Effective Black-Box Testing with Genetic Algorithms

Mark Last[1]♣, Shay Eyal[1], and Abraham Kandel[2]

[1]Department of Information Systems Engineering, Ben-Gurion University of the Negev,
Beer-Sheva 84105, Israel
{mlast, shayey} @bgu.ac.il
[2]Department of Computer Science and Engineering, University of South Florida
Tampa, FL 33620, USA
kandel@csee.usf.edu

Abstract. *Black-box (functional) test cases* are identified from functional requirements of the tested system, which is viewed as a mathematical function mapping its inputs onto its outputs. While the number of possible black-box tests for any non-trivial program is extremely large, the testers can run only a limited number of test cases under their resource limitations. An *effective* set of test cases is the one that has a high probability of detecting faults presenting in a computer program. In this paper, we introduce a new, computationally intelligent approach to generation of effective test cases based on a novel, Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA). The basic idea is to eliminate "bad" test cases that are unlikely to expose any error, while increasing the number of "good" test cases that have a high probability of producing an erroneous output. The promising performance of the FAexGA-based approach is demonstrated on testing a complex Boolean expression.

Keywords. *Black-box testing, Test prioritization, Computational intelligence, Genetic algorithms, Fuzzy logic*.

1 Introduction and Motivation

Software quality remains a major problem for the software industry worldwide. Thus, a recent study by the National Institute of Standards & Technology [3] found that "the national annual costs of an inadequate infrastructure for software testing is estimated to range **from $22.2 to $59.5 billion**" (p. ES-3) which are about 0.6 percent of the US gross domestic product. This number does not include costs associated with catastrophic failures of mission-critical software systems such as the Patriot Missile Defense System malfunction in 1991 and the $165 million Mars Polar Lander shutdown in 1999.

A system *fails* when it does not meet its specification [17]. The purpose of testing a system is to discover faults that cause the system to fail rather than proving the code correctness, which is often an impossible task [3]. In the software testing process, each *test case* has an identity and is associated with a set of inputs and a list of expected outputs [10]. *Functional (black-box) test cases* are based solely on

♣ Corresponding author

functional requirements of the tested system, while *structural (white-box) tests* are based on the code itself. According to [10], black-box tests have the following two distinct advantages: they are independent of the software implementation and they can be developed in parallel with the implementation.

The number of combinatorial black-box tests for any non-trivial program is extremely large, since it is proportional to the number of possible combinations of all input values. On the other hand, testing resources are always limited, which means that the testers have to choose the tests carefully from the following two perspectives:

1. Generate *good* test cases. A *good* test case is one that has a high probability of detecting an as-yet undiscovered error [19]. Moreover, several test cases causing the same bug may show a pattern that might lead the programmer to the real cause of the bug [16].
2. Prioritize test cases according to a *rate of fault detection* – a measure of how quickly those test cases detect faults during the testing process [5].

As indicated by Jorgensen [10], the above requirements of test effectiveness suffer from a circularity: they presume we know all the faults in a program, which implies that we do not need the test cases at all! Since in reality testers do not know in advance the number and the location of bugs in the tested code, the practical approach to generation of black-box tests is to look for bugs where they *are expected* to be found, while trying to avoid redundant tests that test the same functionality more than once. Thus, the best-known black-box testing techniques include *boundary value testing* (presumes that errors tend to occur near the extreme values), *equivalence class testing* (requires to test only one element from each class), and *decision table-based testing* (based on a complete logical specification of functional requirements). Since all the systematic approaches to test generation cover only a tiny portion of possible software inputs, *random tests*, sometimes called "monkey tests", can discover new bugs in the tested system [16]. Even if a "monkey test" is smart enough to examine the results of each test case and evaluate its correctness, this is an extremely inefficient approach, since many random tests may be completely redundant, testing the same functionality and the same code over and over again. On the other hand, as shown in [10], it may take a huge number of random tests to test certain functionality of the software at least once.

The testers are interested to rank their test cases so that those with the highest priority, according to some criteria, are executed earlier than those with lower priority. This criterion may change according to some performance goal (e.g. code coverage, reliability confidence, specific code changes, etc.). The dominant criterion is the *rate of fault detection* as described earlier. An increased rate of fault detection can provide earlier feedback on the tested software and earlier evidence that quality goals have not been met yet [5]. Such information is particularly useful in *version-specific test case prioritization* while using *regression testing*. In the case of regression testing, the information gathered in previous runs of existing test cases might help to prioritize the test cases for subsequent runs. Obviously, such information is unavailable during initial testing.

