a1?d5:(a2?
    d7:(a2?(a0?d3:d7):(a0?
    d3:d7)))))))));

endmodule
```

**Listing 30:** Representation Solution obtained for 11-*bit* Multiplexer Grammar B.

```
module ssd(output logic[6:0]
segment, input logic[3:0] bcd);
always@(bcd)
begin
case(bcd)
    4'b0100:segment=7'b0110011;
    4'b0001:segment=7'b0110000;
    4'b1001:segment=7'b1111011;
    4'b0000:segment=7'b1111110;
    4'b1001:segment=7'b1011111;
    4'b0010:segment=7'b1101101;
    4'b0011:segment=7'b1111001;
    4'b0001:segment=7'b1111001;
    4'b0111:segment=7'b1110000;
    4'b0011:segment=7'b1011011;
    4'b0111:segment=7'b1111011;
    4'b0101:segment=7'b1011011;
    4'b1001:segment=7'b1111011;
    4'b0011:segment=7'b1011011;
    4'b0001:segment=7'b1011011;
    4'b1000:segment=7'b1111111;
    4'b0101:segment=7'b1111011;
    4'b1001:segment=7'b1111011;
    4'b0110:segment=7'b1011111;
    default:segment=7'b0000000;
endcase
end
endmodule
```

**Listing 31:** Representative Solution for SSD Problem.

```
module udcounter(input logic clk, up_ndown,
    en_load, enable_ud, reset, input
    logic [3:0] load, output logic [3:0] q);
always @(posedge clk)
casex ({reset, en_load, enable_ud, up_ndown})
    4'b11??:q<=load;
    4'b0???:q<=0-load+load;
    4'b0???:q<=0-load+0+load-0;
    4'b??10:q<=1-1+q-1;
    4'b1?1?:q<=load+1+q-0-load;
endcase
endmodule
```

**Listing 32:** Representative Solution Obtained for *Mod*-16 Up-Down Counter Problem.

```
module jk_ff (input j, input k, input clk,
    output logic q);
always@(posedge clk)
case({j,k})
    2'b01:q<=0;
    2'b10:q<=1;
    2'b00:q<=q;
    2'b11:q<=~q;
    default:q<=q;
endcase
endmodule
```

**Listing 33:** Representation Solution Obtained for JK Flip-flop Problem

# References


Akira, H., Tomohisa, Y., Takumi, I. and Tetsuyuki, T. (2008), Multi-chromosomal Grammatical Evolution, *in* 'Fourth International Workshop on Computation Intelligence & Applications'. 92, 105

Ali, B., Almaini, A. E. A. and Kalganova, T. (2004), 'Evolutionary algorithms and theirs use in the design of sequential logic circuits', *Genetic Programming and Evolvable Machines* **5**(1), 11–29.
**URL:** *https://doi.org/10.1023/B:GENP.0000017009.11392.e2* 2

Baine, N. (2008), A simple multi-chromosome genetic algorithm optimization of a Proportional-plus-Derivative Fuzzy Logic Controller, *in* 'NAFIPS 2008 - 2008 Annual Meeting of the North American Fuzzy Information Processing Society', pp. 1–5. 94

Borchani, H., Varando, G., Bielza, C. and Larrañaga, P. (2015), 'A survey on multi-output regression', *WIREs Data Mining and Knowledge Discovery* **5**(5), 216–233.
