A Genetic Algorithm Approach to Focused Software Usage Testing

Robert M. Patton, Annie S. Wu, and Gwendolyn H. Walton
University of Central Florida
School of Electrical Engineering and Computer Science
Orlando, FL, U.S.A.

ABSTRACT

Because software system testing typically consists of only a very small sample from the set of possible scenarios of system use, it can be difficult or impossible to generalize the test results from a limited amount of testing based on high-level usage models. It can also be very difficult to determine the nature and location of the errors that caused any failures experienced during system testing (and therefore very difficult for the developers to find and fix these errors). To address these issues, this paper presents a Genetic Algorithm (GA) approach to focused software usage testing. Based on the results of macro-level software system testing, a GA is used to select additional test cases to focus on the behavior around the initial test cases to assist in identifying and characterizing the types of test cases that induce system failures (if any) and the types of test cases that do not induce system failures. Whether or not any failures are experienced, this GA approach supports increased test automation and provides increased evidence to support reasoning about the overall quality of the software. When failures are experienced, the approach can improve the efficiency of debugging activities by providing information about similar, but different, test cases that reveal faults in the software and about the input values that triggered the faults to induce failures.

KEYWORDS:

Genetic algorithms, software usage testing, simulation testing, debugging, system testing, black box testing

1. OVERVIEW

This work focuses on system level, model-based usage testing. The software to be tested is viewed from the perspective of the user as a black box system that operates in a specific environment, receives input, and provides output. One or more state-based models of software use are developed, using domain-specific knowledge to characterize the population of uses of the software (or usage scenarios) and to describe test management

objectives and constraints. The usage models are used to assist with test planning, to generate a sample of test cases that represent usage scenarios, and to support reasoning about test results.

System-level usage testing approaches have proven to be successful for supporting test case selection and reasoning about test results in a variety of software projects. However, the system testing typically consists of only a very small sample from the set of possible scenarios of system use. Thus, it can be difficult or impossible to generalize the test results from a limited amount of testing based on high-level usage models. It can also be very difficult to determine the nature and location of the errors that caused any failures experienced during system testing (and therefore very difficult for the developers to find and fix these errors).

This paper presents a Genetic Algorithm (GA) approach to addresses these issues. As illustrated in Figure 1 and described in detail in section 5, the GA accepts input from two sources: (a) domain data generated by the usage model to define a usage scenarios and (b) the results (pass/fail) of system test. The initial population is defined as a set of test cases generated from a usage model. Each *individual* in the population represents a *single* test case. The individual is sent to the Tester to be processed and supplied to the Software Under Test. The Software Under Test processes this input and provides output that is analyzed for correctness by the Test Oracle. The Test Oracle will determine if the output is correct or flawed or if the software under test crashed. The Test Oracle informs the GA of the result: output is correct, output is flawed, or Software Under Test crashed. The GA uses this result along with the likelihood that it would occur as defined by the usage model to help determine the overall fitness of the individual. The GA outputs individual test cases that caused high intensity failures within the high usage areas of the software, thus driving dynamic testing and system analyses in a focused manner based on test objectives (as described by the usage model) and previous test results.

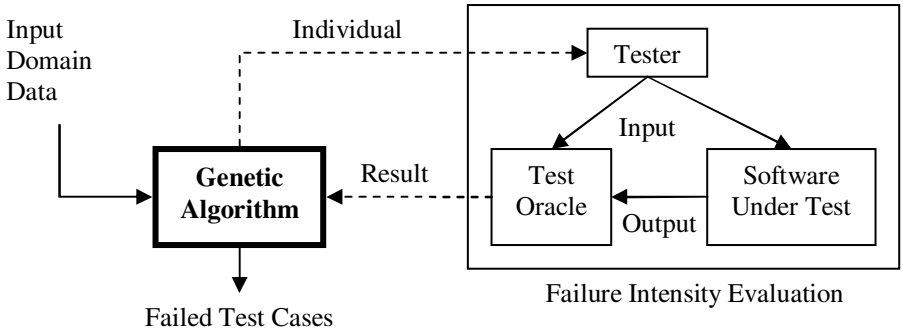

Figure 1. GA Approach to Focused Software Usage Testing

The remainder of this paper is organized as follows. Section 2 provides a high-level introduction to Genetic Algorithms and pointers to related work. Section 3 provides some background information about system testing and debugging activities and challenges that motivate the GA approach presented in this paper. Section 4 introduces the GA approach to focused usage testing, and section 5 provides information about the internal details of the GA. Section 6 provides an example to illustrate application of this approach to drive focused testing of a military simulation system. Conclusions are presented in section 7.

2. INTRODUCTION TO GENETIC ALGORITHMS

A genetic algorithm (GA) is a search algorithm based on principles from natural selection and genetic reproduction [Holland 1975; Goldberg 1989]. GAs have been successfully applied to a wide range of applications, [Haupt 1998; Karr 1999; Chambers 2000] including optimization, scheduling, and design problems. Key features that distinguish GAs from other search methods include:

- A *population of individuals* where each individual represents a potential solution to the problem to be solved.
- A *fitness function* which evaluates the utility of each individual as a solution.
- A *selection function* which selects individuals for reproduction based on their fitness.
- Idealized *genetic operators* which alter selected individuals to create new individuals for further testing. These operators, e.g. crossover and mutation, attempt to explore the search space without completely losing information (partial solutions) that is already found.

Figure 2 provides the basic steps of a GA. First the population is initialized, either randomly or with user-defined individuals. The GA then iterates thru an evaluate-select-reproduce cycle until either a user defined stopping condition is satisfied or the maximum number of allowed generations is exceeded.

```
procedure GA
{
    initialize population;
    while termination condition not satisfied do
    {
        evaluate current population;
        select parents;
        apply genetic operators to parents to create children;
        set current population equal to be the new child population;
    }
}
```

Figure 2. Basic steps of a typical genetic algorithm

The use of a population allows the GA to perform parallel searches into multiple regions of the solution space. Operators such as crossover [Holland 1975; Goldberg 1989; Mitchell 1996] allow the GA to combine discovered partial solutions into more complete solutions. As a result, the GA is expected to search for small building blocks in parallel, and then iteratively recombine small building blocks to form larger and larger building blocks. In the process, the GA attempts to maintain a balance between exploration for new information and exploitation of existing information. Over time, the GA is able to evolve populations containing more fit individuals or better solutions. For more information about GAs, the reader is referred to [Holland 1975; Goldberg 1989; Mitchell 1996; Coley 2001].