Genetic Algorithms (GA) provide a general-purpose search methodology, which uses principles of the natural evolution [6]. In this paper, we introduce a new, computationally intelligent approach to improving the effectiveness of a given test set by eliminating "bad" test cases that are unlikely to expose any error, while increasing

the number of "good" test cases that have a high probability of producing an erroneous output. The algorithm is started by randomly generating an initial test set, where each test case is modeled as a *chromosome* representing one possible set (vector) of input values. To apply a genetic algorithm to the test generation problem, the evaluation function should emulate the capability of a given test case to detect an error in a tested program. "Good" chromosomes (test cases) that discover the errors are given a positive reward, while a zero score is given to those that do not. This way the algorithm *prioritizes* a given set of test cases. At each iteration, the "population" of test cases is updated by mutating and recombining its most prospective chromosomes. Consequently, the percentage of fault-exposing test cases is expected to increase from one generation to the next. The performance of the basic genetic algorithm is enhanced by the novel Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA) introduced by us in [12].

This paper is organized as follows. Section 2 presents an overview of the genetic algorithms methodology and its application to the test generation problem. The design of initial testing experiments and the obtained results are described in Sections 3 and 4 respectively. Section 5 concludes the paper.

2 Software Testing with Genetic Algorithms

2.1 Genetic Algorithms: An Overview

Genetic Algorithms (GAs) are general-purpose search algorithms, which use principles inspired by natural genetics to evolve solutions to problems. As one can guess, genetic algorithms are inspired by Darwin's theory about evolution [9]. They have been successfully applied to a large number of scientific and engineering problems, such as optimization, machine learning, automatic programming, transportation problems, adaptive control, etc. (see [13][14]).

GA starts off with population of randomly generated chromosomes, each representing a candidate solution to the concrete problem being solved, and advances towards better chromosomes by applying genetic operators based on the genetic processes occurring in nature. So far, GAs have had a great measure of success in search and optimization problems due to their robust ability to exploit the information accumulated about an initially unknown search space. Particularly GAs specialize in large, complex and poorly understood search spaces where classic tools are inappropriate, inefficient or time consuming [8].

As mentioned, the GA's basic idea is to maintain a population of chromosomes. This population evolves over time through a successive iteration process of competition and controlled variation. Each state of population is called generation. Associated with each chromosome at every generation is a *fitness* value, which indicates the quality of the solution, represented by the chromosome values. Based upon these fitness values, the selection of the chromosomes, which form the new generation, takes place. Like in nature, the new chromosomes are created using genetic operators such as *crossover* and *mutation*.

The fundamental mechanism consists of the following stages (as described in Figure 1):

1. Generate randomly the initial population.
2. Select the chromosomes with the best fitness values.
3. Recombine selected chromosomes using crossover and mutation operators.
4. Insert offsprings into the population.
5. If a stop criterion is satisfied, return the chromosome(s) with the best fitness. Otherwise, go to Step 2.

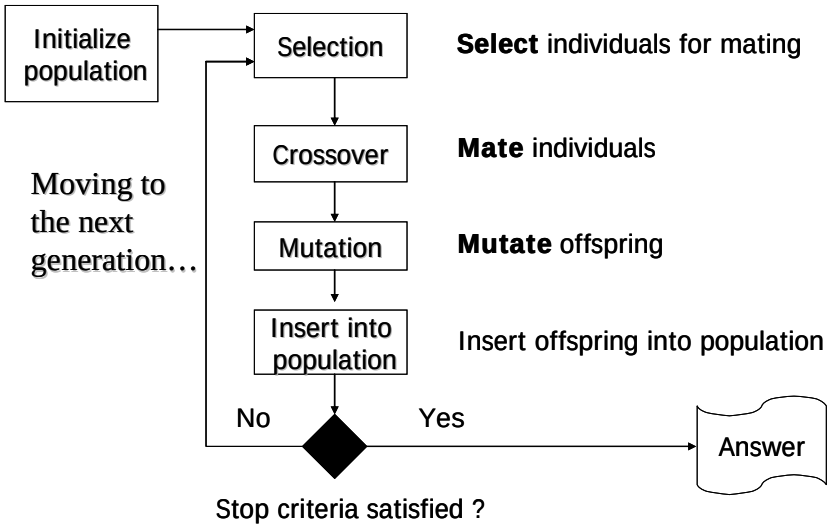

Fig. 1. Simple Genetic Algorithm Fundamental Mechanism

To apply genetic algorithm to a particular problem, such as test case generation, we need to determine the following elements [13]:

1. Genetic representation for potential solutions to the problem (e.g., test cases).
2. Method to create an initial population of potential solutions.
3. Evaluation function, which scores the solution quality (also called objective function).
4. Genetic operators that alter the composition of the off-springs.
5. Values of various parameters used by the genetic algorithm (population size, probabilities of applying genetic operators, etc.).

