Grammatical Evolution of Synthesizable Finite State Machine-Based Behavioural Level Hardware Description Language Codes

Bilal Majeed[1][a], Jack McEllin[1][b], Rajkumar Sarma[1][c], Ayman Youssef[2][d], Douglas Mota Dias[3][e] and Conor Ryan[1][f]

[1]*BDS Labs, Dept. of CSIS, University of Limerick, Limerick, Ireland*

[2]*Dept. of Computers and Systems, Electronics Research Institute, Cairo, Egypt*

[3]*Department of Computer Science & Applied Physics, Atlantic Technological University, Galway, Ireland*

{bilal.majeed, jack.mcellin, rajkumar.sarma, conor.ryan}@ul.ie, aymanmahgoub@eri.sci.eg, douglas.motadias@atu.ie

Keywords: Evolvable Hardware, Grammatical Evolution, Synthesizable Sequential Logic Circuits, Hardware Description Language Design, Electronic Design Automation.

Abstract: The importance of designing efficient and accurate digital circuits has grown due to the widespread use of wearable, ready-made, and custom electronic products. These digital circuits are typically sequential and designed using synthesizable Hardware Description Languages (HDLs) that can be translated into hardware. A large part of this exercise comprises designing synthesizable HDLs for sequential circuits, which are challenging to design and test, thus requiring much time for the engineers to construct them. This paper proposes using Grammatical Evolution (GE) to evolve the synthesizable HDL codes for sequential circuits on the behavioural or algorithmic level in SystemVerilog. The codes evolved in this work are of JK-Flip Flop (JK-FF), 3-bit Up-Down Counter (UDC), and 8-Floor Elevator (8FE), all from the perspective of Finite State Machines (FSMs). Circuits such as 3-bit UDC and JK-FF are the basic blocks in many circuits in the industry, while 8FE is a real-life example mimicking 3-bit UDC but with a few practical exceptions. All circuits are evolved using two types of grammars. The G1 Type Grammar evolves parts of the code, while the more powerful and generic G2 Type Grammar evolves the full HDL codes for these sequential circuits. The GE-based evolution of these synthesizable design codes using both types of grammar achieves a success rate of over 86% for all circuits. Moreover, all the solution circuits evolved with the best achieved success score under the respective hyper-parameter settings for G1 and G2 Type Grammar are synthesised, and their synthesis reports are compared against the synthesis reports of Gold (human-designed) circuits. The synthesis is performed using Cadence Genus at Generic Process Design Kit (GPDK) 45, 90, and 180 nm technology libraries. The synthesis results show that machine-generated designs often perform as well as or better than human-designed circuits.

1 INTRODUCTION

The foundation of contemporary electronics is digital circuits, which enable the processing and modification of digital signals. In contrast to continuous levels in analog circuits, binary digits (‘0’ and ‘1’) represent discrete levels in digital circuits. Logic gates such as AND, OR, and NOT, which carry out logical operations on binary inputs, are the essential components of digital circuits. Combining these gates results in more sophisticated circuits that may execute various computing functions, including flip-flops, registers, and arithmetic units. Because of their binary structure, digital circuits are very dependable in various applications, from microprocessors and memory modules in computers to microcontrollers in everyday devices.

Digital circuits are divided into two main categories depending on their structure and function: combinational and sequential. Digital circuits that rely only on the current input values for their output are known as combinational circuits. Past inputs and outputs are neither stored nor remembered. The circuit's logical functions combine the current inputs to produce the output. The logic gates noted above are the basic building blocks of combinational circuits

[a] https://orcid.org/0000-0001-7528-275X
[b] https://orcid.org/0000-0002-0187-9614
[c] https://orcid.org/0000-0002-5551-1006
[d] https://orcid.org/0000-0001-6145-4071
[e] https://orcid.org/0000-0002-1783-6352
[f] https://orcid.org/0000-0002-7002-5815

Majeed, B., McEllin, J., Sarma, R., Youssef, A., Dias, D. and Ryan, C.
Grammatical Evolution of Synthesizable Finite State Machine-Based Behavioural Level Hardware Description Language Codes.
DOI: 10.5220/0012948300003837
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 16th International Joint Conference on Computational Intelligence (IJCCI 2024)*, pages 222-233
ISBN: 978-989-758-721-4; ISSN: 2184-3236
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

(AND, OR, NOT, etc.). Boolean functions are implemented via the connections between these gates.

Unlike combinational circuits, sequential circuits feature memory components, implying that the circuit's outputs depend on the present inputs and their past states. These circuits can retain data and make decisions depending on previous inputs. Key components of sequential circuits include flip-flops and latches. By storing binary data, these memory components enable the circuit to maintain state information across time. Sequential circuits are mainly represented as FSMs to show the data flow of the circuit through the state transitions depending on the current state and/or the change in inputs of the circuit. This way of representing the sequential circuits is suggested since it shows how these circuits should behave at each clock cycle according to the current state of the FSM (Morris and Ciletti, 2007).

Most common sequential circuits include memory units such as Flip Flops, counters (which can count up or down or both), and registers. Out of these circuits, JK-FF and 3-bit UDC are evolved in this work. A JK-FF is a type of bistable multivibrator, a fundamental building block in digital electronics used for storing and transferring binary data. The JK-FF is a refinement of the SR (Set-Reset) Flip-Flop and eliminates the ambiguous state found in SR Flip-Flops (Morris and Ciletti, 2007). 3-bit UDC is a digital sequential circuit that counts up ('0' to '7') or down ('7' to '0') in binary, based on the control input. When operating as an Up counter, it starts from '0' and counts up to '7', then wraps around back to '0' to begin again. As a Down counter, it starts from '7' and counts down to '0', then wraps around back to '7' to start anew.

The 8FE circuit evolved in this work is a practical example of a counter-like machine used to control the lift/elevator system, yet it is complex to design and hard to evolve. It mimics the 3-bit UDC with the practical exception of being unable to wrap around at Floor-0 or Floor-7. So, each clock cycle can move up one floor, down one floor, or stay on the same floor, depending on the control input. Unlike a 3-bit UDC jumping directly from '0' to '7', it cannot leap directly from Floor-0 to Floor-7 or vice versa.