While, the GA approach presented in this paper is unlike other published approaches to the application of GA to support software testing or software quality assessment, the "failure-pursuit sampling" work of [Dickinson et al. 2001] and the "adaptive testing" work of [Schultz et al. 1992] are particularly noteworthy with respect to their motivation for the work of this paper.

While [Dickinson et al. 2001] does not explicitly make use of a GA, their concept of failure-pursuit sampling helped to provide a foundation for the approach presented in this paper. In failure-pursuit sampling, some initial sample of test cases is selected; the sample is evaluated and failures recorded; and additional samples are then selected that are in the vicinity of failures that occurred in the previous sample.

[Schultz et al. 1992] demonstrated the use of adaptive testing to test intelligent controllers for autonomous vehicles by creating individuals in the population that represented fault scenarios to be supplied to simulators of the autonomous vehicles. A benefit of such testing was to provide more information to the developers. According to [Schultz et al. 1992],

> "In more of a qualitative affirmation of the method, the original designer of the AUTOACE intelligent controller was shown

some of the interesting scenarios generated by the GA, and acknowledges that they gave insight into areas of the intelligent controller that could be improved. In particular, the scenarios as a group tend to indicate classes of weaknesses, as opposed to only highlighting single weaknesses. This allows the controller designers to improve the robustness of the controller over a class as opposed to only patching very specific instances of problems."

3. TESTING AND DEBUGGING CHALLENGES

Reasoning about the overall quality of a system can be difficult. For example, suppose a system accepts some data value *X*, and that the user profile for this system specifies that user is likely to use values in the range $30 < X < 70$. A usage model may generate two test cases that specify *X* as 40 and 60. If both of these test cases pass, it is not necessarily true that test cases will pass for all values of X. Similarly, if both of these test cases fail, it is not necessarily true that test cases will fail for all values of X. Additional focused testing (using similar, but different, test cases to identify more precisely the usage scenarios that induce failures and the scenarios that do not induce failures) may be necessary to support reasoning about the overall quality of the software.

In addition, in the situation when failures are observed during system testing, more testing can be required in order to precisely determine nature and location of the error(s) that caused the failures so the developers can find and fix the error. This find-and-fix process is referred to as "debugging". According to [Myers 1979], "of all the software-development activities, [debugging] is the most mentally taxing activity." This statement is often true today and can be the source of software quality problems. Test cases that reveal failures are often dissimilar to each other, the test results often provide little information concerning the cause of the failure and whether a similar scenario would fail in a similar manner. Without additional information, and with limited development resources, developers may be tempted to apply a small patch to the software to work around the failure rather than perform the analyses necessary to support complete understanding and correction of the problems that caused the failures.

A competitive mentality of "developers versus testers" often exists during testing. Because debugging requires additional information concerning the usage of the system and performing additional testing, once failures occur and the system must be corrected, this mentality should transition to "developers *and* testers versus the system" to facilitate the debugging effort. Developers often need the support of the testers during debugging because the developers may not have the necessary testing

resources to do additional system level testing, or additional information concerning the usage of the system. As described by [Zeller 2001],

> "Testing is another way to gather knowledge about a program because it helps weed out the circumstances that aren't relevant to a particular failure. If testing reveals that only three of 25 user actions are relevant, for example, you can focus your search for the failure's root cause on the program parts associated with these three actions. If you can automate the search process, so much the better."

This description is consistent with the often-used induction approach to debugging described by [Myers 1979]. The induction approach begins by locating all relevant evidence concerning correct and incorrect system performance. As noted by [Myers 1979], "valuable clues are provided by similar, but different, test cases that *do not* cause the symptoms to appear. It is also useful to identify similar, but different, test case that *do* cause the symptoms to appear.

Similar to the notion of taking several "snapshots" of the evidence from different angles and with different magnification to look for clues from different perspectives, the debugging team needs to follow up on any failures identified during testing by more finely partitioning the input domain according to test results. This yields new evidence to be compared and organized in an attempt to identify and characterize patterns in the system's behavior. The next step is to develop a hypothesis about the cause of an observed failure by using the relationships among the observed evidence and patterns. Analyses can then be performed to prove that the hypothesis completely explains the observed evidence and patterns.

In practice, debugging can be very time-consuming, tedious, and error-prone when system-level testing reveals failures. Success of the debugging activity depends critically upon the first step in the process: the collection of evidence concerning correct and incorrect system performance. Assuming the total amount of evidence is manageable, an increase in useful evidence about correct and incorrect system performance can make it easier to identify patterns and develop and prove hypotheses. Thus, a mechanism is needed to drive testing and system analyses in a focused manner based on previous test results.

4. USING A GA FOR FOCUSED SOFTWARE USAGE TESTING

The genetic algorithm (GA) approach described in this paper drives dynamic generation of test cases by focusing the testing on high usage (frequency) and fault-prone (severity) areas of the software. This GA approach can be described as analogous to the application of a microscope. The microscope user first quickly examines an artifact at a macro-level to

locate any potential problems. Then the user increases the magnification to isolate and characterize these problems.

Using the GA approach to focused software usage testing, the macro-level examination of the software system is performed using the organization's traditional model-based usage testing methods. Based on the results of this macro-level examination, a genetic algorithm is used to select additional test cases to *focus* on the behavior around the initial test cases to assist in identifying and characterizing the types of test cases that induce system failures (if any) and the types of test cases that do not induce system failures. If failures are identified, the genetic algorithm *increases the magnification* by selecting certain test cases for further analysis of failures. This supports isolation and characterization of any failure clusters that may exist.

Whether or not any failures are experienced, this genetic algorithm approach provides increased evidence for the testing team and managers to support reasoning about the overall quality of the software. In the situation where failures are experienced, the genetic algorithm approach yields information about similar, but different, test cases that reveal faults in the software and about the input values that triggered the faults to induce failures. This information can assist the developer in identifying patterns in the system's behavior and in devising and proving a hypothesis concerning the faults that caused the failures.