Each element is briefly discussed below.

Representation refers to the modeling of chromosomes into data structures. Once again terminology is inspired by the biological terms, though the entities genetic algorithm refers to are much simpler than the real biological ones. *Chromosome* typically refers to a candidate data solution to a problem, often encoded as a bit string. Each element of the chromosome is called *allele*. In other words, chromosome is a sequence of alleles. For example, consider 1-dimension binary representation: each allele is 0 or 1, and a chromosome is a specific sequence of 0 and 1's. According to

[14], a *proper representation* for a given problem should cover the whole search space, avoid infeasible solutions and redundant data, and require a minimal computational effort for evaluating the objective function.

Initialization. This genetic operator creates an initial population of chromosomes, at the beginning of the genetic algorithm execution. Usually initialization is random, though it may be biased with a pre-defined initialization function, as suggested in [18].

The *selection* operator is used to choose chromosomes from a population for mating. This mechanism defines how these chromosomes will be selected, and how many offsprings each will create. The expectation is that, like in the natural process, chromosomes with higher fitness will produce better offsprings. Therefore, selecting such chromosomes at higher probability will eventually produce better population at each iteration of the algorithm. Selection has to be balanced: too strong selection means that best chromosomes will take over the population reducing its diversity needed for exploration; too weak selection will result in a slow evolution. Classic selection methods are *Roulette-Wheel*, *Rank based*, *Tournament*, *Uniform*, and *Elitism* [6][9].

The *crossover* operator is practically a method for sharing information between two chromosomes: it defines the procedure for generating an offspring from two parents. The crossover operator is considered the most important feature in the GA, especially where building blocks (i.e. schemas) exchange is necessary [1]. The crossover is data type specific, meaning that its implementation is strongly related to the chosen representation. It is also problem-dependent, since it should create only feasible offsprings. One of the most common crossover operators is *Single-point crossover*: a single crossover position is chosen at random and the elements of the two parents before and after the crossover position are exchanged to form two offsprings.

The *mutation* operator alters one or more values of the allele in the chromosome in order to increase the structural variability. This operator is the major instrument of the genetic algorithm to protect the population against pre-mature convergence to any particular area of the entire search space [14]. Unlike crossover, mutation works with only one chromosome at a time. In most cases, mutation takes place right after the crossover, so it practically works on the offsprings, which resembles the natural process. The most common mutation methods are:

1. Bit-flip mutation: given chromosome, every bit value changes with a mutation probability.
2. Uniform mutation: choose one bit randomly and change its value.

Evaluation function, also called objective function, rates the candidate solutions quality. This is the only single measure of how good a single chromosome is compared to the rest of the population. So given a specific chromosome, the evaluation function returns its score. The score value is not necessary the fitness value, which the genetic algorithm works with, as mentioned earlier. The fitness value is typically obtained by a transformation function called *scaling*. It is important to note that the evaluation process itself has been found to be very ‘expensive’ due to the time and resources it consumes [1]. Therefore it is worthwhile to simplify the evaluation function as much as possible.

The basic genetic algorithm has a set of parameters, which define its operation and behavior. Within the basic genetic algorithm, the parameter settings are fixed along the run [9]. The list of common GA parameters is given below:

