Evolving and Analysing "Useful" Redundant Logic

Asbjoern Djupdal and Pauline C. Haddow

CRAB Lab
Department of Computer and Information Science
Norwegian University of Science and Technology
{djupdal,pauline}@idi.ntnu.no
http://crab.idi.ntnu.no

Abstract. Fault Tolerance is an increasing challenge for integrated circuits due to semiconductor technology scaling. This paper looks at how artificial evolution may be tuned to the creation of novel redundancy structures which may be applied to meet this challenge. An experimental setup and results for creating "useful" redundant structures is presented.

1 Introduction

As the semiconductor feature size decreases and the number of transistors on a single chip increases, one of the growing challenges facing the electronic design community is faulty behaviour [1]. This challenge may be met by improved fault tolerance methods. The semiconductor fault challenge may be, in general, a long term challenge but is here today for large ICs, like FPGAs. The mass production of FPGAs enables FPGAs to be produced in the newest technologies. Xilinx Virtex 5 [2] is an example of a new FPGA series from Virtex produced in 65nm technology with up to 330,000 logic cells.

If faults are expected to occur in a digital circuit, fault tolerance — the ability to function correctly in the presence of faults, may be achieved by incorporating redundance (additional resources) in some form. These additional resources may be in the form of additional hardware, in which case it is called *hardware redundancy* [3], the focus of this paper.

To find new redundancy techniques it is important to free oneself from the constraints brought upon us by thinking in the way of traditional redundance techniques. The way one thinks when designing circuits is influenced by the way that one is taught electronics, designed electronics and the tools used in the design process. One way of freeing oneself from these human and design automated constraints is to search for ideas using some sort of heuristic search process. One such process is that of evolutionary algorithms [4].

The application of evolutionary algorithms to the design of hardwares is termed evolvable hardware [5]. The goal being, either to explore for unique solutions or to optimise existing solutions. However, in both cases, the goal is usually to obtain a given behaviour e.g. a binary adder [6]. Further, evolution may be applied when seeking some sort of structure, such as evolving the french flag [7]. In both these cases the goal may be explicitly defined and given to the evolutionary algorithm for comparison between the evolving solutions and the sought solutions. In the former case it is the functionality that needs to be explicitly defined whereas in the latter case it is the structure that needs to be explicitly defined.

In this work, the goal is to push evolution to find useful redundant structures for achieving fault tolerance whilst retaining full functionality. However, these redundant structures are unknown, unlike the case of the earlier mentioned french flag problem. It is not possible to explicitly describe the structure that one is seeking, only the functionality of the sought circuit — perhaps in terms of a truth table.

Section 2 gives an overview of necessary background material. Section 3 presents relevant previous work. The experimental setup is found in section 4 with results and discussion in section 5. The paper concludes in section 6.

2 Background

2.1 Fault Models and Simulated Faults

Two fault models are considered in this work: *the gate reliability model* and the *single fault model*. In the gate reliability model, each gate has a certain probability of failing. A *fault scenario* is one possible configuration of faulty gates for a given circuit. If a fault scenario for the gate reliability model is to be created, each gate in the circuit is tested against a random number generator and selected to be faulty or not based on a chosen gate reliability. This may be said to be a reasonable model of reality as the probability of having failing gates in a circuit is directly proportional to the number of gates in the circuit.

In the single fault model, a circuit can have exactly one fault at any time. If a fault scenario for the single fault model is to be created, one and only one of the gates are selected to fail.

A failing gate can be modelled in several ways. This paper models a failing gate by inverting its output, something that can be said to be a worst-case scenario. Although an inverted output is not a realistic fault for a defect CMOS gate, this fault model is useful for simulation and analysis purposes because it ensures a wrong output for all possible input values.