FSMs are essential for modeling behavior and managing states based on inputs in sequential circuit design. There are two types of FSM: Mealy machines, where outputs depend on both the current state and inputs, and Moore machines, where outputs depend only on the current state. Moore machines typically require more states than Mealy machines (Klimovich and Solov'ev, 2010). The JK-FF is implemented using a Moore machine, as shown in Figure 1. Flip-flops are not typically represented as FSMs; if needed, they are represented as Moore machines since Mealy machines maintain a single state. A single-state FSM lacks dynamic behavior crucial for modeling state changes based on input events.

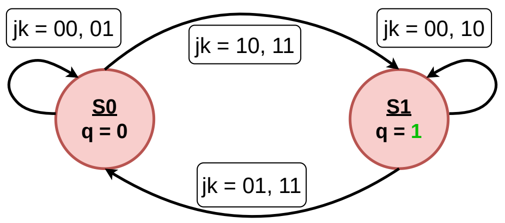
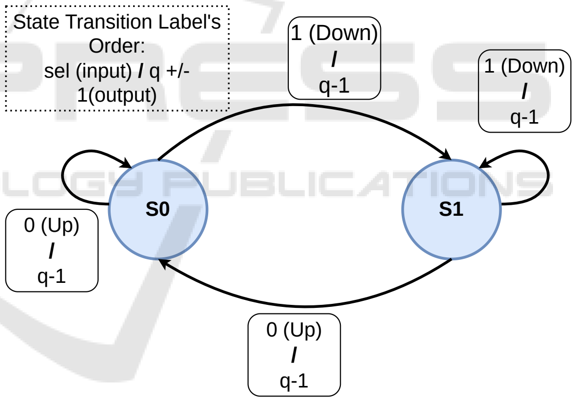


Figure 1: Moore FSM of JK-FF.

The FSMs of the 3-bit UDC and 8FE evolved in this paper are Mealy machines and are shown in Figure 2 and Figure 3, respectively. Typically, these machines would have eight states each, but their FSMs are optimized to generate a compact and optimized system in terms of HDL code. Since 8FE cannot jump directly from Floor-0 to Floor-7 or from Floor-7 to Floor-0, it is shown in Figure 3 that at each state, it is working on two inputs, one of which is keeping track of the current floor where the elevator is standing.


State Transition Label's Order:
sel (input) / q +/- 1(output)
1 (Down) / q-1
1 (Down) / q-1
S0
S1
0 (Up) / q-1
0 (Up) / q-1
Figure 2: Mealy FSM of 3-bit UDC.

HDLs are specialized programming languages that model and describe digital circuits and systems. Modern circuits are built, with very few exceptions, using HDLs rather than logic gates; this relationship is analogous to that between high-level programming languages and assemblers. These languages offer a greater abstraction than typical programming languages, giving engineers a systematic and disciplined means to represent digital circuits' timing, structure, and behavior. Verilog (Ciletti, 2010) and VHDL (Navabi, 2007) are the two most popular HDLs. Verilog is renowned for its straightforward, C-like syntax, which makes it comparatively simple to learn, but VHDL is verbose and provides a more explicit way to express hardware elements. Sys-

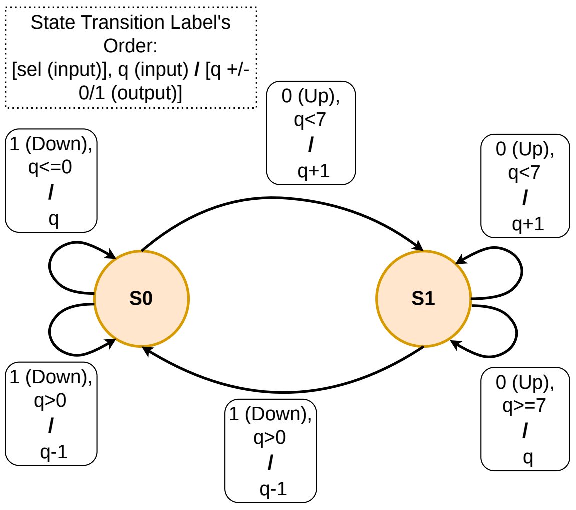


Figure 3: Mealy FSM of 8FE.

temVerilog (Spear, 2008) is another widely used HDL that enables engineers to use complicated testbench topologies and random stimuli in simulation to validate the design, thus working as both a Hardware Verification Language (HVL) and an HDL, which is why it is used here for the evolution of the circuits.

Logic synthesis follows the generation of HDL code for circuits. The HDL code offers a high-level abstraction of the circuit at the Register Transfer Level (RTL). During logic synthesis, this description is converted into an equivalent gate-level netlist of registers and/or Boolean equations, representing the circuit's logical structure.

Logic synthesis is typically performed using tools like Genus by Cadence (Cadence, 1998). These tools analyse HDL code and optimise it for performance, area, and power consumption. Logic synthesis also considers technology parameters, like transistor sizes, based on the target manufacturing process and environmental requirements. A logically perfect circuit from functional simulators may become non-synthesizable (cannot be implemented on a chip) if it does not use synthesizable HDL constructs, such as consistent use of blocking (=) and non-blocking (<=) HDL assignments in sequential code. Issues can be more complex, so it's crucial to create synthesizable code, as synthesis follows creating logically correct code.

Artificial Intelligence (AI), particularly Machine Learning (ML), has shown promising capabilities in automatic problem-solving across various fields, leading to its growing application in circuit design. In Electronic Design Automation (EDA) tools, which are crucial for designing electronic circuits, some can automatically generate specific circuit parts. Currently, ML-based systems like Solido (Solido, 2005), and synthetic intelligence tools like Eagle (Eagle, 1988) and Kicad (Kicad, 1992), exist. However, these systems cannot generate entire circuits and are primarily used for designing specific components only.

Search-based methods from the field of Evolutionary Computation (EC), such as Genetic Algorithms (GA) (Mirjalili, 2019), Genetic Programming (GP) (Koza, 1992), Evolutionary Strategies (ES) (Rudolph, 2012), and GE (Ryan et al., 1998) have shown massive success in automatically generating parts or complete circuits. The field of study that focuses on the use of EC for the automatic generation of electronics is called Evolvable Hardware (EH). GE is used here since it is, so far, best to evolve the sequential circuits' hardware on a behavioural level in HDL due to its ability to evolve behavioural level code with a great success rate (Majeed. et al., 2023) and does not suffer from issues such as the problem of closure, which GP does even if we use grammar-based GP.