1. *Population size (N)* – this parameter defines the size of the population, which may be critical in many applications: If *N* is too small, GA may converge quickly, whereas if it is too large the GA may waste computational resources. Regarding evaluation, if the number of fitness evaluations is not a concern, larger population sizes improve the GA's ability to solve complex problems [1]. However, in natural environments the population size changes according to growth rate, and tends to stabilize around an appropriate value [2].
2. *Chromosome length (L)* – defines the number of allele within each chromosome. This number is influenced by the chosen representation and the problem being issued.
3. *Number of generations (Ngen)* – defines the number of generations the algorithm will run. It is frequently used as a stopping criterion.
4. *Crossover probability (Pc)* – the probability of crossover between two parents. The crossover probability has another trade-off: if *Pc* is too low, then the sharing of information between high fitness chromosomes will not take place, hence reducing their capability to produce better offsprings. On the other hand, the crossover may also bring an offspring with lower fitness, and therefore if the *Pc* value is too high there is a chance we may get trapped in a local optimum.
5. *Mutation probability (Pm)* – the probability of mutation in a given chromosome. As mentioned earlier, this method element helps to prevent the population from falling into local extremes, but a too high value of *Pm* will slow down the convergence of the algorithm.

Finding robust methods for determining the universally best GA parameter settings is probably impossible, since the optimal values are problem-dependent and the GA parameters interact with each other in a complex way [2]. Furthermore, to attain an optimal exploitation/exploration balance, the parameter values may need adjustment in the course of running the algorithm. The GAVaPS is a Genetic Algorithm with Varying Population Size where each individual has a lifetime, which is measured in generations [1]. An individual remains in the population during its lifetime. Since parents are selected randomly, the selective pressure is assured by the fact that genomes representing better solutions have longer lifetimes. In [12], we have introduced a fuzzy-based extension of the GAVaPS algorithm, called FAexGA, which has outperformed the simple GA and the GAVaPS on a set of benchmark problems. The FAexGA algorithm is briefly described in the next sub-section.

2.2 Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA)

Our general principle is that crossover probability *Pc* should vary according to age intervals during the lifetime: for both young and old individuals the crossover probability is naturally low, while there is a certain age interval, where this probability is high. The concepts of "young", old", and "middle-aged" are modeled as

linguistic variables using the concepts of fuzzy set theory [11]. Therefore, very young offsprings would be less crossovered thus enabling exploration capability. On the other hand, old offsprings being less crossovered and eventually dying out would help avoiding a local optimum (premature convergence). The middle-age offsprings being crossovered most frequently would enable the exploitation aspect. As shown in [12], this approach is able to enhance the exploration and exploitation capabilities of the algorithm, while reducing its rate of premature convergence.

The general structure of the FAexGA algorithm is described in [12]. It is similar to the GAVaPS [1], with the following difference: in GAVaPS, the recombination of two parents takes place by crossover with a fixed crossover probability; in the fuzzy-based age extension of GA, the crossover probability *Pc* is determined by a Fuzzy Logic Controller (FLC). The FLC state variables include the age and the lifetime of the chromosomes to be crossovered (parents) along with the average lifetime of the current population.

The *fuzzification interface* of the Fuzzy Logic Controller defines for each parent the truth values of being {*Young*, *Middle-age*, *Old*}, which are the linguistic labels of the *Age* inputs. These values determine the grade of membership for each rule premise in the FLC rule base. This computation takes into account the age of only one parent at a time, and relies on the triangular membership functions like the ones shown in Fig. 2. The relative age mentioned on the x-axis in Fig. 2 refers to the ratio between chromosome's age and the average lifetime in the current population. Inspired by nature, the concept of person being *Young*, *Middle-age* or *Old* is relative to the average lifetime in the population; for example, if the average lifetime of a human population is about 77 years (76 for male and 78 for female), then a person whose age is between 35-55 is still considered as middle-age.

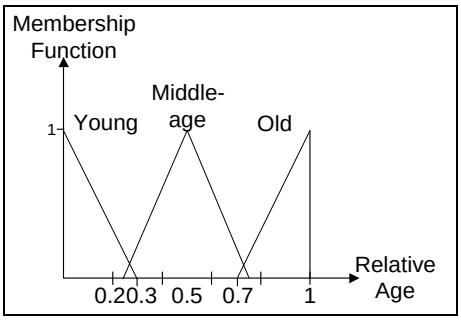

Fig. 2. Fuzzy Logic Controller

In our software testing case study (see the next section), we have experimented with three different settings of *Young upper limit* and *Old lower limit* parameters. These three configurations, denoted as **FAexGA#1, FAexGA#2**, and **FAexGA#3** respectively, are described in Table 1.