Because different software failures vary in severity to the user and in frequency of occurrence under certain usage profiles, certain failures can be more important than others. Factors such as the development team's uncertainty about particular requirements, complexity of particular sections of the code, and varying skills of the software development team can result in clusters of failure in certain partitions of the set of possible use of the software. As discussed in section 5.4 and section 5.5.1, the genetic algorithm's fitness function and selection function can address this issue, and help support the generation of test cases to identify failure clusters.

In the case of usage testing, highly fit individuals in the population are those that maximize two objectives. The first objective is likelihood of occurrence. Maximizing this objective means that the test case individual represents a scenario that closely resembles what the user will do with the system. The second objective is failure intensity (defined as a combination of failure density and failure severity). Maximizing this objective means that the test case individual has revealed spectacular failures in the system. Highly fit individuals with respect to the rest of the population are those that maximize both objectives as much as possible. To address this issue, a multi-objective GA technique [Fonseca 1995; Deb 1999; Coello Coello et al. 2002] is needed. As described in section 5.4, this application makes use of a nonlinear aggregating fitness combination [Coello Coello et al. 2002] to handle multiple objectives.

Furthermore, the purpose of the GA in this application is not to find a single dominant individual. This does not make sense from a testing perspective. Instead, the purpose is to locate and maintain a group of individuals that are highly fit. To do so, the GA for this application uses niching [Holland 1975; Horn 1994; Mahfoud 1995]. A niche represents some subpopulation of individuals who are similar, but different. As the GA runs, the most dominant niches (not the most dominant individual) survive. Niching used for this application is described in section 5.4.3.

The GA approach is applicable to testing many types of software. For example, in section 6 illustrative examples are presented of the application of a GA to support high-level usage testing of a military simulation system. For this case study, the test cases for a military simulation system consists of a variety of scenarios involving entities such as tanks, aircraft, armored personnel carries, and soldiers. Each entity can perform a variety of tasks. At a basic level, these scenarios involve some primary actor performing a task that may or may not involve a secondary actor, depending on the task. Each scenario is performed on a specific terrain map. For example, a scenario may consist of using a terrain map of Fort Knox with an M1A1 tank performing an Assault on a T-80 tank. In this example, the M1A1 tank is the primary actor since it performs the task (Assault), and it is the focus of the scenario. The T-80 tank is the secondary actor.

5. GA APPROACH DETAILS

To implement the GA for this case study example, a number of issues had to resolved, including the encoding of real world data, population initialization, fitness evaluation, and the use and operation of genetic operators. The following subsections discuss these issues and describe the internal details of the genetic algorithm.

5.1 Input Domain Data

As illustrated in Figure 2, there are three sources for the Input Domain Data that serves as input to the GA application shown in Figure 1. First, there is data that represents the bounds of the input domain for the software under test. This boundary data set does not necessarily specify all possible data values; rather it could merely specify the extreme values. For example, suppose the system accepts some data value X. Then the input boundary data might specify $0 < X < 10$. Second, there is data that represents the user's profile. This data defines what input data the user is likely to use and, implicitly, what data the user is not likely to use. For the previous example of the data value X, the user profile may specify $3 < X < 7$. The third source of input domain data is the set of test cases generated according to the user profile. For example, there may be two test cases that specify X as 4 and 6. The test cases and user profile data sets must be subsets of the input boundary data set.

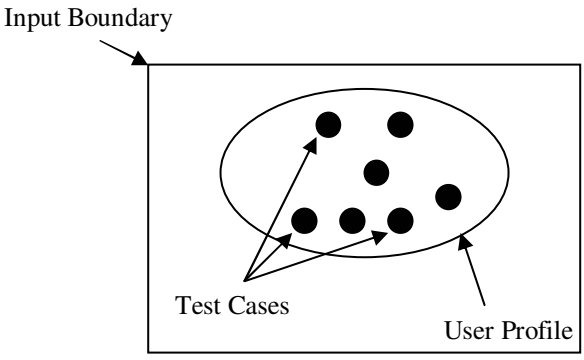

Figure 2. Input Domain Data

Each of these three sources of input domain data is used for a specific purpose. The test cases are used to initialize the population. The user profile data set is used to help evaluate the fitness of individuals, specifically used to determine likelihood of occurrence. This causes the GA to focus its search to a particular area of the input domain. The input boundary data set is used to validate that new individuals are consistent with what the software under test allows the user to do. If an individual is created that lies outside of the defined input boundary data set, then that individual will be discarded by the GA.

5.2 Encoding

The test cases generated by the usage model are converted to an encoding based on real numbers for use in the GA population. This type of encoding was used so that there is a one to one correspondence between the gene and the variable it represents. In addition, it eliminates the problem of Hamming cliffs [Goldberg 1990]. Table 1, Table 2, and Table 3 illustrate a sample of the assigned identification numbers (IDs) for use in the GA.

Table 1. Terrain Identification Numbers

Terrain ID Number	Terrain Map
1	NTC
2	Knox
3	Hunter
4	Itsec

Table 2. Entity Identification Numbers

Entity ID Number	Simulation Entity
2	M1A1
6	T-80
9	M3A3
17	SA-9
27	UH-60

Table 3. Task Identification Numbers

Task ID Number	Task
1	Move
3	Assault
7	Attack
11	Hover
15	Suppressive Fire

The individuals in the population of the GA consist of variations of these IDs. There are ten genes in each individual. The genotype is shown in Table 4.