2 RELATED WORK ON SEQUENTIAL CIRCUIT EVOLUTION

The first work was presented on the evolution of sequential circuits by (Hemmi et al., 1996) where a sequential adder was evolved using Production Genetic Algorithm (PGA) (Mizoguchi et al., 1994), a unique type of GA specially designed for EH and which used the production rules written in the form of Backus Naur Form (BNF). The designers called this an HDL grammar since it was designed to evolve circuits in HDL. Their mapping process differs from GE, although it also employs BNF grammars.

Next, two more sequential circuits, a modulo-6 counter and the ISCAS'89 benchmark circuit named lion, were evolved using Developmental Cartesian Genetic Programming (DCGP) (Shanthi et al., 2005). Cartesian Genetic Programming (CGP) (Miller, 2011), which was initially designed to evolve gate-level combinational circuits, is an evolutionary algorithm that represents programs or circuits as directed acyclic graphs, providing a flexible and efficient way to encode and manipulate solutions to various problems. DCGP is an enhanced form of CGP, which uses two levels of evolution. The first level identifies the best solution in the search space with the most negligible hazards. Three types of hazards could be found here: static hazard, which occurs when a signal changes twice in a row while it should change only once; dynamic hazard, which occurs when a signal changes several times in a row while it should change only once; and functional hazard, which usually occurs when two inputs change simultaneously. The second level of evolution aims to eliminate any such pending hazards. Involving two evolutionary stages, the proposed work cannot provide an end-to-end automatic system to generate problem-specific sequential circuits.

(Liang et al., 2009) presented a 3-stage decomposition system for the evolution of complex sequential circuits where the sequential circuit is decomposed into state decomposition, input decomposition, and output decomposition. Each stage is separately evolved using ES. They have evolved a series of sequential logic circuits from the Microelectronics Center of North Carolina (MCNC) benchmark library. However, they evolved the circuits at the gate level, which is not scalable to complex and large circuits. Additionally, no circuits were evolved using single-stage direct evolution, unlike in this work.

In another work, a 7-bit sequence signal generator was evolved (Zhiwu et al., 2011) using a fully connected feed-forward neural network, where gates such as XOR and NAND served as basic modules, and then the primary circuits were built on top of them. This work used the GA to evolve this neural network through which the signal generator was created.

Over time, (Xiong and Rafla, 2009), (Tao et al., 2012), (Majeed. et al., 2023), and (Majeed et al., 2023) presented their work on the automatic generation of Sequence Detectors (SDs) through evolutionary methods. SDs are crucial in digital systems and can be used to trigger alarms when sensing a specific sequence. They are also used to detect the specific sequence of events and keep track of them. Of these works, only our previous works (Majeed. et al., 2023) and (Majeed et al., 2023) evolved the circuits at the behavioural level, while all others evolved them at the gate level.

No previous work has attempted to evolve FFs from the perspective of an FSM. However, FFs are the modules for many EA- or other ML-based sequential circuit designs and synthesis since they are the essential memory elements in any sequential circuit. It makes them equally important to be evolved since they can save area, power, delay, and, most importantly, the design time and effort for complex circuits if evolved than using a human-designed or hardcoded module.

In (Manovit et al., 1998), the gate level code of the FSM of a 3-bit UDC (mentioned as Reversible 8-counter in their work) along with some other sequential circuits such as a serial adder and frequency divider was evolved, presented with enhanced correctness (Chongstitvatana and Aporntewan, 1999). It was evolved online on an Field Programmable Gate Array (FPGA) (Aporntewan and Chongstitvatana, 2001) using a GA. In a later work, another 3-bit Up counter was evolved at the gate level using GA (Soliman and Abbas, 2004). They evolved it using two different techniques; in the first round, they evolved just the combinational part, while the entire sequential circuit evolved in the second. No counter in these works has evolved on a behavioural level from the perspective of an FSM, which means that, specifically, an FSM has not evolved.

The work shown in (Lucas and Reynolds, 2003) evolves the transition matrix of an elevator using ES and shows that their work outperforms the Evidence-Driven State Merging (EDSM) algorithm, which is considered a premium FSM learning algorithm. An FSM was evolved in (Tsarev and Egorov, 2011) using a genetic algorithm that caters to control the doors of an elevator system. This FSM has three states, five different kinds of inputs, two outputs, and seven transitions. However, this work does not evolve any gate or behavioural level code for this FSM. Multiple works are shown for either run-time or offline ML-based fault diagnosis of elevator systems such as (Bao et al., 2012) and (Zhang et al., 2022). Also, many works are shown on either manual or automatic optimization of elevator control and performance, such as (Markon et al., 2006) and (Pham et al., 2015). However, no work has automatically generated either the gate level or behavioural level code for the elevator FSM through evolution or any other method.

In this work, we propose the GE-based evolution of the synthesizable behavioural level HDL codes of the FSMs of three crucial sequential circuits named JK-FF, 3-bit UDC, and 8FE using specially designed training and test data sets (explained in Section 4). To the authors' knowledge, this is the first work to evolve these sequential circuits using GE. In addition, all the circuits in this work are evolved using single-stage evolution on the behavioural level, so none of the Divide-and-Conquer techniques, such as Bidirectional Incremental Evolution (BIE) (Kalganova, 2000), or any Generalised Disjunction Decomposition (GDD) (Stomeo et al., 2006) are used in this work. This work emphasises evolving the behavioural-level code and not the gate-level code due to the scalability issues faced in gate-level codes while designing a complex system. Designing complex circuits using gate-level codes at one stage becomes almost impossible due to the exponential growth of gate-level instances with increased complexity in digital circuits. Such lengthy gate-level codes are too challenging to handle when fixing bugs and making modifications.

3 GRAMMATICAL EVOLUTION

GE (Ryan et al., 1998) employs BNF grammars to map genotypes to phenotypes. BNF is a formal notation used to describe the syntax of programming languages, data structures, and communication protocols. It provides a way to express the rules for the structure of a language in a precise and unambiguous manner. In BNF, a language is defined through a series of production rules, each specifying how a particular syntactic element can be composed of other elements. Utilizing context-free grammar, GE evolves structures/objects in any programming language. Successful applications include combinational (Youssef et al., 2021) and sequential circuit design (Majeed. et al., 2023), symbolic regression (Ali et al., 2021), and classification (Murphy et al., 2021). GE can generate HDL codes for circuits, enabling analysis of efficiency and power/hardware usage.