Table 1. Fuzzy-Based Age Extension of GA Tested Configurations

Parameter	FAexGA#1	FAexGA#2	FAexGA#3
FLC – "Young" upper limit	30%	25%	20%
FLC – "Old" lower limit	70%	75%	80%

The fuzzy rule base we used in our experiments is presented in Table 2. Each cell defines a single fuzzy rule. For example, the upper left cell refers to the following rule: "**If** Parent I is *Young* and Parent II is *Young* **Then** crossover probability is *Low*".

Table 2. Fuzzy Rule Base

Parent I / Parent II	**"Young"**	**"Middle-age"**	**"Old"**
"Young"	Low	Medium	Low
"Middle-age"	Medium	High	Medium
"Old"	Low	Medium	Low

The inference method used is MAX-MIN (see [11]): each rule is assigned a value equal to the minimum value of its conditions. Afterwards, the inference engine assigns each linguistic output a single value, which is the maximum value of its relative fired rules Thus, by the end of the inference process, the crossover probability will have degrees of truth for being *Low*, *Middle* and *High*.

The *Defuzzification interface* computes a crisp value for the controlled parameter *crossover probability* based on the values of its linguistics labels {*Low*, *Medium*, *High*}, which are the outputs of the rule base, and the triangular membership functions shown in Fig. 3. The settings of the linguistic labels were chosen according to previous studies. The defuzzification method used is *center of gravity (COG)* [11]: the crisp value of the output variable is computed by finding the center of gravity under the membership function for the fuzzy variable.

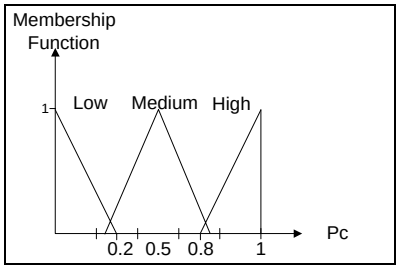

Fig. 3. The *Crossover Probability* Linguistic Variable

2.3 Generating Effective Test Cases with Genetic Algorithms

The problem of test set generation is a typical optimization problem in a combinatorial search space: we are looking for a minimal set of test cases that are most likely to reveal faults presenting in a given program. A genetic algorithm can be applied to the problem of generating an effective set of test cases as follows:

- *Representation*. Each test case can be represented as a vector of binary or continuous values related to the inputs of the tested software.
- *Initialization*. An initial test set can be generated randomly in the space of possible input values.
- *Genetic operators*. Selection, crossover, and mutation operators can be adapted to representation of test cases for a specific program.
- *Evaluation function*. Following the approach of [5], test cases can be evaluated by their *fault-exposing-potential* using buggy (mutated) versions of the original program.

3 Design of Experiments

Our case study is aimed at generating an effective set of test cases for a Boolean expression composed of 100 Boolean attributes and three logical operators: AND, OR, and NOT (called *correct expression*). This expression was generated randomly by using an external application. To define a simple evaluation function for each test case, we have generated an *erroneous expression* – the same Boolean expression as the correct one, except for a single error injected in the correct expression by randomly selecting an OR operator and switching it to the AND one (or vice versa). Such errors are extremely hard to detect and backtrack in complex software programs, since they usually affect the outputs of only a small portion of combinatorial test cases (2^{100} test cases for the expression we have defined).

The chromosomes are 1-dimesional binary strings of 100-bit length. The value of the evaluation function *F* is calculated as follows:

$$F(T) = \begin{cases} 1, if Eval\_Correct(T) \neq Eval\_Erroneous(T) \\ 0, respectively \end{cases}$$

Where *T* is a 100-bit 1-dimesional binary chromosome representing a single test case and *Eval_Correct (T)* / *Eval_Erroneous (T)* are the binary results of applying chromosome T to the correct / erroneous expression respectively.

The experimental GAs' settings are stated in Table 3.

Table **3.** Experimental Settings

#	Parameter	Fuzzy-based Age Extension of GA	GAVaPS	SimpleGA
	N = Population size	100	100	100
	L = chromosome length	100	100	100
	Ngen = total number of generations	200	200	200
	Pc = crossover probability	Adaptive	0.9	0.9
	Pm = mutation probability	0.01	0.01	0.01
	Selection method	Random	Random	Roulette Wheel
	Crossover method	1-point	1-point	1-point
	Mutation method	Flip	Flip	Flip
	1-elitism	-	-	True
	Lifetime calculation strategy	Bi-Linear	-	-
	Reproduction ratio	0.3	0.3	-
	MinLT = Minimum lifetime (number of generations)	1	1	-
	MaxLT = Maximum lifetime (number of generations)	7	7	-

4 Summary of Results

Table 4 presents a summarized comparison for each performance measure obtained with each algorithm. The results of each algorithm were averaged over 20 runs.