2.2 Redundancy

A *redundant gate* in a circuit is a gate that may fail without damaging the circuits outputs. To find if a gate in a circuit is redundant or not, a gate redundancy test may be performed where the gate is temporarily made defect. If this does not affect the circuit outputs, the gate is redundant. Finding all redundant gates in a circuit involves applying the redundancy test on all gates one by one.

The ultimate goal of this work is not redundancy, but reliability. Some forms of redundancy are known to enhance a circuits reliability, while other forms of redundancy consist of “dead meat” that does not contribute and should be optimised away from the circuit. In this paper the term *useful redundancy* is used for redundant gates that have a useful purpose in the circuit, while *fake redundancy* is used for gates that have no useful purpose.

2.3 Measuring Functionality and Reliability

The functionality of a circuit is found by trying all possible input values and recording the respective output values of the circuit. If all recorded output values correspond exactly to the desired truth table for the function, the circuit is working perfectly, otherwise 100% functionality is not achieved. Traditionally, the result of such a test for functionality is either "not working" (0) or "working" (1), referred to as f_{bool} herein.

When using artificial evolution to create circuits, f_{bool} is too coarse grained to be used for guiding evolution towards a working circuit. One way of giving evolution more information about how far an individual is from a working solution, is to measure the *hamming distance* between the circuit output and the desired output i.e. the number of bits that are different between these two solutions. This is then normalised to the interval $[0, 1]$ where 1 is 100% working. This measure of functionality is called f_{ham} in this paper.

A reliability metric measures how well a circuit functions in the presence of faults. The traditional reliability metric used in this paper is called R_{trad} and is the average of all f_{bool} results after having tested a number of randomly selected fault scenarios. The possible fault scenarios depend on the fault model chosen. In this paper the traditional reliability metric R_{trad} is used together with the single fault model and is named R_{trad_single}.

A reliability metric may also be based on f_{ham} and is called R_{ehw}. R_{ehw} is the average of all f_{ham} results after having tested a number of randomly selected fault scenarios.

3 Previous Work

In the earlier work of Hartmann and Haddow [8], circuits were evolved with an R_{ehw} based fitness function using the gate reliability model. The results provided clear evidence that evolution traded off functionality for reliability. Instead of making 100% functional circuits and tolerating faults using redundancy, evolution shrunk the circuits. For the gate reliability model, the probability of having a faulty gate in a circuit is directly proportional to the number of gates in the circuit. Evolution took the easiest path to tolerating the faults — it avoided many of them by removing gates to a point where the circuit was no longer 100% functional. While [8] only looked at R_{ehw}, [9] investigated and compared both the R_{ehw} and R_{trad} reliability metrics for evolved and traditional circuits.

In traditional electronics 100% functionality is considered essential. In previous work [10] the problem of evolving 100% functional circuits with redundancy was investigated. Like in this paper, reliability in itself was not the main goal, but rather the creation of redundant structures. To ensure 100% functionality, the fitness function was designed such that f_{ham} was the only contributor to fitness unless functionality was 100%. Thus reliability only affected fitness after 100% functionality was reached.

Several experimental setups were tried, using both the gate reliability model and the single fault model. When using the gate reliability model, no form of

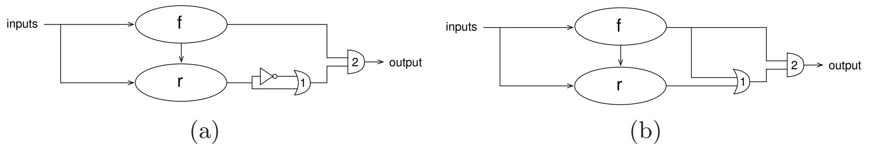

Fig. 1. Structures evolved in [10]

redundancy was achieved as the simplest solution for evolution was to minimise the number of gates used in implementing a fully functional circuit. The single fault model experiments on the other hand created larger circuits containing redundant gates. It was concluded that the single fault model does not discourage large circuits and evolution can therefore more easily introduce new redundant structures.