BNF grammars consist of a four-tuple $< N, T, P, S >$. That is a set of non-terminals N, which act as placeholders that can be expanded into other symbols. T represents the set of terminal symbols, which are the concrete symbols or tokens of the language and cannot be further expanded. P is a set of production rules (also called rules) that replace a N with another N or T. Finally, S is the starting symbol, which is a specific N from which the mapping starts.

We have three rules in the example shown in Figure 4 where N are the grammar parts found on either side of equality, which can be further mapped to a symbol, such as *<var>* (on the RHS of the first P and the LHS of the third P in the example), which is further mapped to T. In contrast, T, as mentioned before, cannot be further mapped and appears only on the right side, e.g., '!' and ' & ' in the RHS of the second P. In the example shown here, leftmost *<exp>* in the first P serves as S.

The example shown in Figure 4 illustrates the genotype to phenotype mapping at the gate level for HDL circuit design. Each 8-bit chunk, such as '00101000', is converted to the respective decimal, '40' in this case, and decimal values expand the P of grammar until expressions with only T are reached. The example depicts logic gate generation between input variables x and y, yielding AND, OR, and NOT gates as options. It can be seen that starting from the first rule, a modulus of '40' is computed with '2' since there are only two options available in that rule on the RHS separated by '|'. As a result of '40%2' equal to '0', the first option available on the RHS of that rule is selected. Now, suppose this selected option is comprised of more than one N. In that case, the leftmost will be expanded first, and to expand that N, the modulus of the following decimal (coming from the next chunk of 8-bits) will be computed with the number equal to the available options on the RHS of the rule used to map this N. This process continues until an expression has only T in it. Note that the NOT gate, denoting inversion, can only be applied to a single variable (e.g., $!x$). The final expression in this example implies an OR gate between the circuit's input variables, x and y, with a detailed explanation available for this example in our previous work (Majeed. et al., 2023).

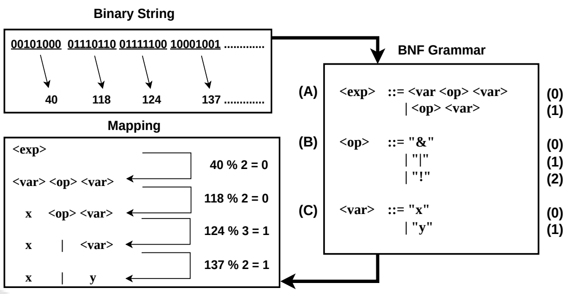


Figure 4: Genotype to phenotype mapping in GE.

4 TRAINING AND TEST DATASET GENERATION

The data set used in this work comprises specialized exhaustive training cases where the system is set to a specific state of its FSM and then examined separately for random and shuffled as well as a minimal required combination of all the inputs. For example, in the case of the 8FE, the system is set to its first state (S0) and then checked for the random inputs such as [0101101001xx1010] (shown in the data set diagram given in the Supplementary Material (SM)[1] where '0' reflects move up a floor, '1' reflects move down a floor, and 'x' reflects stay on the same floor). After a reset of the whole system, the system is again set to S0 and checked for a minimum and in-lined required combination to complete a clockwise and anti-clockwise cycle such as [0000x0001111x111], which checks its whole cycle from Floor-0 to Floor-7 and then back to Floor-0 from Floor-7. The inputs shown as 'x' check the system for unfortunate conditions where the input system can fail or if the system is in the idle state. In such a case, the elevator will stay on the same floor, or for this kind of input, the counter will stay in the same state until reset is applied. The same approach is used for the training data set of 3-bit UDC. The training data set of JK-FF is different

[1]Supplementary Material: https://github.com/bmmajeed/Basic_Sequential_CKTS_ECTA24

since we take its initial state from the training case as well, but we cover all the maximum input combinations according to the values of ‘j’ and ‘k’, which are 40 in total (shown in Table 1). However, taking the maximum input combinations for the other two circuits is impossible since they involve random inputs, including ‘x’, to mimic their real-life usage.

According to the minimum requirement of the training dataset, the number of used training/test cases and the length of each case are shown in Table 1 for JK-FF, 3-bit UDC, and 8FE, respectively. Note that for all three circuits, as shown in Table 1, the length of the training vector is one bit larger than the test vector since we are not setting the system to a specific state in the test dataset. This extra bit is only used for the exhaustive training of the system, so it can be trained practically to take a perfect start from any state at any time.

Table 1: Size and length of train/test data set used for evolution.

Circuits	**No. of Train/ Test Vectors**	**Each Vector’s Length in bits**	
		Training	**Test**
JK-FF	40	5	4
UCD	80	6	5
8FE	128	6	5

The advantage of this approach can be seen in the results in Table 3 and Table 4, which show that all solutions evolved using the training data set perform excellently on the unseen test data set, which has the same structure but does not have the one bit used to set the system at a specific state only for training. It is a significant step forward compared to the results shown in the complex sequential circuits’ FSM evolution given in our previous works (Majeed. et al., 2023) and (Majeed et al., 2023), where the perfect success rate on the test dataset was never achieved for sequential circuits’ evolution.

```
                  .
                  .
<set_state>     ::=
        "if (state_in || !state_in)
            state = state_in;
        else \n\t\t\t
            state = state;"
<states_block>  ::=
        "if (state == "<states>")
        begin
            q = "<values>";"
            <conditional> "end"
                  .
                  .
<conditional>   ::=
        "if ("<vars>" == "<values>")
            state = "<states>";
        else
            state = "<states>";"
                  .
                  .
<vars>          ::= "j"  | "k"
<states>        ::= "S0" | "S1"
<values>        ::= "0"  | "1"
```

Figure 5: BNF grammar having full structure of the HDL code for FSM to evolve the current state, input variables, input values, output values, and the next state of JK-FF.