Table 4. Comparative Evaluation of Genetic Algorithms

Measure\Algorithm	SimpleGA	GAVaPS	FAexGA#1	FAexGA#2	FAexGA#3
% Runs with solution	70%	60%	95%	70%	75%
% Solutions in the final population	1.267%	81.519%	99.934%	98.857%	87.183%
Unuse Factor	65.571%	62.269%	70.658%	68.714%	46.267%

The first measure (% Runs with solution) shows that the first configuration of the Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA#1) has reached a solution in a significantly higher percentage of runs compared to all other algorithms. Solution means a test case that detects the injected error by returning different values when applied to the correct expression and the erroneous one. Surprisingly, GAVaPS reached a solution in fewer runs than the SimpleGA. The rest of the measures demonstrate a clear advantage of the FAexGA configurations: the number of solutions found in the SimpleGA final population is very small (about 1-2 solutions). GAVaPS found much more solutions than the SimpleGA while all FAexGA configurations found even more. FAexGA#1 is the most *effective* configuration, since it found the highest number of distinct solutions in its final population.

Table 4 also shows that all genetic algorithms prove to be considerably more *efficient* for this problem than the conventional test generation methods. According to [10], a program with logical variables, like a Boolean expression, should be tested by

the *strong equivalence class* approach. Since each binary input in the tested expression has two equivalence classes (0 and 1), this would mean enumeration of $2^{100} \approx 1.27 \cdot 10^{30}$ distinct tests, which is a completely impossible task. Another alternative for testing such an expression is *random testing*, but the probability of one random test case to reveal a single operator change in a 100-term Boolean expression may be as low as $0.5^{99} \approx 1.58 \cdot 10^{-30}$, which is practically zero.

In addition, the FAexGA#1 unuse factor is the highest one. Unuse Factor was measured as $u = 1 - g / g_{max}$ where g is the number of generations until the first solution (i.e., test case that finds an error) appeared and $g_{\max}$ is the maximum number of generations [2]. The SimpleGA unuse factor is better than GAVaPS and the third configuration of the Fuzzy-Based Age Extension of Genetic Algorithms. Overall, the first configuration of the Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA#1) has reached the best results with respect to all three performance measures.

Figure 4 shows the average percentage of solutions in each generation. As mentioned earlier, the SimpleGA found a very small number of solutions, while the first and the second configuration of FAexGA found solutions at quite a linear rate.

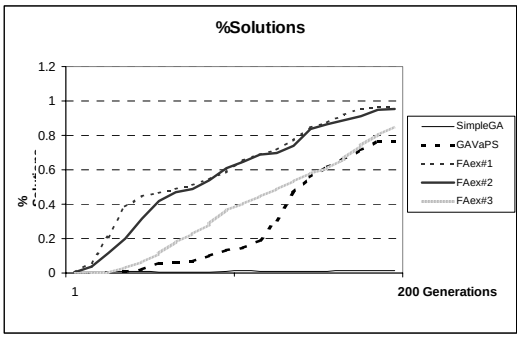

Fig. 4. Average Percentage of Solutions in the Population

Figure 5 shows the population average genotypical diversity in runs where solution was found. It is calculated by comparing the inner structure of the chromosomes [7]. The SimpleGA diversity as well as the GAVaPS diversity remains high most of the generations because the number of solutions found is relatively low. The Fuzzy-Based Age Extension of GA diversities are significantly lower (especially #1 and #2) and get even lower in the final generations due to the fact that a high percentage of the population are good solutions.

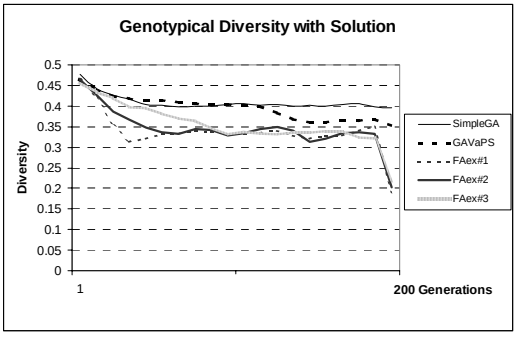

Fig. 5. Genotypical Diversity of Each Algorithm Solutions

Figure 6 describes the population average genotypic diversity in runs where solution was *not* found. All algorithms diversity is relatively high due to the fact that exploration occurs, but without a success.