The first evolved structure in [10] containing redundant gates had the form shown in figure 1(a). The subcircuit marked f implements the desired function and the subcircuit marked r implements any function. All gates in r are redundant. The three gates in the figure makes sure that r does not have any impact on the output at all — the output of gate 1 is constant 1 no matter what r evaluates to. This gate is called *unreachable* because no input vector has any impact on the output of the gate. This structure was evolved using the fitness function $f = k_1 \cdot f_{ham} + k_2 \cdot R_{trad\_single}$ and evolution achieved high fitness by making r as large as possible and f as small as possible, thus scoring high on R_{trad_single}. The redundant gates in r are fake and thus not useful for any purpose. They do not, in any way, influence the output and could just as well be removed.

One way of avoiding the structure in figure 1(a) is to detect unreachable gates. This was also tried in [10]. Any subcircuit with unreachable gates as the only outputs can be excluded when R_{trad_single} is calculated. In this way, such structures do not contribute to fitness, i.e. R_{trad_single} and evolution is encouraged to find another way to improve fitness. The result is typically a structure as in figure 1(b). Here there are no unreachable gates but the redundant gates in r are still just as “useless” for the same reason: r contains only fake redundancy and could just as well be removed from the circuit without affecting functionality or reliability.

The work in [10] managed to create several circuits with redundant gates. However, the method used did not manage to evolve any circuits with useful redundancy. It was concluded that evolution chooses the easiest way to solve the problem, and the easiest way in the experimental setup in [10] was fake redundancy. When the fitness function is not good enough at separating circuits with useful redundancy from circuits with fake redundancy, the result is large amounts of fake redundancy and no useful redundancy. The goal of this paper is to tune the evolutionary process further in order to be able to evolve useful redundancy.

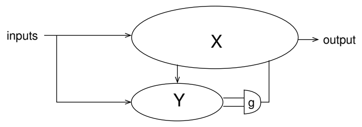

Fig. 2. Circuit partition after selecting any gate g

4 Experiments

This paper builds on the lessons learned in [10]. In [10], a fitness function using R_{trad_single} seemed most promising with regard to introducing redundancy and R_{trad_single} is therefore chosen in this paper.

A key point for improving on the previous experiments is to correctly separate useful redundancy from fake and only include useful redundant gates when R_{trad_single} is to be calculated. Detecting known unwanted structures, like the unreachable gate subcircuits in figure 1(a), is not the answer. Experiments in [10] show that evolution is only going to come up with new ways of cheating by introducing new forms of fake redundancy.

The solution chosen in this paper is to use a more general way of classifying redundancy as useful or fake. Instead of detecting unwanted structures, a gate is simply classified as useful redundant if it has some observable influence on the circuits output. More specifically, a gate is said to be useful redundant if, when the gate becomes defect, some other redundant gate becomes non-redundant in order to maintain correct circuit functionality.

4.1 Algorithm for Classifying Redundant Gates

Algorithm 1, FindFake, is a heuristic for classifying the redundant gates in a given circuit as being either useful redundant or fake redundant. The algorithm works on a given circuit. First, all redundant gates are marked as useful redundant. Then a gate g is selected. For the selected gate g, the circuit can be partitioned into two sets of gates X and Y, both of which may be the empty set, as shown in figure 2. The gates in Y can be disconnected from the circuit by changing the chosen gate g to either Vcc or Gnd, both of which are tried. If this change does not damage the output of the circuit, the number of redundant gates in X after the change is compared to the number of redundant gates in X before the change. If the number of redundant gates in X is unchanged, the gates in Y have no impact on the output and are useless. They are then marked as fake. This is repeated for all the gates in the circuit.