5 EXPERIMENTS AND RESULTS

5.1 Experimental Setup, Tools, and Evolutionary Parameters

Each of the three circuits is evolved using two different BNF grammars, G1 Type Grammars and G2 Type Grammars, which present constrained (G1 Type Grammars) and unconstrained spaces (G2 Type Grammars). G1 Type Grammars evolve five parts of the HDL sequential code: the present state, input variables, input values, output values, and the next state of the system. The G1 Type Grammars used for JK-FF (Figure 5) show that the structure of nested if-else statements in this grammar is given to the system to implement the FSM, and the specified bits are evolved. This grammar is used to evolve JK-FF, but the same structure and approach are used to evolve the 3-bit UDC and 8FE. G2 Type Grammars are much more general grammars that evolve the entire if-else code structure from scratch.

In both grammars, *<set_state>* is used to set the system at a specific state (discussed in detail in Section 4) after each reset, and then the system starts from there by giving the values of ‘j’ and ‘k’ as inputs. Note that after taking the input, ‘0’ or ‘1’ for the current state (‘state-in’) from the first training case and setting the system to that state, this input is set to ‘x’ (don’t care) in all the remaining training cases until the next reset of the system since this input is just used in first clock cycle to put the system in that specific state. The mapping in both grammars starts from *<states_block>* where it can be seen in Figure 5 that the structure of if else-if is hardcoded in G1 Type Grammars. At the same time, the conditionals of if and else-if, which are values of the current states, and the input variables (‘j’ and ‘k’) and input values (‘0’

```
                  .
                  .
<states_block>   ::= <if> <else> | <if> <else_ifs> <else>
<else_ifs>       ::= <else_if> | <else_if> <else_ifs>
<if>             ::= "if ("<lhs1>" == "<rhs>") begin"
                         <op_options> "
                     end" |
                     "if ("<lhs1>" == "<rhs>") begin"
                         <op_options>
                         <states_block> "
                     end"
                  .
                  .
<op_options>     ::= <lhs2>" <= "<rhs>";" | <lhs2>" <= "<rhs>";"
                                            <lhs2>" <= "<rhs>";"
<lhs1>           ::= "j" | "k" | "q" | "state"
<lhs2>           ::= "q" | "state"
<rhs>            ::= "0" | "1" | "S0" | "S1" | "j" | "k" | "q"
```

Figure 6: BNF grammar evolving the HDL code's full structure in addition to the current state, input variables, input values, output values, and the next state for an FSM of JK-FF.

and '1') are evolved. In addition, the values of output 'q' and the next state are also evolved using G1 Type Grammars.

In contrast, using G2 Type Grammars, the system evolves its whole nested if else-if structure and the appropriate conditionals and assignments, as seen in Figure 6. <*rhs*>, which represents the RHS of conditionals of if and else-if as well as of assignments coming from <*op_options*>, is the same wherever it is applied. However, the LHS is not the same and is divided in <*lhs*1> and <*lhs*2>. <*lhs*1> is used in the conditionals where 'j' and 'k' can come on the LHS of assignment since we are evaluating their value, but they cannot become a part of <*lhs*2> since they are defined as inputs and cannot be assigned as outputs of the circuit. If a circuit tries to do so, the system gives a syntax error and discards the phenotype. In contrast, <*rhs*> holds all the values and variables that can fit everywhere on the RHS of any assignment or conditional. Since all three FSMs evolved here have two states only, the RHS is happy to use the values of 'S0' and 'S1' as 1-bit '0' and 1-bit '1' (can be seen in Figure 9). If we use the same production rule in some more complex systems, which involve more than two states, then it might not take the state parameter having a 2-bit value of '11' at the RHS when LHS is defined to have a 1-bit value only.

To evolve the HDL codes for JK-FF, 3-bit UDC, and 8FE, an experimental setup was used, combining libGE (a C++ library for GE mapping) with Icarus Verilog, a simulator for Verilog/SystemVerilog, to evaluate the individuals. All experiments were run on a Dell OptiPlex 5070 Desktop computer comprising a single unit RAM of 16 GB, 1 TB HDD, and 256 GB SSD. It has a 64-bit quad-core 9th generation i7 processor with a 12MB cache processor.

For all the experiments, the evolutionary parameters which are kept the same are shown in Table 2. All experiments with the G1 Type Grammars use a population size of 100, and this parameter is only increased when required to get a better success rate for the 8FE, as shown in Table 3.

Table 2: Evolutionary parameters.

Parameter	Value
No. of Runs	30
Initialisation	Sensible
Crossover Probability	0.9 (One Point)
Mutation Probability	0.01
Parent Selection	Tournament
Elitism	Yes

Due to the ample search space of G2 Type Grammars, a larger population size of 1000 was used. Table 4 shows that JK-FF did not evolve even with this population size, so the maximum generations and population size were increased where required. For all the circuits, the maximum generations in the initial experiments using G1 Type Grammars were set to 30. In contrast, for experiments using G2 Type Grammars, this value was set to 60 and only increased where the error bars on the resulting graph showed room for significant improvement. In Table 3 and Table 4, the blue colour indicates the success rate on the training data set out of 30 runs, while the bold green colour indicates the success rate on the test data set out of the perfect solutions captured on the training data set.

For all the experiments with both types of grammar, a success rate equal to or better than 26/30 is considered good enough and acceptable. It means that if in 26 runs out of the 30 runs, we have evolved a perfect solution according to the training data set, we do not have to increase the number of individuals (maximum generations or population size) to get a better success rate than this. Not Needed (NN) is mentioned in Table 3 and Table 4 to indicate that the acceptable success rate is achieved and there is no need to run more experiments.

5.2 Evolution of the Circuits Using G1 Type Grammars

As described in section 5.1, the first part of the experiments ran for this work evolves the current state of the system, the input variables and their values, the output of the system, and the following states, using the structure of G1 Type Grammars. Population sizes of 100 and a maximum of 30 generations were used for the three circuits' starting points. Each experiment is run 30 times, and the success rates are presented accordingly, where the success rate reflects the number of runs in which the system produced a perfect solution.

For JK-FF, it can be seen in Table 3 that a high success rate of 29/30 is achieved. Since this is a high success and the resulting graph (all graphs discussed in this paper refer to the mean of maximum fitness values across generations) showed no progress in the evolution of the individuals due to the tiny error bars, no more experiments were run in this case.