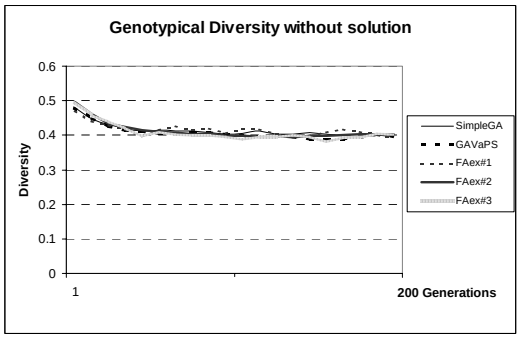

Fig. 6. Genotypical Diversity of Each Algorithm Non-Solutions

Figures 7-11 illustrate for each algorithm the differences between the diversities in runs where solutions were found to those where solutions were not found (*G. Average* stands for runs with solutions and *B. Average* for runs without solutions). Since SimpleGA and GAVaPS found a smaller number of solutions, there are no differences between their diversities. On the contrary, all three configurations of the Fuzzy-Based Age Extension of Genetic Algorithms present significant differences between runs where solutions were found to those where solutions were not found. Therefore, it can be concluded that FAexGA maintains the diversity in a suitable manner: it enables exploitation while still retaining some level of diversity for potential exploration.