Table 4. Genotype for individuals in the genetic algorithm

Gene	Meaning	Valid Value Range
1	Terrain	Values 1 – 4
2	Primary Actor	Values 1 – 37
3	Task	Values 1 – 17
4	Secondary Actor	Values 0 – 37
5	X_1	Values greater than or equal to 0
6	Y_1	Values greater than or equal to 0
7	Z_1	Values greater than or equal to 0
8	X_2	Values greater than or equal to 0
9	Y_2	Values greater than or equal to 0
10	Z_2	Values greater than or equal to 0

Each gene represents an input value that a user could supply to the software being tested. The collection of ten genes represents a specific simulation scenario that may be run by the user on the Software Under Test. For example, Gene 1 represents the terrain map selected by the user. Gene 2 represents the primary actor selected by the user, such as a tank (i.e., M1A1, T-80), plane, helicopter, etc. Gene 3 represents the task assigned by the user to the primary actor, such as Move, Attack, Transport, etc. If the selected

task requires a secondary actor, the user selects another actor, such as an enemy tank, enemy plane, friendly soldier, etc. Gene 4 represents the selected secondary actor. If the selected task does not require a secondary actor, Gene 4 is assigned a zero value. Genes 5 – 7 specify the location of the primary actor on the terrain map. If there is a secondary actor involved, then Genes 8 – 10 specify the location of the secondary actor on the terrain map. If there is no secondary actor, then Genes 8 – 10 represent some destination location that the primary actor must reach. An example of an individual is shown in Figure 3. This individual represents a scenario with an M1A1 tank assaulting a T-80 tank on the Fort Knox terrain map. The values shown in the first 4 genes of the individual are taken from Table 1, Table 2 and Table 3. The values for genes 5 – 10 are taken from the location values specified by the test case.

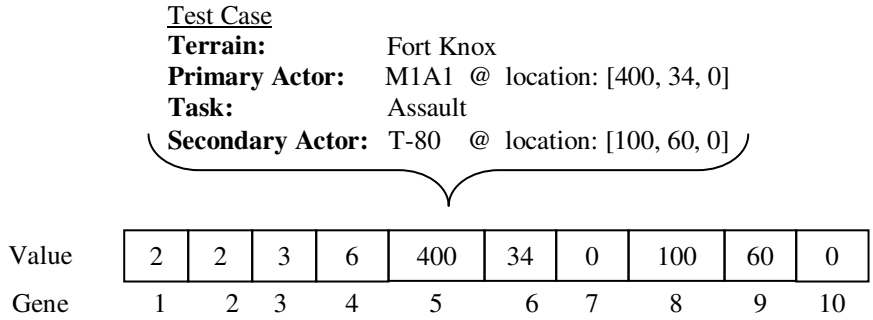

Figure 3. Representation of test cases within the genetic algorithm

Invalid individuals are discarded. For example, because a tank cannot attack an aircraft, an individual that represents this scenario would be discarded. Other invalid scenarios are those that specify locations (Genes 5 – 10) that lie outside the bounds of the terrain map. In addition, land vehicles cannot be assigned Z coordinate values greater than 0.

5.3 Population Initialization

To provide the GA with a semi-ideal starting position, individuals in the GA are initialized according to the test cases generated by the usage model. If the individuals in the GA were initialized randomly, the GA would 'waste' generation cycles looking for individuals located within the user profile. Furthermore, with random initialization, it is possible that the GA may not find the individuals located in the user profile, and the results will be of little value. Because some of the individuals located in the user profile are already known, initializing the population with these known individuals can reduce the number of GA iterations.

5.4 Fitness Evaluation

The fitness of individuals is based primarily on maximizing two objectives, as graphically depicted in Figure 4. Optimal individuals are those that have a high likelihood of occurring and that result in failures with high failure intensity. Optimal individuals occur in zone 1. Inferior individuals are those with a low likelihood of occurring and would be located in zone 6.

While the GA system strives to find optimal individuals, there are two reasons that this is not always achievable. First, the software under test may be of such high quality that optimal individuals simply do not exist. Second, optimal individuals may exist outside of the defined user profile, but not within it. If the GA finds such individuals, they will be in zone 6 if they lie outside of the high usage areas of the software as defined by the usage model. Note that the boundary between the optimal, sub-optimal, and inferior zones is not necessarily a hard, distinct boundary. Since the user profile is simply an approximation for what the user may do, inferior individuals near the boundaries of the optimal and sub-optimal zones may also be of interest.

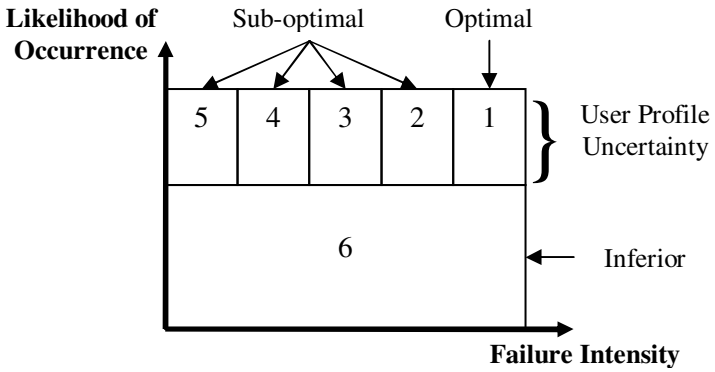

Figure 4. Fitness of Individuals

The height of the optimal and sub-optimal zones is determined by the uncertainty in the accuracy of the user profile. If the user profile is based on historical evidence, or if the profile represents expert users, then the uncertainty in the accuracy of the user profile will be lower, resulting in a shorter height of the zones. However, if the user profile is based on guesswork, or if it represents novice users, then the uncertainty in the user profile accuracy will be higher, resulting in a taller height of the zones. The width and number of the optimal and sub-optimal zones is chosen according to the level of importance given to the GA concerning various levels of failure intensity. For example, if each failure were of equal importance, there would be only one optimal zone, no sub-optimal zones, and a width ranging from the lowest intensity level to the highest.

The overall fitness of an individual is based on likelihood of occurrence, the failures intensity, and the similarity to other individuals in the population. Each of these criteria is discussed in the following sections.

5.4.1 Likelihood of Occurrence

Individuals are first evaluated in terms of the likelihood they will be used by the user. Individuals containing input data that is very likely to be used by the user are very highly fit individuals for this particular objective. Individuals that contain input data that is not likely to be used by the user are very poorly fit individuals. This evaluation is based on the supplied user profile data set. The likelihood of the input data is calculated by multiplying the probability of occurrence of each input value that is used in the test case. For example, suppose the probability distribution for the input data is as shown in Table 5. The likelihood that the user would select Input Values 1 and 2 is 0.15. The likelihood that the user would select Input Values 1 and 3 is 0.0375. Consequently, a test case involving Input Values 1 and 2 would be rated as being more highly fit than one involving Input Values 1 and 3. The case study described in this paper only considers the first 4 genes in determining the likelihood of occurrence. This is because genes 1 – 4 provide the basics of the test scenarios while genes 5 – 10 provide the details. Likelihood of occurrence is based on the basics, not the details, of the scenario.