For 3-bit UDC, with a population size of 100 and 30 generations, a success rate of 20/30 was achieved. The resulting graph (Figure 7) showed no ongoing progress through the significantly lengthened error bars. The second experiment was run with 50 generations, and a success rate of 26/30 was achieved to improve the score. Since the resulting graph (shown in Figure 8) showed no progress, and the success rate was good enough, no experiments were run further.

For the 8FE, with the population size of 100 and 30 generations, only two solutions were found out of 30 runs, so the number of generations was increased to 50 and 100, respectively. Still, a maximum success rate of 17/30 was achieved. The population was increased to 500 to improve the success rate, which increased to 30/30.

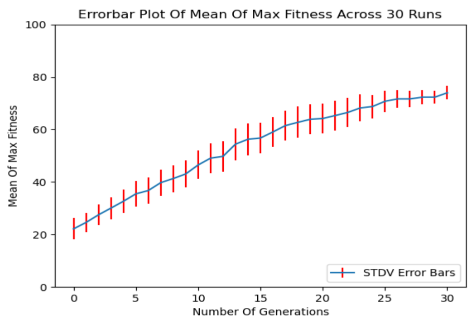


Figure 7: Mean of the best fitness values across generations recorded in the evolution of UDC using G1 Type Grammar with max. of 30 generations.

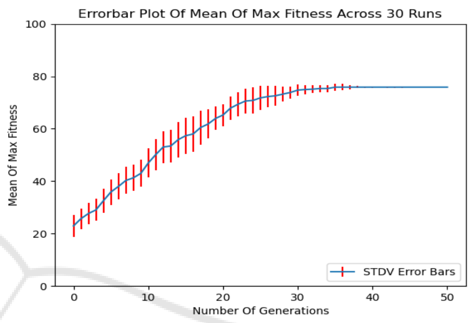


Figure 8: Mean of the best fitness values across generations recorded in the evolution of UDC using G1 Type Grammar with max. of 50 generations.

Since a success rate equal to or higher than 26/30 is considered good enough in this work, the computational cost in terms of time taken for the evolution to hit this success rate for the relevant circuits is shown in Table 5.

For all the experiments, all evolved solutions performed flawlessly on the test data set, which is one of the perks of using the method proposed to design the training data set.

5.3 Evolution of the Circuits Using G2 Type Grammars

As noted in Section 5.1, the experiments ran using G2 Type Grammars employed a starting population size of 1,000. With 60 generations, the success rate 08/30 was achieved for the JK-FF, so the number of generations was increased until we got no progress in the graph at the maximum generations of 300. After that, the population size was increased to 2000, and with the maximum generation of 150, a success rate of 27/30 was achieved (shown in Table 4), which was good enough, so no experiments were run further. An example code of a fully evolved sequential block of JK-FF using G2 Type Grammar is shown in Figure 9. Note that since the FSM used for this circuit is a 2-

Table 3: Results of evolved circuits using G1 Type Grammars, shown according to the number of generations and population size. (STrD: Success rate on training data set; STeD: Success rate on test data set of evolved solutions; NN: Not Needed).

Circuit	**Pop. Size = 100**						**Pop. Size = 500**	
	Max. Gens. = 30		***Max. Gens. = 50***		***Max. Gens. = 100***			
	STrD	**STeD**	**STrD**	**STeD**	**STrD**	**STeD**	**STrD**	**STeD**
JK-Flip Flop	29/30	**29/29**	NN	NN	NN	NN	NN	NN
Up-Down Counter	20/30	**20/20**	26/30	**26/26**	NN	NN	NN	NN
8-Floor Elevator	02/30	**02/02**	09/30	**09/09**	17/30	**17/17**	30/30	**30/30**

Table 4: Results of evolved circuits using G2 Type Grammars, shown according to the number of generations and population size.

Circuit	**Pop. Size = 1,000**						**Pop. Size = 2,000**	
	Max. Gens. = 60		***Max. Gens. = 150***		***Max. Gens. = 300***		***Max. Gens. = 150***	
	STrD	**STeD**	**STrD**	**STeD**	**STrD**	**STeD**	**STrD**	**STeD**
JK-Flip Flop	08/30	**08/08**	16/30	**16/16**	21/30	**21/21**	27/30	**27/27**
Up-Down Counter	27/30	**27/27**	NN	NN	NN	NN	NN	NN
8-Floor Elevator	15/30	**15/15**	29/30	**29/29**	NN	NN	NN	NN

Table 5: Time taken by the evolution of the circuits to hit a good enough success rate on the training dataset.

	Circuit	**STrD**	**No. of Individuals**	**Time (secs)**
G1	**JK**	29/30	15,000	6.15
	UDC	26/30	25,000	19.62
	8FE	30/30	50,000	659.67
G2	**JK**	27/30	300,000	4187.75
	UDC	27/30	60,000	528.91
	8FE	29/30	150,000	5075.67

state machine and according to the code definitions of states 'S0 = 0' and 'S1 = 1', you can see that 'S0' and 'S1' can be seen at the RHS of some assignments as well which are reflecting their 1-bit values of '0' and '1' respectively.

For 3-bit UDC, the first experiment with a population size of 1,000 and 60 generations achieved a success rate of 27/30, which is a high and good success rate, so no further experiments were run.

In the case of the 8FE, the first experiment with a population size of 1,000 and 60 generations gave a success rate of 15/30. The number of generations was then increased to 150, giving a success rate of 29/30, which is an acceptable success rate, so no further experiments were run. The time taken for the evolution to get an acceptable success rate for the relevant circuit is shown in Table 5.

Comparison of our evolutionary work presented in this work with either state-of-the-art presented works or works which are similar to some extent is shown in Table 6. This comparison is shown in terms of the evolutionary approach used in the work, the form or parts of the evolved design, the design type, and the number of individuals used per run during the evolution. Since there is no work shown in the literature where JK-FF evolved from the perspective of an FSM, we are unable to compare it with any other work. 3-bit

```
always @(posedge clk) begin
    if(!rst) begin
        if (state == 0) begin
            q <= k;
            q <= S0;
                if (j == S1) begin
                    state <= 1;
                end
                else begin
                    q <= S0;
                end
        end
        else if (state == k) begin
            q <= 1;
            state <= S0;
        end
        else begin
            q <= S1;
        end
    end
    else begin
        q <= 0;
        state <= S0;
    end
end
```

Figure 9: Fully evolved code of JK-FF using G2 Type Grammar.