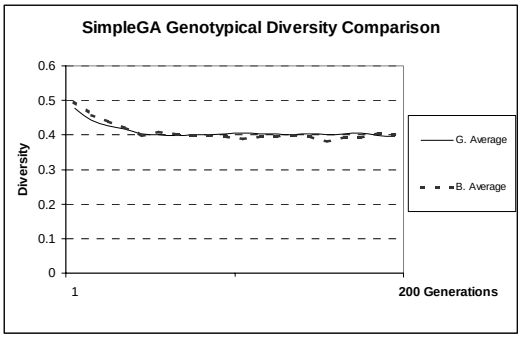

Fig. 7. : SimpleGA Genotype Diversities

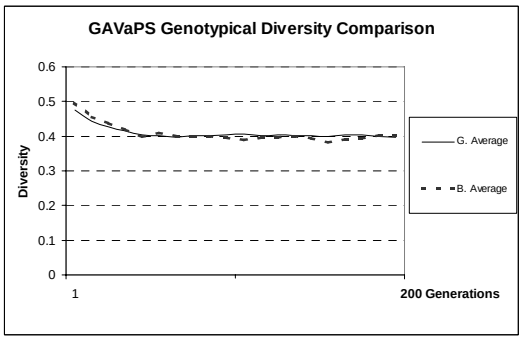

Fig. 8. GAVaPS Genotype Diversities

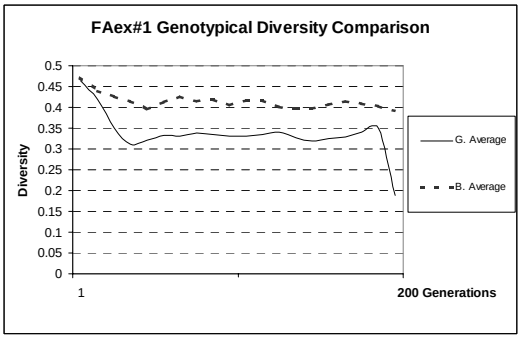

Fig. 9. FAexGA#1 Genotype Diversities

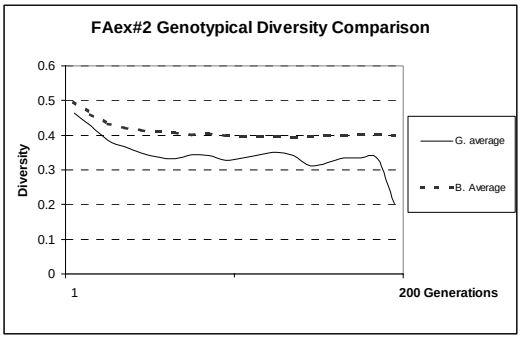

Fig. 10. FAexGA#2 Genotype Diversities

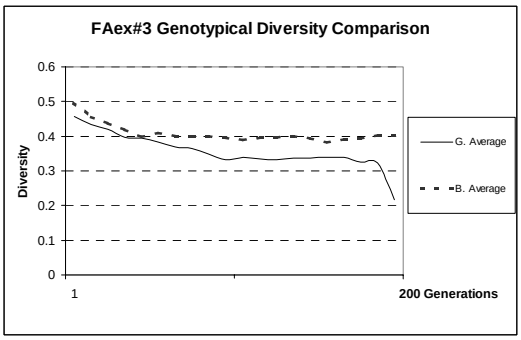

Fig. 11. FAexGA#3 Genotype Diversities

5 Conclusions

In this paper, we have introduced a new, GA-based approach to generation of effective black-box test cases. From the case study, we can conclude that the Fuzzy-Based Age Extension of Genetic Algorithm (FAexGA) is much more efficient for this problem than the two other evaluated algorithms (SimpleGA and GAVaPS). First, FAexGA has a much higher probability to find an error in the tested software. Second, the error would be found much faster, which should result in saving a lot of resources for the testing team. In addition, the number of distinct solutions produced by FAexGA is significantly higher, which may be useful for investigation and identification of the error itself by the software programmers. Future research includes application of the proposed methodology to testing of real programs and development of more sophisticated, possibly continuous evaluation functions for the evolved test cases.

Acknowledgments. This work was partially supported by the National Institute for Systems Test and Productivity at University of South Florida under the USA Space and Naval Warfare Systems Command Grant No. N00039-01-1-2248 and by the

Fulbright Foundation that has awarded Prof. Kandel the Fulbright Senior Specialists Grant at Ben-Gurion University of the Negev in November-December 2005.

References

[1] Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS – a Genetic Algorithm with varying population size, In Proc. of the first IEEE Conference on Evolutionary Computation, (1994) 73-78.
[2] Deb, K., Agrawal, S.: Understanding interactions among genetic algorithm parameters, In: W. Banzhaf, C. Reeves (eds.), Foundations of Genetic Algorithm 5, Morgan Kaufmann, San Francisco, CA, (1998) 265-286.
[3] DeMillo, R.A. & Offlut, A.J.: Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering 17, 9 (1991) 900-910.
[4] Eibon, A. E., Hinterding, R., Michalewicz, Z.: Parameter Control in Evolutionary Algorithm, IEEE Transactions on Evolutionary Computation, (1999) 124-141.
[5] Elbaum, S., Malishevsky, A. G., Rothermel, G.: Prioritizing Test Cases for Regression Testing, in Proc. of ISSTA '00 (2000). 102-112.
[6] Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA, 1989.
[7] Herrera F., and Lozano, M.: Adaptation of genetic algorithm parameters based on fuzzy logic controllers. In Herrera F. and Verdegay J. (eds) Genetic Algorithms and Soft Computing, Physica Verlag, (1996) 95-125.
[8] Herrera F., and Magdalena, L.: Genetic Fuzzy Systems: A Tutorial. Tatra Mt. Math. Publ. (Slovakia), 13, (1997) 93-121
[9] Holland, J. H.: Genetic Algorithms, Scientific American, 267(1) (1992) 44-150.
[10] Jorgensen, P. C.: Software Testing: A Craftsman's Approach. Second Edition, CRC Press, 2002.
[11] Klir G. J., and Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications, Prentice-Hall Inc., 1995.
[12] Last M., and Eyal, S.: A Fuzzy-Based Lifetime Extension of Genetic Algorithms, Fuzzy Sets and Systems, Vol. 149, Issue 1, January 2005, 131-147.
[13] Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs, Verlag, Heidelberg, Berlin, Third Revised and Extended Edition, 1999.
[14] Mitchell, M.: An Introduction to Genetic Algorithms, MIT Press, 1996.
[15] National Institute of Standards & Technology. “The Economic Impacts of Inadequate Infrastructure for Software Testing”. Planning Report 02-3 (May 2002).
[16] Patton, R.: Software Testing, SAMS, 2000.
[17] Pfleeger, S. L.: Software Engineering: Theory and Practice. 2nd Edition, Prentice-Hall, 2001.
[18] Reeves, C. R.: Using Genetic Algorithms with Small Populations, Proc. Of the Fifth Int. Conf. On Genetic Algorithms and Their Applications, Morgan Kaufmann, (1993) 92-99.
[19] Schroeder P. J., and Korel, B.: Black-Box Test Reduction Using Input-Output Analysis. In Proc. of ISSTA '00 (2000). 173-177.