Table 5. Input Data Probability Distribution.

Input Value 1	0.75
Input Value 2	0.20
Input Value 3	0.05

5.4.2 Failure Intensity

In addition to likelihood of use, the test team is also interested in test case individuals that find failures. Consequently, the second objective to be maximized is Failure Intensity, defined as a combination of failure density and failure severity. For example, suppose some individual causes a single failure that results in the crash of the software being tested. The Failure Intensity consists of a low failure density (there is only 1 failure) and a high failure severity (the system crashes). In contrast, suppose another individual causes multiple failures that give erroneous output but do not crash the software being tested. In this situation, the Failure Intensity consists of a high failure density (there were multiple failures) and a low failure severity (the system does not crash, but gives erroneous output). Both of these individuals would be of interest, even though the composition of their Failure Intensity is different.

Consider the situation where a test manager differentiates failure severity according to five levels, with level 1 the lowest severity and level 5 the highest. For an individual test case that causes two level 3 failures, the failure intensity could be computed to equal 6, the sum of the failure severities. An individual that causes one level 5 failure would have failure intensity equal to 5. However, this approach to calculating failure intensity may not be satisfactory to the test manager. A single level 5 severity failure may be more important than a test case that produces multiple failures of lower severity. To handle this situation, a non-linear scoring method such as that shown in Table 6 is recommended.

Table 6. Example Scoring Technique for Different Severity Levels

Severity Level	Score
5	18
4	12
3	3
2	2
1	1

If this scoring technique were applied, an individual that caused two level 3 failures would receive a failure intensity score of 6, and an individual that caused a single level 5 failure would receive an intensity score of 18. Similarly, an individual that caused three level 2 failures and two level 3 failures would receive an intensity score of 12. This yields a more useful result to the test manager than a linear scoring method. Obviously, the choice of scoring algorithm depends on the characteristics of the software being tested and the test management objectives.

5.4.3 Niching

As a genetic algorithm runs, the population of individuals will eventually converge to a single solution that dominates the population, and the diversity of the population is ultimately lowered. When a GA is applied to software usage testing, each individual represents a single test case. Consequently, the genetic algorithm would eventually converge to some test case that is both likely to occur and reveals failures of high intensity. To avoid having a single individual dominate the population, a niching technique [Holland 1975; Mahfoud 1995; Horn 1997] is used.

A niche represents some subpopulation of individuals who share some commonality. To apply this technique to software usage testing, a niche is formed for each unique combination of likelihood, failure intensity, and genetic values for the genes 1 through 4. That is, individuals that share the same likelihood, failure intensity, and genes 1 through 4 will occupy the same niche, or subpopulation. For example, a niche would be represented by a likelihood value of .07, a failure intensity value of 12, and genes values {2

2 3 6} for genes 1 through 4. In a population of 500, there may be 20 individuals who have these same values and would, consequently share this same niche. Another niche would be represented by a likelihood value of .05, a failure intensity of 10, and gene values {1 3 3 5} for genes 1 through 4. This type of niching is based on both the phenotype and partial genotype of the individuals. By implementing niches in the GA, the population will converge not to a single dominant individual, but to multiple dominant niches.

Specifically, niching is performed based on fitness sharing [Holland 1975]. Fitness sharing reduces the fitness values of individuals that are similar to other individuals in some way (i.e., the various niches in the population). This type of niching was used because of its success in prior work [Mahfoud 1995]. For this application, an individual's fitness value is reduced by dividing its fitness by the number of individuals that share its same niche.

5.4.4 Determining Overall Fitness

Highly fit individuals in the population are those maximize the objectives of likelihood of occurrence and failure intensity. A nonlinear aggregating fitness combination [Coello Coello et al. 2002] is used to identify individuals based on these two objectives. Determining failure intensity is already time consuming, therefore, this type of fitness combination was selected for its simplicity and speed. In addition, it directly addressed the needs of this particular case study.

Each individual *i* is given a combined fitness value that is based on the likelihood of occurrence of individual *i*, the failure intensity revealed by individual *i*, and the total number of individuals in the population *p* that also occupy the same niche as individual *i*. The fitness function to calculate the overall fitness value for an individual *i* is given as follows:

$$Fitness(i) = \frac{\left(Likelihood(i) \times Intensity(i)\right)^{y}}{Niche\ Size(p, i)}$$

Equation 1. GA Fitness Function

The variable *y* represents a nonlinear scaling factor that can be adjusted by the test team. This scaling factor is independent of the individuals in the population. Using the microscope analogy, the *y* value is analogous to the magnification level of the microscope. A higher *y* value represents a higher magnification, and vice versa. The higher the value of *y* used in the GA, the faster the population will converge to the most dominant niches, and the less diversity there will be in the population. The lower the value of *y*, the slower the population will converge and the more diversity

there will be in the population (assuming that there is no one individual that is exceptionally fit).

If the scaling factor is not high enough, optimal individuals may not be found, or would be lost in the process. This may occur in large populations when weaker individuals may dramatically outnumber more optimal individuals. A higher scaling factor will help optimal individuals survive in a large mass of weaker individuals.

5.5 Genetic Operators

To create children from a given population, genetic operators such as selection, crossover, and mutation operators are applied to the individuals. Selection is first used to select parents from the population according to the overall fitness value, as discussed in section 5.4. Strongly fit individuals (higher fitness values) are more likely to be selected for reproduction than weaker individuals (lower fitness values). Consequently, the average population fitness should improve with each generation. Once parents are selected, crossover and mutation operators are applied to the parents to create children. The crossover and mutation operators provide the GA with the ability to explore the search space for new individuals and to create diversity in the population. The final result is a new population representing the next generation.