4.2 R_{trad_single} Based on Measured Redundancy

A measure like R_{trad_single} depends on the function the circuit is supposed to perform — R_{trad_single} is 0 when functionality is not 100%. To encourage redundancy early during evolution, before the individuals reach 100% functionality,

Algorithm 1. Classifying redundant gates as useful or fake

```
 1: procedure FindFake(circuit)
 2:     markAllRedundantGatesAsUseful
 3:     for all gates $g$ do
 4:         $partitionCircuit(X, Y, g)$        ▷ Find gate sets X and Y given g
 5:         $redundantInX \leftarrow numberRedundant(X)$
 6:         $g \leftarrow vcc$        ▷ Disconnect Y by substituting g with Vcc
 7:         if outputsUnchanged then        ▷ If circuit is still working
 8:             $redundantVcc \leftarrow numberRedundant(X)$
 9:             if $redundantInX \geq redundantVcc$ then
10:                 markAsFake(Y)
11:             end if
12:         end if
13:         $g \leftarrow gnd$        ▷ Disconnect Y by substituting g with Gnd
14:         if outputsUnchanged then
15:             $redundantGnd \leftarrow numberRedundant(X)$
16:             if $redundantInX \geq redundantGnd$ then
17:                 markAsFake(Y)
18:             end if
19:         end if
20:         restoreCircuit        ▷ Change circuit back to the original
21:     end for
22: end procedure
```

the current behaviour of the individual is measured. The measured behaviour is then used when calculating R_{trad_single} instead of the desired target behaviour, resulting in a score for R_{trad_single} even when 100% functionality is not reached.

4.3 Experimental Setup

All experiments are conducted on simulations of circuits in a digital feed forward circuit simulator. Only Boolean logic is allowed and the following gates are available: AND, OR, NAND, NOR, NOT. Cartesian genetic programming [11] is applied with the following GA parameters:

– Maximum number of gates: 100
– Population size: 20
– Tournament selection with elitism ($g = 3$, $p = 0.7$)
– Crossover rate: 0.2
– Mutation rate: 0.05 (mutation applied at the gate level)

The experiments in this paper use the single fault model. The algorithm explained in section 4.1 classifies redundant gates as either useful or fake and only useful redundant gates are included when R_{trad_single} is calculated. R_{trad_single} is calculated based on the current measured behaviour and not the target behaviour.

Evolving function and redundancy at the same time. For experiments evolving functionality and redundancy at the same time, the following fitness function is used:

$$f_1 = 0.7 \cdot f_{ham} + 0.3 \cdot R_{trad\_single} \tag{1}$$

Three sets of experiments are performed using the fitness function in equation (1), differing in target functionality: Two input AND, two input OR and two input XOR.

Evolving function first, then redundancy. If 100% functionality is required before evolving redundancy, the following fitness function is used:

$$f_2 = 0.7 \cdot f_{ham} + 0.3 \cdot \begin{cases} 0 & \text{if } f_{ham} < 1.0 \\ R_{trad\_single} & \text{if } f_{ham} = 1.0 \end{cases} \tag{2}$$

The fitness function in (2) is used when evolving CIR4, a four input one output function with the truth table "1001011101100110" (bit zero to the right)

Evolving with unspecified function. If the target functionality is not specified but instead evolved together with the circuits, the following fitness function is used:

$$f_3 = R_{trad\_single}. \tag{3}$$

5 Results and Discussion

The results and their discussions are separated into three subsections, based on the complexity and type of target behaviour.

5.1 Simple Functionality

The chosen functionality for the simple experiments is a two-input Boolean function that can be implemented with a single gate circuit. Both AND2 and OR2 have been tried. The reason for evolving these very simple functions is to see what redundancy structures emerge when the function requires little effort to evolve.

Table 1 shows the best individuals after running ten independent experiments for both AND2 and OR2. The best results from these experiments all have the same basic idea behind the introduced redundancy: a voter structure similar to figure 3(a) is introduced just before the output of the circuit. Four independent circuit modules are connected to this voter that all perform the desired function. If three of the four modules work correctly, the voter outputs the correct value. This voter structure is created by the evolutionary algorithm to solve the problem, nothing in the experimental setup predefines a voter as the preferred result.