**URL:** *https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1157* 91

Castle, T. and Johnson, C. G. (2010), Positional effect of crossover and mutation in grammatical evolution, *in* A. I. Esparcia-Alcázar, A. Ekárt, S. Silva, S. Dignum and A. Ş. Uyar, eds, 'Genetic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 26–37. 90

Coffman, K. (1999), *Real World FPGA Design with Verilog*, chapter Chapter 1: Verilog Designs in the realword, pp. 137–206. 32

Coimbra, V. and Lamar, M. V. (2016), Design and Optimization of Digital Circuits by Artificial Evolution Using Hybrid Multi Chromosome Cartesian Genetic Programming, *in* V. Bonato, C. Bouganis and M. Gorgon, eds, 'Applied Reconfigurable

Computing', Lecture Notes in Computer Science, Springer International Publishing, pp. 195–206. 94

Cullen, J. (2008), Evolving Digital Circuits in an Industry Standard Hardware Description Language, *in* X. Li, M. Kirley, M. Zhang, D. Green, V. Ciesielski, H. Abbass, Z. Michalewicz, T. Hendtlass, K. Deb, K. C. Tan, J. Branke and Y. Shi, eds, 'Simulated Evolution and Learning', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 514–523. 16, 28, 48

da Silva, J. E. H., de Souza, L. A. M. and Bernardino, H. S. (2019), Cartesian genetic programming with guided and single active mutations for designing combinational logic circuits, *in* G. Nicosia, P. Pardalos, R. Umeton, G. Giuffrida and V. Sciacca, eds, 'Machine Learning, Optimization, and Data Science', Springer International Publishing, Cham, pp. 396–408. 24, 25

de Garis, H. (1991), Genetic programming artificial nervous systems artificial embryos and embryological electronics, *in* H.-P. Schwefel and R. Männer, eds, 'Parallel Problem Solving from Nature', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 117–123. 19

de Garis, H. (1993), Evolvable hardware genetic programming of a Darwin machine, *in* R. F. Albrecht, C. R. Reeves and N. C. Steele, eds, 'Artificial Neural Nets and Genetic Algorithms', Springer Vienna, Vienna, pp. 441–449. 20

Eiben, A. E. and Smith, J. E. (2015), *What Is an Evolutionary Algorithm?*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 25–48.
**URL:** *https://doi.org/10.1007/978-3-662-44874-8_3* 10

Fenton, M., McDermott, J., Fagan, D., Forstenlechner, S., Hemberg, E. and O'Neill, M. (2017), Ponyge2: grammatical evolution in python, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference Companion', GECCO '17, Association for Computing Machinery, New York, NY, USA, p. 1194–1201.
**URL:** *https://doi-org.glucksman.idm.oclc.org/10.1145/3067695.3082469* 105

Fredivianus, N., Prothmann, H. and Schmeck, H. (2010), XCS revisited: A novel discovery component for the extended classifier system, *in* K. Deb, A. Bhattacharya, N. Chakraborti, P. Chakroborty, S. Das, J. Dutta, S. K. Gupta, A. Jain, V. Aggarwal, J. Branke, S. J. Louis and K. C. Tan, eds, 'Simulated Evolution and Learning', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 289–298. 51

Fritzsch, C., Hoffmann, J. and Bogdan, M. (2022), Evolving hardware by direct bitstream manipulation of a modern FPGA, *in* '2022 IEEE Congress on Evolutionary Computation (CEC)', pp. 1–8. 23

Harper, R. (2010), 'GE, explosive grammars and the lasting legacy of bad initialisation', *IEEE Congress on Evolutionary Computation* pp. 1–8. 50

Hartenstein, R. W., Herz, M. and Gilbert, F. (1998), Designing for xilinx xc6200 FPGAs, *in* R. W. Hartenstein and A. Keevallik, eds, 'Field-Programmable Logic and Applications From FPGAs to Computing Paradigm', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 29–38. 22

Helmuth, T., Pantridge, E. and Spector, L. (2019), Lexicase selection of specialists, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference', GECCO '19, Association for Computing Machinery, New York, NY, USA, p. 1030–1038.
**URL:** *https://doi.org/10.1145/3321707.3321875* 65

Helmuth, T., Pantridge, E. and Spector, L. (2020), 'On the importance of specialists for lexicase selection', *Genetic Programming and Evolvable Machines* **21**(3), 349–373.
**URL:** *https://doi.org/10.1007/s10710-020-09377-2* 65

Henson, B., Walker, J. A., Trefzer, M. A. and Tyrrell, A. M. (2018), *Designing Digital Systems Using Cartesian Genetic Programming and VHDL*, Springer International Publishing, Cham, pp. 57–86.
**URL:** *https://doi.org/10.1007/978-3-319-67997-6_3* 49, 50, 69, 85, 86

Hernandez, J. G., Lalejini, A., Dolson, E. and Ofria, C. (2019), Random subsampling improves performance in lexicase selection, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference Companion', GECCO '19, Association for Computing Machinery, New York, NY, USA, p. 2028–2031.