UDC is compared to state-of-the-art works presented in the literature, and it can be seen that this is the first work which has evolved the complete FSM of 3-bit UDC on a behavioural level using the minimum individuals comparatively. A complete FSM for 8FE or any floor elevator has not evolved before. However, the evolution of 8FE is compared to a 7-state Discrete Finite Automata (DFA) of a floor elevator and is also

Table 6: Comparison with the state-of-the-art or somewhat similar works. (PCE: Partial code evolved; FCE: Full code evolved).

Circuit	Work	Evol. appr. used	Evolved form/parts of design	Design type	Number of indiv. used per run
JK-FF	No comparison given in literature				
	This work (PCE)	**GE**	**Complete FSM**	**Behavioral-level**	**3,000**
	This work (FCE)	**GE**		**Behavioral-level**	**300,000**
3-bit UDC	Manovit et al., 1998	GA	State transition functions	Gate-level	5,500,000
	Chongstitvatana et al., 1999	GA		Gate-level	5,000,000
	Aporntewan et al., 2001	GA		Gate-level	6,400,000
	Soliman et al., 2004	GA	Combinational and sequential cells	Gate-level	150,000
	This work (PCE)	**GE**	**Complete FSM**	**Behavioral-level**	**5,000**
	This work (FCE)	**GE**		**Behavioral-level**	**60,000**
DFA of floor elevator (not a circuit)	Lucas et al., 2003	ES	Transition matrix	-	1,000,000
Elevator door control circuit	Tsarev et al., 2011	GA	Complete FSM	-	2,000
8FE	This work (PCE)	**GE**	**Complete FSM**	**Behavioral-level**	**50,000**
	This work (FCE)	**GE**		**Behavioral-level**	**150,000**

compared to a 6-state elevator door control unit. It is highlighted that this work is the first to evolve any floor elevator using GE on a behavioural level.

6 SYNTHESIS COMPARISON OF GOLD AND EVOLVED CIRCUITS

This section evaluates the performance of synthesised circuits to demonstrate how well they compare with Gold circuits on actual silicon. Circuits meeting an acceptable success rate in G1 and G2 Type Grammars are selected for synthesis. Genus Synthesis Solution by Cadence uses GPDK 45, 90, and 180 nm Complementary Metal-Oxide Semiconductor (CMOS) technologies for synthesis. The circuits operate at a clock frequency of 100 MHz, with the reset signal configured to the ideal network in the constraints file. Synthesis is conducted at the Fast Corner (FC) and Slow Corner (SC) for all technologies. FC represents optimal operating conditions with factors like supply voltage and temperature at their most favourable, while SC simulates worst-case conditions with lower supply voltage or higher temperature. Table 7 provides details on temperature and supply voltage.

Since the complete synthesis reports contain extensive detail, a summary is provided in Table 8. This summary presents the percentage by which the performance of the evolved solutions' synthesis reports is better or worse than that of the Gold circuits. Percentages highlighted in bold and green indicate that

Table 7: Nominal temperature and supply voltage for relevant technologies.

Parameters	90 nm		180 nm	
	FC	**SC**	**FC**	**SC**
Temperature (°C)	0	125	0	125
Voltage (V)	1.1	0.9	1.98	1.62

the synthesis solutions of the evolved circuits outperformed the Gold circuits in terms of circuit area (cell count) and Power-Delay Product (PDP). Red indicates that the performance of the evolved circuits is worse than the Gold circuits, while blue signifies equivalent performance.

The summary table reveals that the evolved circuits of JK-FF generally performed on par with the Gold circuits based on the best values taken from the group of solution circuits in terms of both area and PDP. While the synthesis reports for UDC show that the evolved UDC circuits did not perform as well as the evolved JK-FF circuits, they still matched or surpassed the Gold UDC circuits over 70% of the time overall. However, the synthesis reports for 8FE indicate significantly improved performance of the evolved 8FE solutions compared to the other two evolved circuits, with evolved values surpassing the Gold circuits more than 95% of the time. Given that 8FE is a complex, real-world circuit, its superior synthesis performance underscores the credibility of using such a system for automatically synthesizable digital sequential design using HDL.

Table 8: Summary of the synthesis reports showing the performance of synthesized solutions in percentages.

Circuit	Param.	45 nm	90 nm	180 nm
JK-FF	**Area**	(**50**+50)%	100%	100%
	PDP	100%	100%	(**50**+50)%
UDC	**Area**	100%	(**50**+50)%	(50+50)%
	PDP	(75+25)%	(50+50)%	(50+50)%
8FE	**Area**	**100%**	(**75**+25)%	**100%**
	PDP	**100%**	**100%**	**100%**

7 CONCLUSION AND FUTURE WORK

The automatic partial and complete HDL code generation of three digital sequential circuits, JK-FF, 3-bit UDC, and 8FE, from the perspective of FSM, is presented here using GE as the ML engine on two different kinds of grammars for each circuit labelled as G1 Type Grammars and G2 Type Grammars. In addition, evolved solutions of all three circuits are synthesised to show that all the evolved circuits are perfectly synthesizable. This work uses a specialised method to generate training data sets, which gives a one hundred percent performance of evolved solutions on the unseen test data set. To the best of the author's knowledge, this is the first work that has evolved some of the highly used and crucial sequential circuits/modules from the perspective of an FSM, such as JK-FF, an essential memory element for digital circuits, as they could not find any work in literature which has done this. Also, this is the first work to evolve the full and partial synthesizable codes of all these three circuits on a behavioural level from the perspective of an FSM using an end-to-end automated system involving just one evolutionary stage. Future work will focus on evolving synthesizable full and partial HDL codes of more complex sequential circuits through the proposed system and the technique of training data set generation.

ACKNOWLEDGMENTS

This work was supported by the Science Foundation Ireland (SFI) grant 16/IA/4605.

REFERENCES

Ali, M., Kshirsagar, M., Naredo, E., and Ryan, C. (2021). Towards automatic grammatical evolution for real-world symbolic regression. In *Proceedings of the 13th International Joint Conference on Computational Intelligence - Volume 1: ECTA,*, pages 68–78. INSTICC.

Aporntewan, C. and Chongstitvatana, P. (2001). An on-line evolvable hardware for learning finite-state machine. In *Proceedings of International Conference on Intelligent Technologies*.