This design may be compared to the most well known traditional fault tolerance method, Triple Modular Redundancy (TMR), that has three modules and a majority voter. It is interesting to see that evolution in fact finds a voter as the

Table 1. Results, simple functionality. “Type” indicates redundancy type: voter or something else. “Red.” is the number of redundant gates. “Non-red.” is the number of non-redundant gates.

(a) AND2

#	Type	Red.	Non-red.
0	Voter	23	3
1	Voter	32	3
2		33	5
3		37	7
4		50	4
5	Voter	39	3
6	Voter	40	3
7	Voter	38	3
8		35	5
9		23	7

(b) OR2

#	Type	Red.	Non-red.
0		18	7
1	Voter	21	3
2	Voter	38	3
3	Voter	17	3
4	Voter	28	5
5		23	5
6		33	6
7	Voter	29	4
8		33	6
9		29	5

best solution. Of all the possible solutions that evolution could have found it chose something close to the traditional solution. The evolved voter is smaller than the TMR-voter (three gates as opposed to four), but needs more working modules. This is no disadvantage when simulating using the single fault model, in fact a three-gate four-input voter is the best solution in this case. In the more realistic gate reliability model, TMR is better as it requires fewer gates in total and, therefore, has fewer gates that may fail.

It is also clear from table 1 that when evolution has managed to create redundancy, the redundant subcircuits are expanded. This can be explained by the use of the single fault model. It is favourable for fitness to have as many redundant gates as possible because R_{trad_single} is the same as the number of redundant gates divided by the total number of gates.

Analysis of Evolved Voter. Why is the voter structure in figure 3(a) successful at hiding single defects in the modules connected to the voters inputs? The voter can be explained by doing a don’t care (DC) analysis of the circuit.

If one input to an AND-gate is zero, the other input is DC because no matter what it is, the output of the AND-gate is zero. Likewise, if one input to an OR-gate is one, the other input is DC. In addition, an input DC is in most cases propagated to the subcircuit connected to this input, meaning that all gates in the subcircuit have a DC for this specific case. This is not true for all possible circuits, but is true for the voter in figure 3(a).

These simple rules can now be used to explain the voter. All four modules connected to the voter should perform the same function, so every wire in figure 3(a) has the same value. The purpose of the voter is to make sure that any single fault in any of the modules is tolerated. The voter should therefore be designed such that if any three of the four inputs to the voter is correct, the fourth input is DC. To see if the voter fulfils this requirement, one should separately examine the two possible

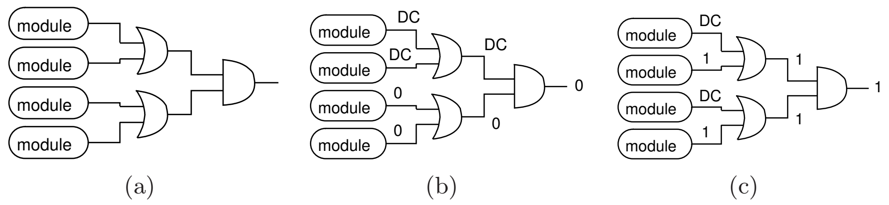

Fig. 3. Evolved voter

cases of voter operation: When the voter output should be zero, and when the voter output should be one.

Zero-case: This case is illustrated in figure 3(b). When the result of the voter should be zero, only one input to the AND-gate of the voter needs to be zero. This means the other input and both modules that are indirectly connected to the input are DC.

One-case: This case is illustrated in figure 3(c). When the result of the voter should be one, both inputs to the AND-gate must also be one. This case must therefore be handled by the OR-gates. For each of the OR-gates to output one, only one of the inputs to each OR-gate needs to be one. This means the other input and the module connected to it are DC.