**URL:** *https://doi.org/10.1145/3319619.3326900* 112

Higuchi, T., Iwata, M., Kajitani, I., Yamada, H., Manderick, B., Hirao, Y., Murakawa, M., Yoshizawa, S. and Furuya, T. (1996), Evolvable hardware with genetic learning, *in* '1996 IEEE International Symposium on Circuits and Systems (ISCAS)', Vol. 4, pp. 29–32 vol.4. 2

Higuchi, T., Murakawa, M., Iwata, M., Kajitani, I., Weixin Liu and Salami, M. (1997), Evolvable hardware at function level, *in* 'Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)', pp. 187–192. 2

## REFERENCES


Higuchi, T., Niwa, T., Tanaka, T., Iba, H., de Garis, H. and Furuya, T. (1993), Evolving hardware with genetic learning: A first step towards building a darwin machine, *in* 'Proceedings of the Second International Conference on From Animals to Animats 2: Simulation of Adaptive Behavior: Simulation of Adaptive Behavior', MIT Press, Cambridge, MA, USA, p. 417–424. 20

Hodan, D., Mrazek, V. and Vasicek, Z. (2020), Semantically-oriented mutation operator in cartesian genetic programming for evolutionary circuit design, *in* 'Proceedings of the 2020 Genetic and Evolutionary Computation Conference', GECCO '20, Association for Computing Machinery, New York, NY, USA, p. 940–948. 25, 68, 69, 86

Hrbacek, R., Mrazek, V. and Vasicek, Z. (2016), Automatic design of approximate circuits by means of multi-objective evolutionary algorithms, *in* '2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)', pp. 1–6. 22

Hrbacek, R. and Sekanina, L. (2014), Towards Highly Optimized Cartesian Genetic Programming: From Sequential via SIMD and Thread to Massive Parallel Implementation, *in* 'Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation', GECCO '14, Association for Computing Machinery, New York, NY, USA, p. 1015–1022. 25, 27, 69, 86

Jundt, L. and Helmuth, T. (2019), Comparing and combining lexicase selection and novelty search, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference', GECCO '19, Association for Computing Machinery, New York, NY, USA, p. 1047–1055.
**URL:** *https://doi.org/10.1145/3321707.3321787* 65

Kai-hui, C., Markov, I. L. and Bertacco, V. (2009), *Functional Design Errors in Digital Circuits*, 1st edn, Springer Netherlands. 1

Kalganova, T. (2000*a*), Bidirectional incremental evolution in extrinsic evolvable hardware, *in* 'Proceedings. The Second NASA/DoD Workshop on Evolvable Hardware', pp. 65–74. 26

Kalganova, T. (2000*b*), An extrinsic function-level evolvable hardware approach, *in* R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin and T. C. Fogarty, eds, 'Genetic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 60–75. 24

Karpuzcu, U. R. (2005), Automatic Verilog Code Generation Through Grammatical Evolution, *in* 'Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation', GECCO '05, ACM, New York, NY, USA, pp. 394–397. 16, 27, 64

Keymeulen, D., Zebulum, R., Rajeshuni, R., Stoica, A., Katkoori, S., Graves, S., Novak, F. and Antill, C. (2006), Self-adaptive system based on field programmable gate array for extreme temperature electronics, *in* 'First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)', pp. 296–300. 2, 21

Koza, J. R. (1992), *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT press. 14, 51

LaMeres, B. J. (2019), *Introduction to Logic Circuits & Logic Design with Verilog*, Springer International Publishing, chapter Verilog (Part 1), p. 157. 32

Leveson, N. and Turner, C. (1993), 'An investigation of the therac-25 accidents', *Computer* **26**(7), 18–41. 1

Li, Z., Luo, W. and Wang, X. (2008), A stepwise dimension reduction approach to evolutionary design of relative large combinational logic circuits, *in* G. S. Hornby, L. Sekanina and P. C. Haddow, eds, 'Evolvable Systems: From Biology to Hardware', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 47–58. 26, 27, 68, 69, 86

Lohn, J., Haith, G., Colombano, S. and Stassinopoulos, D. (2000), Towards evolving electronic circuits for autonomous space applications, *in* '2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484)', Vol. 5, pp. 473–486 vol.5. 2, 21

Luke, S. (2000), 'Two fast tree-creation algorithms for genetic programming', *Trans. Evol. Comp* **4**(3), 274–283.