Bao, J. et al. (2012). Fault diagnostic method of elevator control system based on finite state machine. *Journal of Computer Applications*, 32(06):1692.

Cadence (1998). Cadence design systems. https://www.cadence.com/.

Chongstitvatana, P. and Aporntewan, C. (1999). Improving correctness of finite-state machine synthesis from multiple partial input/output sequences. In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pages 262–266.

Ciletti, M. D. (2010). *Advanced Digital Design with the Verilog HDL*. Prentice Hall Press, USA, 2nd edition.

Eagle (1988). Eagle by autodesk. https://www.autodesk.com/products/eagle/overview. [Online; accessed 09-Jan-20242].

Hemmi, H., Mizoguchi, J., and Shimohara, K. (1996). Development and evolution of hardware behaviors. In Sanchez, E. and Tomassini, M., editors, *Towards Evolvable Hardware (TEH)*, pages 250–265, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kalganova, T. (2000). Bidirectional incremental evolution in extrinsic evolvable hardware. In *Proceedings. The Second NASA/DoD Workshop on Evolvable Hardware*, pages 65–74.

Kicad (1992). Kicad electronic design automation. https://www.kicad.org/. [Online; accessed 09-Jan-20242].

Klimovich, A. S. and Solov'ev, V. V. (2010). Transformation of a mealy finite-state machine into a moore finite-state machine by splitting internal states. *Journal of Computer and Systems Sciences International*, 49(6):900–908.

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

Liang, H., Luo, W., and Wang, X. (2009). A three-step decomposition method for the evolutionary design of sequential logic circuits. *Genetic Programming and Evolvable Machines*, 10(3):231–262.

Lucas, S. and Reynolds, T. (2003). Learning dfa: evolution versus evidence driven state merging. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 1, pages 351–358 Vol.1.

Majeed, B., Carvalho, S., Dias, D. M., Youssef, A., Murphy, A., and Ryan, C. (2023). Performance upgrade of sequence detector evolution using grammatical evolution and lexicase parent selection method. In Collet, P., Gardashova, L., El Zant, S., and Abdulkarimova, U., editors, *Complex Computational Ecosystems*, pages 90–103, Cham. Springer Nature Switzerland.

Majeed., B., Ryan., C., McEllin., J., Youssef., A., Dias., D., Murphy., A., and Carvalho., S. (2023). Evolving behavioural level sequence detectors in systemverilog using grammatical evolution. In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence - Volume 3: ICAART*, pages 475–483.

Manovit, C., Aporntewan, C., and Chongstitvatana, P. (1998). Synthesis of synchronous sequential logic circuits from partial input/output sequences. In Sipper, M., Mange, D., and Pérez-Uribe, A., editors, *Evolvable Systems: From Biology to Hardware*, pages 98–105, Berlin, Heidelberg. Springer Berlin Heidelberg.

Markon, S., Kise, H., Kita, H., and Bartz-Beielstein, T. (2006). Optimal control of multicar elevator systems by genetic algorithms. *Control of Traffic Systems in Buildings*, pages 221–233.

Miller, J. F. (2011). *Cartesian Genetic Programming*, pages 17–34. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mirjalili, S. (2019). *Genetic Algorithm*, pages 43–55. Springer International Publishing, Cham.

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*, pages 661–664 vol.2, Orlando, FL, USA. IEEE.

Morris, M. and Ciletti, M. D. (2007). *Digital design*. Pearson Prentice Hall, New Jersey, USA.

Murphy, A., Murphy, G., Amaral, J., Mota Dias, D., Naredo, E., and Ryan, C. (2021). Towards incorporating human knowledge in fuzzy pattern tree evolution. In *European Conference on Genetic Programming (Part of EvoStar)*, pages 66–81. Springer.

Navabi, Z. (2007). *VHDL: Modular Design and Synthesis of Cores and Systems*. McGraw-Hill, New York.

Pham, T. H., Prodan, I., Genon-Catalot, D., and Lefèvre, L. (2015). Efficient energy management for an elevator system under a constrained optimization framework. In *2015 19th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 613–618.

Rudolph, G. (2012). *Evolutionary Strategies*, pages 673–698. Springer Berlin Heidelberg.

Ryan, C., Collins, J. J., and Neill, M. O. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *European Conference on Genetic Programming*, pages 83–96, Berlin, Heidelberg. Springer.

Shanthi, A., Singaram, L., and Parthasarathi, R. (2005). Evolution of asynchronous sequential circuits. In *2005 NASA/DoD Conference on Evolvable Hardware (EH'05)*, pages 93–96, Washington, DC, USA. IEEE.

Solido (2005). Solido design solutions. https://eda.sw.siemens.com/en-US/ic/solido/. [Online; accessed 09-Jan-20242].

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)*, volume 4, pages 2013–2016 Vol.4.

Spear, C. (2008). *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. Springer, New York, 2nd edition.

Stomeo, E., Kalganova, T., and Lambert, C. (2006). Generalized disjunction decomposition for evolvable hardware. *IEEE Trans Syst Man Cybern B Cybern*, 36(5):1024–1043.

Tao, Y., Cao, J., Zhang, Y., Lin, J., and Li, M. (2012). Using module-level evolvable hardware approach in design of sequential logic circuits. In *2012 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, New York. IEEE.

Tsarev, F. and Egorov, K. (2011). Finite state machine induction using genetic algorithm based on testing and model checking. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, page 759–762, New York, NY, USA. Association for Computing Machinery.

Xiong, F. and Rafla, N. I. (2009). On-chip intrinsic evolution methodology for sequential logic circuit design. In *2009 52nd IEEE International Midwest Symposium on Circuits and Systems*, pages 200–203, New York. IEEE.

Youssef, A., Majeed, B., and Ryan, C. (2021). Optimizing combinational logic circuits using grammatical evolution. In *2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 87–92, New York. IEEE, IEEE.

Zhang, S., Yin, Q., and Wang, J. (2022). Elevator dynamic monitoring and early warning system based on machine learning algorithm. *IET Networks*, n/a(n/a).

Zhiwu, Z., Jian'an, L., Xinfeng, C., and Liming, Z. (2011). Design of sequential logic circuits based on evolvable hardware. In *IEEE 2011 10th International Conference on Electronic Measurement Instruments (ICEMI)*, volume 3, pages 240–243.