5.5.1 Selection

The GA selection process used for this application is the Fitness Proportional Selection [Holland 1975]. With this process, an individual's probability of being selected for reproduction is proportional to the individual's fitness with respect to the entire population. Each individual's fitness value is divided by the sum of the fitness values for all the individuals in the population. The resulting fitness value is then used to select parents, who then have the opportunity to pass on their genetic material (encoded information) to the next generation. Highly fit individuals are therefore more likely to reproduce. This helps to improve the quality of the population. An example of fitness proportional values is shown in Table 7. As can be seen, individual 4 is the most likely to be selected, and individual 2 is the least likely to be selected. Since this process depends on an individual's fitness proportional to the population, the tester can easily influence the selection process by altering the scaling factor of the fitness function, as discussed in section 5.4.4.

Table 7. Example of fitness proportional values

Individual	Original Fitness Value	New Fitness Value
1	2	2 / 21 = .0952
2	1	1 / 21 = .0476
3	4	4 / 21 = .1904
4	9	9 / 21 = .4285
5	5	5 / 21 = .2381
Sum	21	.9998

5.5.2 Crossover

To create children, the GA for this application uses a single-point crossover operator that takes two parent individuals as input and outputs two children that are similar, but different, from the parents. This operator randomly selects a point in the genetic code of two parents and then swaps all genes between the parents that lie after the crossover point. When crossover is allowed between parents from different niches, diversity is encouraged. For this case study, every individual in each generation is processed by the crossover operator, and, if a child represents an invalid scenario, it is discarded from the population and replaced by its corresponding parent. For example, if Child 1 were invalid, it would be removed and replaced by Parent 1. The basic operation of crossover is shown in Figure 5.

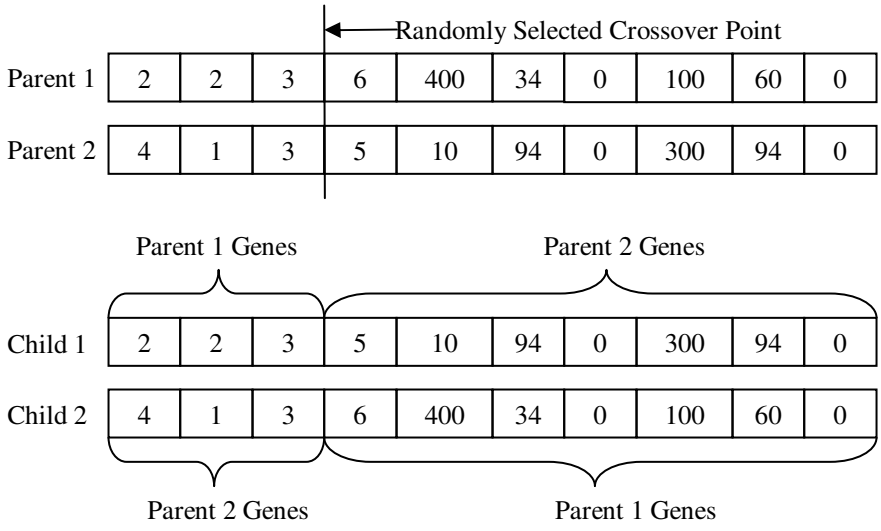

Figure 5. One-point crossover

5.5.3 Mutation

In addition to the crossover operator, the GA for this application uses a single-point mutation operator that takes one individual as input, makes a small, random change to the genetic code of this individual, and outputs one mutant that is similar, but different, to the original individual. This operator randomly selects a gene in the genetic code of an individual and mutates that gene by randomly selecting some new value. For this case study, every individual in each generation is processed by the mutation operator, and, if the mutant represents an invalid scenario, it is discarded from the population and replaced by the original individual. The basic operation is shown in Figure 6.

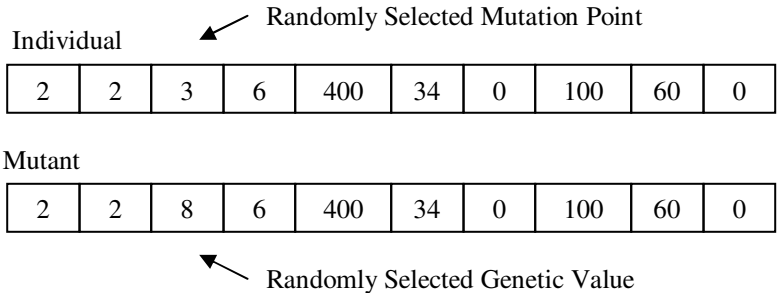

Figure 6. One-point mutation

6. EXAMPLE

The application of the GA to software usage testing was based on a military simulation system. The population of interest for the examples included four terrain maps, thirty-seven primary and secondary actors, and seventeen tasks that are available for use with OTB.

To focus on observing and understanding the behavior of the GA for use in software testing, the Failure Intensity Evaluation portion of Figure 1 was simulated. Test cases were not actually performed on the military simulation system. A set of simulated failures was developed for use in all the examples. Simulated failures included problems with terrain maps, problems with a specific entity or task regardless of terrain, actor, etc. These simulated failures were representative of the types of problems seen in the real system. Failure intensities greater than 12 represented system crashes. Failure intensities less than 12 represented non-terminating failures. The scoring system used is shown in Table 8. This is the same scoring technique proposed in Table 6. Multiple failures per test case were also simulated. As a result, a test case may reveal a failure intensity of 5, meaning that there were two failures of with a score of 3 and 2, respectively.

Table 8. Failure intensity scoring system

Score	Meaning
18	Repeatable, terminating failure
12	Irregular, terminating failure
3	Repeatable, non-terminating failure
2	Irregular, non-terminating failure
1	No failures

Two similar, but slightly different, user profiles were developed to examine the behavior of the GA when slight changes in a user profile occur. Sample test cases were generated for each user profile. The GA was initialized using each set of sample test cases, the corresponding user profile, and the input boundary (as described in section 5.1). For all the GA runs, the population size was 100 and the number of generations was 30.The results for three examples of the GA are shown in Figure 7, Figure 8, Figure 9, Figure 10, Figure 11 and Figure 12. Each point on the graphs represents a niche in the population, not a single individual. The data supporting these figures is shown in Table 9,

Table 10, Table 11, Table 12, Table 13, and Table 14, respectively. These tables also show how many individuals occupy each niche.