**URL:** *https://doi.org/10.1109/4235.873237* 50

Manfrini, F. A. L., Bernardino, H. S. and Barbosa, H. J. C. (2016), A novel efficient mutation for evolutionary design of combinational logic circuits, *in* J. Handl, E. Hart, P. R. Lewis, M. López-Ibáñez, G. Ochoa and B. Paechter, eds, 'Parallel Problem Solving from Nature – PPSN XIV', Springer International Publishing, Cham, pp. 665–674. 24

Medvet, E. (2017), Hierarchical grammatical evolution, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference Companion', GECCO '17, Association for Computing Machinery, New York, NY, USA, p. 249–250.
**URL:** *https://doi.org/10.1145/3067695.3075972* 92

## REFERENCES


Miller, F. P., Vandome, A. F. and McBrewster, J. (2009), *Hamming Code: Parity Bit, Two- out- of- Five Code, Hamming(7,4), Reed-Muller Code, Reed-Solomon Error Correction, Turbo Code, Low- Density Parity-Check Code, Telecommunication, Linear Code*, Alpha Press. 53

Miller, J. F. and Thomson, P. (2000), Cartesian Genetic Programming, *in* R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin and T. C. Fogarty, eds, 'Genetic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 121–132. 17

Mizoguchi, J., Hemmi, H. and Shimohara, K. (1994), Production genetic algorithms for automated hardware design through an evolutionary process, *in* 'Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence', pp. 661–664 vol.2. 27

Murakawa, M., Yoshizawa, S., Kajitani, I., Furuya, T., Iwata, M. and Higuchi, T. (1996), Hardware evolution at function level, *in* H.-M. Voigt, W. Ebeling, I. Rechenberg and H.-P. Schwefel, eds, 'Parallel Problem Solving from Nature — PPSN IV', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 62–71. 2, 23

Nicolau, M. (2017), 'Understanding grammatical evolution: Initialisation', *Genetic Programming and Evolvable Machines* **18**(4), 467–507.
**URL:** *https://doi.org/10.1007/s10710-017-9309-9* 49, 65

Nicolau, M. and Agapitos, A. (2018), *Understanding Grammatical Evolution: Grammar Design*, Springer International Publishing, Cham, pp. 23–53. 90

O'Neill, M. and Brabazon, A. (2004), Grammatical swarm, *in* K. Deb, ed., 'Genetic and Evolutionary Computation – GECCO 2004', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 163–174. 13

O'Neill, M. and Ryan, C. (2000), Crossover in grammatical evolution: A smooth operator?, *in* R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin and T. C. Fogarty, eds, 'Genetic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 149–162. 15, 90

O'Neill, M. and Ryan, C. (2004), Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code, *in* M. Keijzer, U.-M. O'Reilly, S. Lucas, E. Costa and T. Soule, eds, 'Genetic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 138–149. 126

O'Neill, M., Ryan, C., Keijzer, M. and et al. (2003), 'Crossover in grammatical evolution', *Genetic Programming and Evolvable Machines* **4**(1), 67–93. 90, 96

Osojnik, A., Panov, P. and Džeroski, S. (2017), 'Multi-label classification via multi-target regression on data streams', *Machine Learning* **106**(6), 745–770.
**URL:** *https://doi.org/10.1007/s10994-016-5613-5* 91

Rabey et al. (2003), *Digital Integrated Circuits (2nd Edition)*, Pearson. 1

Reyes, O., Moyano, J., Luna, J. and Ventura, S. (2018), A gene expression programming method for multi-target regression, *in* 'Proceedings of the International Conference on learning and optimization algorithms: theory and applications', LOPAL '18, ACM, pp. 1–6. 94

Rothlauf, F. and Oetzel, M. (2006), On the locality of grammatical evolution, *in* 'EuroGP'. 90

Ryan, C. and Azad, R. M. A. (2003), Sensible initialisation in grammatical evolution, *in* A. M. Barry, ed., 'GECCO 2003: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference', AAAI, Chigaco, pp. 142–145. 13, 14

Ryan, C., Collins, J. and Neill, M. O. (1998), Grammatical evolution: Evolving programs for an arbitrary language, *in* W. Banzhaf, R. Poli, M. Schoenauer and T. C. Fogarty, eds, 'Genetic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 83–96. 3, 10, 16, 90

Ryan, C., O'Neill, M. and Collins, J., eds (2018), *Handbook of Grammatical Evolution*, 1 edn, Springer Cham.