These two cases show that the voter outputs the correct value even when one of the four modules connected to the voter fails. Note the symmetry in figures 3(b) and 3(c). For example in figure 3(b), it is just as correct to mark the lower two modules having a DC output and the upper two modules having output 0. It can now be seen that if a single module is selected as faulty, if the three other modules work correctly the output will still be correct.

5.2 Complex Functionality

If the functionality of the circuit is more complex it becomes harder to evolve a functional circuit. How does this affect the redundancy structures that are evolved?

XOR2 is a step up in functionality. XOR is not among the gates available for evolution and requires minimum a three gates implemention. In the XOR-case evolution has a much harder time finding a solution as efficient as the voter in figure 3(a). Table 2(a) shows the best individuals after running ten independent experiments for XOR2. The same kind of voter was observed in one of the evolved XOR-circuits, but mostly functionality and the structures used for introducing redundancy in the circuit were mixed together in an intricate way. An example of this is given later in this section.

The most complex functionality evolved in this paper is the four-input CIR4 circuit that requires a nine gate minimum implementation. To ensure 100% functionality, it was necessary to apply the fitness function in equation (2) that forces evolution of functionality first and then redundancy. Table 2(b) shows the best

Table 2. Results, complex functionality. Same layout as in table 1.

(a) XOR2

#	Type	Red.	Non-red.
0		22	6
1		38	8
2		19	10
3		22	6
4		(not working)	
5		32	7
6	Voter	43	3
7		19	10
8		42	11
9		36	7

(b) CIR4

#	Type	Red.	Non-red.
0		17	13
1		24	21
2		24	15
3		21	16
4		11	14
5		24	15
6		25	18
7		15	17
8		18	13
9		32	16

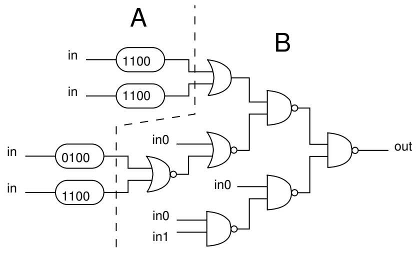

Fig. 4. Redundant XOR2 without voter. IN0 and IN1 are the main circuit inputs

individuals after running ten independent experiments for CIR4. In this case there are no voters evolved and like most of the XOR2 evolutionary runs, functionality and redundancy are mixed together. As can be seen from the number of non-redundant gates in the evolved circuits, the introduced redundancy is not very efficient.

Although not as efficient as the voter solutions, these solutions are still interesting. The purpose of this work is not to evolve the voter but to find new ways of introducing redundancy to a circuit. The solutions in table 2 do represent new redundancy solutions. The inefficiency might come from the fact that the fitness function forces 100% functionality before redundancy. The evolutionary runs were also stopped after a certain amount of time. More efficient redundancy might have been the result if the experiments were allowed to run longer.

Example of Non-Voter Based Redundant Circuit. What does a non-voter based redundant circuit look like? An example of such a circuit is the XOR circuit number nine in table 2(a). This circuit is illustrated in figure 4. The four rounded boxes are subcircuits having the truth table written inside the box (bit zero to the

Table 3. Results, evolving function together with redundancy. Same layout as in table 1, with the addition of column "Function" which is the evolved functionality. IN0 and IN1 are the circuit inputs.

#	Function	Type	Red.	Non-red.
0	IN0	Voter	28	3
1	$\neg$ IN0		39	5
2	AND		23	4
3	$\neg$ IN1	(Voter)	32	3
4	IN1	Voter	59	3
5	IN0	Voter	28	3
6	IN0		49	5
7	IN1	Voter	40	3
8	$\neg$ IN1	Voter	17	3
9	IN1	Voter	31	3

right). All gates in region A (to the left of the dotted line) are redundant while all gates in region B are non-redundant.

The redundant gates in figure 4 are useful redundant, they do have an impact on the circuit output. The XOR functionality is, however, not produced exclusively in the redundant part of the circuit. None of the rounded boxes in the redundant part of the circuit represent XOR. Instead, XOR is formed with a combination of the redundant and non-redundant gates. An analysis similar to the DC analysis for the voter in section 5.1 can be used to understand why the gates in region A are redundant.