In the first example, Figure 7 shows the niches that were formed after the fitness evaluation of the first generation formed from test cases generated according to User Profile 1. Figure 8 shows the niches that were formed after the fitness evaluation of the thirtieth generation. Notice that after 30 generations, the GA has converged to a few dominant niches. A comparison of Figure 7 and Figure 8 indicates that the GA has found four more niches that are very likely to occur and contain high failure intensities. Weaker niches did not survive.

In the second example, Figure 9 shows the niches that were formed after the fitness evaluation of the first generation formed from test cases that were generated according to User Profile 2. Figure 10 shows the niches that were formed after the fitness evaluation of the thirtieth generation. Notice that after 30 generations, the population of the GA has not converged sufficiently, but rather grew more divergent. This suggests that the fitness function and selection process are not sufficiently countering the effects of the crossover and mutation operators.

In the third example, the GA was reapplied using the same input data as in the second example. However, the scaling factor of the fitness function was increased from a value 1 to 2. This was done to increase the convergence of the population, so that the final population does not grow more divergent as in the second example. The initial niches for this example of the GA, shown in Figure 11, were the same as for the second example (i.e., Figure 11 is identical to Figure 9). However, as shown in Figure 12, the

results were much different from that of Figure 10. These results are very similar to those shown in Figure 8. The GA has found four new niches that are very likely to occur and contain high failure intensities. The weaker niches did not survive.

The third example demonstrates a key aspect of the fitness function of the GA. The scaling factor of the fitness function plays a critical and delicate role in the finding and maintaining of optimal solutions. As illustrated in the second example, if scaling factor is too low, optimal solutions may not be found because the level of exploitation is diminished. However, if the scaling factor is too high, diversity and exploration will be diminished.

In the last two examples, the GA was able to overcome a less than optimal initial population. Notice in Table 11 and Table 13 that the initial populations were heavily biased towards the niche with the highest likelihood and low failure intensity. Table 12 and Table 14 show that the final populations are more balanced (in comparison to Table 11 and Table 13, respectively), and resulted in niches that are more interesting in terms of high failure intensity, while also being very likely to occur.

Finally, in each example, the final populations consist of niches that are:

1. Very likely to occur and resulted in a high failure intensity
2. Similar, but different. As described in [Myers 1979], similar, but different, test cases help to identify the failure's root cause.

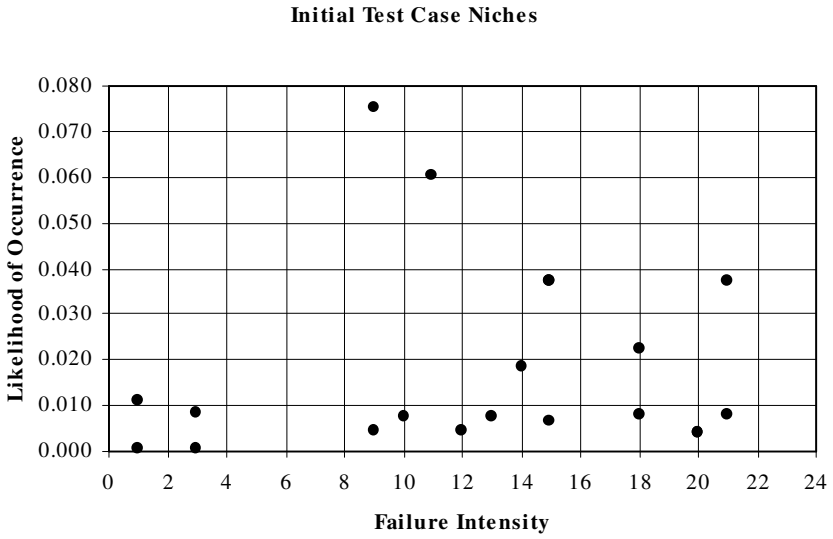

Figure 7. Test case niches for User Profile 1 after Generation 1.

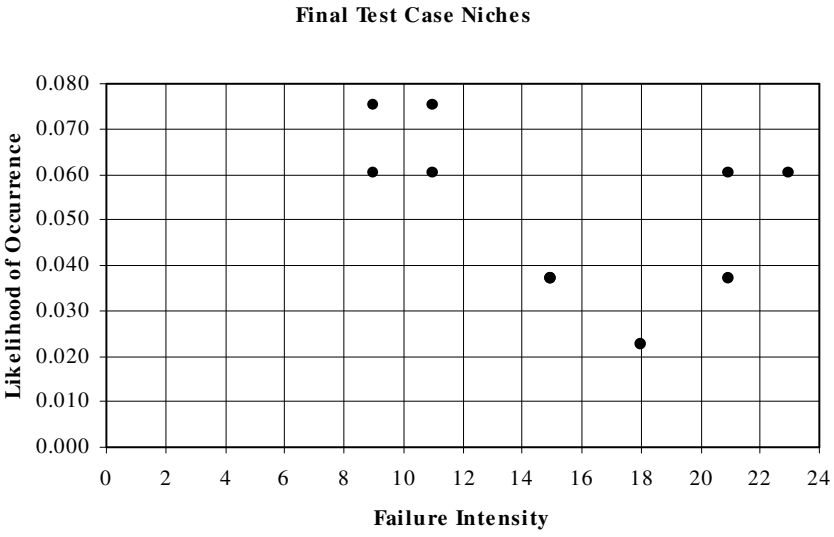

Figure 8. Test case niches for User Profile 1 after Generation 30.