**URL:** *https://doi.org/10.1007/978-3-319-78717-6* 7

Ryan., C., Tetteh., M. and Dias., D. (2020), Behavioural modelling of digital circuits in system verilog using grammatical evolution, *in* 'Proceedings of the 12th International Joint Conference on Computational Intelligence - ECTA,', INSTICC, SciTePress, pp. 28–39. 36, 47

Ryan, C., Tetteh, M., McEllin, J., Mota-Dias, D., Conway, R. and Naredo, E. (2022), Addc: Automatic design of digital circuit, *in* S. Ventura, J. M. Luna and J. M. Moyano, eds, 'Genetic Algorithms', IntechOpen, Rijeka, chapter 4.
**URL:** *https://doi.org/10.5772/intechopen.104410* 31

Sekanina, L. (2006), Evolutionary design of digital circuits: Where are current limits?, *in* 'First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)', pp. 171–178. 2, 21

Sekanina, L., Walker, J. A., Kaufmann, P. and Platzner, M. (2011), *Evolution of Electronic Circuits*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 125–179. 17, 48

Senthilkumar, K., Kumarasamy, K. and Dhandapani, V. (2020), 'Approximate multipliers using bio-inspired algorithm', *Journal of Electrical Engineering & Technology* **16**, 559–568. 24

Slaný, K. and Sekanina, L. (2007), Fitness landscape analysis and image filter evolution using functional-level cgp, *in* M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi and A. I. Esparcia-Alcázar, eds, 'Genetic Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 311–320. 24

Slowik, A. and Białko, M. (2006), 'Evolutionary design and optimization of combinational digital circuits with respect to transistor count', *Bulletin of the Polish Academy of Sciences: Technical Sciences* **54**. 93

Soleimani, P., Sabbaghi-Nadooshan, R., Mirzakuchaki, S. and Bagheri, M. (2011), 'Using genetic algorithm in the evolutionary design of sequential logic circuits', *CoRR* **abs/1110.1038**.
**URL:** *http://arxiv.org/abs/1110.1038* 50

Soliman, A. and Abbas, H. (2004), Synchronous sequential circuits design using evolutionary algorithms, *in* 'Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)', Vol. 4, pp. 2013–2016 Vol.4. 50

Stoica, A., Zebulum, R. and Keymeulen, D. (2000), Mixtrinsic Evolution, *in* J. Miller, A. Thompson, P. Thomson and T. C. Fogarty, eds, 'Evolvable Systems: From Biology to Hardware', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 208–217. 21

Stomeo, E., Kalganova, T. and Lambert, C. (2006*a*), 'Generalized disjunction decomposition for evolvable hardware', *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* **36**(5), 1024–1043. 22, 26, 68

Stomeo, E., Kalganova, T. and Lambert, C. (2006*b*), Generalized disjunction decomposition for the evolution of programmable logic array structures, *in* 'First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06)', pp. 179–185. 86

Stuart Sutherland, Simon Davidmann, P. F. (2006*a*), *A Guide to Using SystemVerilog for Hardware Design and Modeling*, in , chapter Chapter 6: SystemVerilog Procedural Blocks, Tasks and Functions and Chapter 7: SystemVerilog Procedural Statements, pp. 137–206. 33, 34

Stuart Sutherland, Simon Davidmann, P. F. (2006*b*), *A Guide to Using SystemVerilog for Hardware Design and Modeling*, Springer US. 34

Sutherland, S. (2017), *RTL Modeling with SystemVerilog For Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design*, Sutherland HDL, Inc., Tualatin, Oregon. 32, 33

Tetteh, M., Dias, D. M. and Ryan, C. (2022), 'Grammatical evolution of complex digital circuits in systemverilog', *SN Computer Science* **3**(3), 188.