5.3 No Specified Functionality

From the previous experiments in this paper it is clear that functionality affects how redundancy is achieved and how effective this redundancy is. As the complexity of the functionality increases, more focus is placed on getting a circuit working and it becomes harder to find an efficient way of creating a redundant version of the circuit.

The evolved redundancy structures are the goal for this paper, not a specific functionality. A set of experiments are performed that does not explicitly state what function the evolved circuits should perform. The only requirement is that the circuit must have two inputs and one output. Evolution is thus free to create any function and focus all efforts on creating circuits with redundancy. This is accomplished by using the fitness function in equation (3). As R_{trad_single} is the only factor in this fitness function and because R_{trad_single} is based on the current measured functionality of an individual, the target functionality of the circuits is evolved together with the redundant circuits themselves. It is likely that the resulting function is something that can easily be made redundant in an efficient way. This is backed up by the results. Table 3 shows the best individuals after running ten independent experiments where the target functionality is not specified. The evolved functions are very simple (typically cloning an input or being the equivalent of a single Boolean gate) and most individuals use a voter similar to figure 3(a).

6 Conclusion and Further Work

This paper has presented an experimental setup that sucessfully uses artificial evolution to create digital circuits with useful redundancy. The purpose of this experimental setup is to find new ways of building redundant circuits.

The results show that although there is no explicit guiding towards creating a voter structure, evolution does in some cases create a voter resembling the voter used in traditionally designed reliable circuits. This is typically the result when evolving circuits with simple functionality. The voter is a known way of making redundant structures and while it is interesting that evolution creates voter like structures, the main goal is to find new ways of introducing redundancy. When evolving more complex functions, the result is non-voter based redundancy. Although not as efficient as a voter based solution, these results are interesting examples on how to do redundancy without the traditional voter.

Planned further work includes experiments where evolution is allowed to leave the strict Boolean logic domain and exploit the analog properties of the CMOS technology.

References

1. ITRS: Int. techn. roadmap for semiconductors. Technical report, ITRS (2005)
2. Xilinx: Xilinx virtex 5 overview. `http://www.xilinx.com/virtex5`
3. Lala, P.K.: Self-Checking and Fault Tolerant Digital Design. Morgan Kaufmann Publishers (2001)
4. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer (2003)
5. Higuchi, T., Niwa, T., Tanaka, T., Iba, H., de Garis, H., Furuya, T.: Evolving hardware with genetic learning: a first step towards building a darwin machine. In: Proc. Int. Conf. From animals to animats: simulation of adaptive behavior. (1993) 417–424
6. Hemmi, H., Mizoguchi, J., Shimohara, K.: Development and evolution of hardware behaviors. In: Artificial Life IV: Proc. 4th Int. Workshop Synthesis Simulation Living Syst., MIT Press (1994) 371–376
7. Miller, J.F.: Evolving a self-repairing, self-regulating, french flag organism. In: Genetic and Evolutionary Computation (GECCO). (2004) 129–139
8. Hartmann, M., Haddow, P.C.: Evolution of fault-tolerant and noise-robust digital designs. IEE Proc. - Computers and Digital Techniques **151**(4) (jul 2004) 287–294
9. Haddow, P.C., Hartmann, M., Djupdal, A.: Addressing the metric challange: Evolved versus traditional fault tolerant circuits. In: Adaptive Hardware and Systems. (2007)
10. Djupdal, A., Haddow, P.C.: Evolving redundant structures for reliable circuits – lessons learned. In: Adaptive Hardware and Systems. (2007)
11. Miller, J.F., Job, D., Vassilev, V.K.: Principles in the evolutionary design of digital circuits Â part i. Journal of Genetic Programming and Evolvable Machines **1**(1) (2000) 8–35