**URL:** *https://doi.org/10.1007/s42979-022-01045-9* 67

Tetteh, M. K., Mota Dias, D. and Ryan, C. (2021), Evolution of complex combinational logic circuits using grammatical evolution with systemverilog, *in* T. Hu, N. Lourenço and E. Medvet, eds, 'Genetic Programming', Springer International Publishing, Cham, pp. 146–161. 36, 67

Thompson, A. (1996), An evolved circuit, intrinsic in silicon, entwined with physics, *in* 'ICES'. 2, 21

Thompson, A. (1997), An evolved circuit, intrinsic in silicon, entwined with physics, *in* T. Higuchi, M. Iwata and W. Liu, eds, 'Evolvable Systems: From Biology to Hardware', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 390–405. 21, 22, 23, 28

Torresen, J. (1998), A divide-and-conquer approach to evolvable hardware, *in* M. Sipper, D. Mange and A. Pérez-Uribe, eds, 'Evolvable Systems: From Biology to Hardware', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 57–65. 25

Torresen, J. (2001), Two-step incremental evolution of a prosthetic hand controller based on digital logic gates, *in* Y. Liu, K. Tanaka, M. Iwata, T. Higuchi and M. Yasunaga, eds, 'Evolvable Systems: From Biology to Hardware', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–13. 25

Vasicek, Z. and Sekanina, L. (2014), How to evolve complex combinational circuits from scratch?, *in* '2014 IEEE International Conference on Evolvable Systems', pp. 133–140. 27, 68

Vasicek, Z. and Sekanina, L. (2015), 'Evolutionary approach to approximate digital circuits design', *IEEE Transactions on Evolutionary Computation* **19**(3), 432–444. 24, 48

Vassilev, V., Job, D. and Miller, J. (2000), Towards the automatic design of more efficient digital circuits, *in* 'Proceedings. The Second NASA/DoD Workshop on Evolvable Hardware', pp. 151–160. 2

Vassilev, V. K. and Miller, J. F. (2000), Embedding Landscape Neutrality To Build a Bridge from the Conventional to a More Efficient Three-bit Multiplier Circuit, *in* 'Proc. Genetic and Evolutionary Computation Conference', Morgan Kaufmann. 24, 28, 48

Walker, J. A., Völk, K., Smith, S. L. and Miller, J. F. (2009), 'Parallel evolution using multi-chromosome cartesian genetic programming', *Genetic Programming and Evolvable Machines* **10**(4), 417–445.
**URL:** *https://doi.org/10.1007/s10710-009-9093-2* 93

Williams, J. M. (2014), *Digital VLSI Design with Verilog*, 2nd edn, Springer Cham.
**URL:** *https://doi.org/10.1007/978-3-319-04789-8* 75

Williams, S. and Baxter, M. (2002), 'Icarus verilog: Open-source verilog more than a year later', *Linux J.* **2002**(99), 3–.
**URL:** *http://dl.acm.org/citation.cfm?id=513581.513584* 21

Wolf, C. (n.d.), 'Yosys open synthesis suite', `http://www.clifford.at/yosys/`. 83

Wolf, C. and Lasser, M. (2015), 'Project icestorm', `http://bygone.clairexen.net/icestorm/`. 23

Yao, X. and Higuchi, T. (1997), Promises and challenges of evolvable hardware, *in* T. Higuchi, M. Iwata and W. Liu, eds, 'Evolvable Systems: From Biology to Hardware', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 55–78. 2, 21

Zdenek, V. (2018), *Bridging the Gap Between Evolvable Hardware and Industry Using Cartesian Genetic Programming*, Springer International Publishing, Cham, pp. 39–55.
**URL:** *https://doi.org/10.1007/978-3-319-67997-6_2* 22

Zebulum, R. S., Stoica, A. M., Keymeulen, D., Sekanina, L., Ramesham, R. and Guo, X. (2005), Evolvable hardware system at extreme low temperatures, *in* 'ICES'. 2, 21

Zhen, X., Yu, M., He, X. and Li, S. (2018), 'Multi-target regression via robust low-rank learning', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **40**(2), 497–504. 90