Table 9. Number of individuals for test case niches shown in Figure 7

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0005	1	Itsec	M16A2	Suppressive Fire	AK47	6
0.0005	3	Itsec	M16A2	Suppressive Fire	AK47	1
0.0040	20	Hunter	AH-64	Recon	SA-9	1
0.0040	20	Hunter	AH-64	Recon	SA-15	7
0.0043	9	Itsec	M16A2	Location Fire	AK47	5
0.0043	12	Itsec	M16A2	Location Fire	AK47	1
0.0067	15	NTC	M1A1	Assault	BMP-2	7
0.0072	10	Itsec	AH-64	Attack	T-72	3
0.0072	13	Itsec	AH-64	Attack	T-72	3
0.0080	18	Itsec	M3	Transport	SAW Gunner	5
0.0080	21	Itsec	M3	Transport	SAW Gunner	1
0.0083	3	Knox	M1A1	Assault	SA-9	7
0.0111	1	Knox	AC-130	Attack	SA-15	6
0.0185	14	Knox	AH-64	Recon	BMP-2	6
0.0223	18	NTC	M3	Transport	DI-M224	7
0.0370	15	Knox	AC-130	Ingress	SA-15	7
0.0370	15	Knox	AC-130	Ingress	SA-9	7
0.0370	21	Knox	AC-130	Ingress	T-80	7
0.0603	11	NTC	M1A1	Assault	T-72	7
0.0750	9	Knox	M1A1	Assault	T-80	6

Table 10. Number of individuals for test case niches shown in Figure 8

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0223	18	NTC	M3	Transport	SAW Gunner	5
0.0223	18	NTC	M3	Transport	DI-M224	7
0.0370	15	Knox	AC-130	Ingress	SA-15	3
0.0370	15	Knox	AC-130	Ingress	SA-9	8
0.0370	21	Knox	AC-130	Ingress	T-80	8
0.0603	9	NTC	M1A1	Assault	T-80	6
0.0603	11	NTC	M1A1	Assault	T-72	5
0.0603	21	NTC	M1A1	Assault	T-80	19
0.0603	23	NTC	M1A1	Assault	T-72	13
0.0750	9	Knox	M1A1	Assault	T-80	11
0.0750	11	Knox	M1A1	Assault	T-72	15

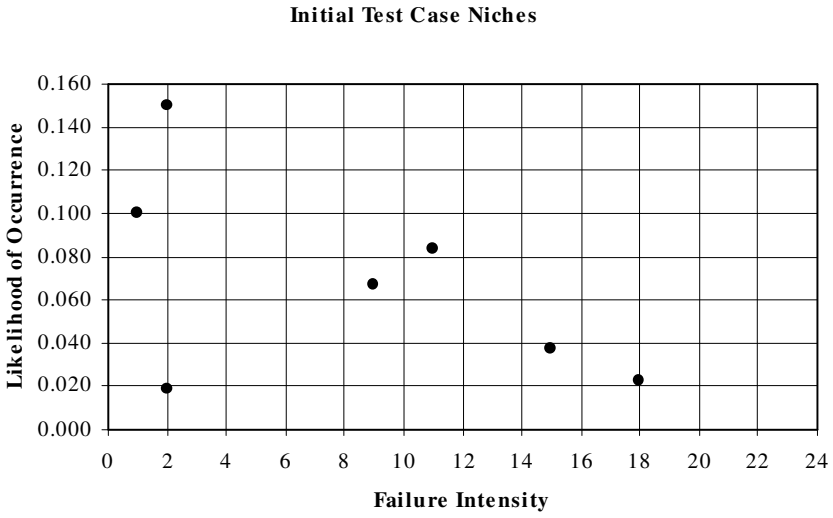

Figure 9. Test Case Niches for User Profile 2 after Generation 1

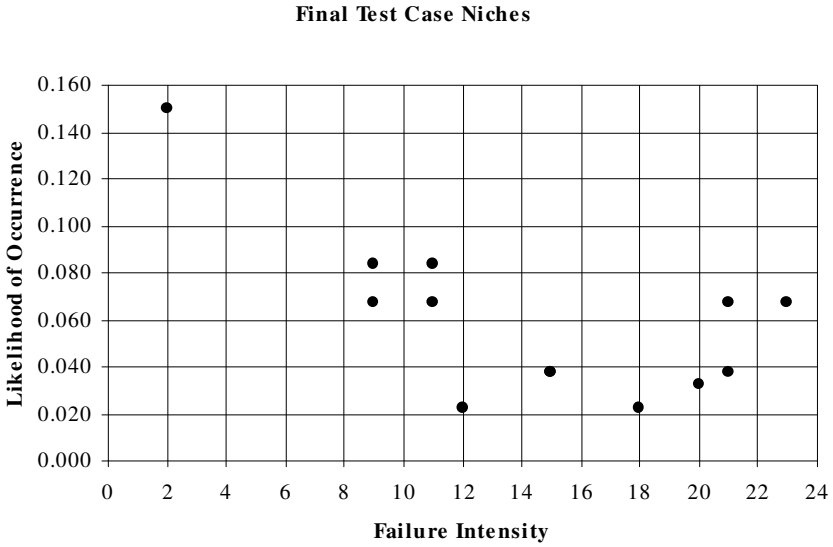

Figure 10. Test Case Niches for User Profile 2 after Generation 30

Table 11. Number of individuals for test case niches shown in Figure 9

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0185	2	Knox	AH-64	Recon	SA-9	8
0.0223	18	NTC	M3	Transport	SAW Gunner	8
0.0370	15	Knox	AC-130	Ingress	SA-15	7
0.0669	9	NTC	M1A1	Assault	T-80	8
0.0833	11	Knox	M1A1	Assault	T-72	8
0.0999	1	Knox	AC-130	Attack	SA-9	8
0.1499	2	Knox	AH-64	Attack	SA-9	53

Table 12. Number of individuals for test case niches shown in Figure 10

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0223	12	NTC	M3	Transport	DI-M224	4
0.0223	18	NTC	M3	Transport	DI-M224	2
0.0321	20	Hunter	AH-64	Attack	SA-9	7
0.0370	15	Knox	AC-130	Ingress	SA-15	10
0.0370	15	Knox	AC-130	Ingress	SA-9	5
0.0370	21	Knox	AC-130	Ingress	T-80	4
0.0669	9	NTC	M1A1	Assault	T-80	9
0.0669	11	NTC	M1A1	Assault	T-72	8
0.0669	21	NTC	M1A1	Assault	T-80	18
0.0669	23	NTC	M1A1	Assault	T-72	13
0.0833	9	Knox	M1A1	Assault	T-80	5
0.0833	11	Knox	M1A1	Assault	T-72	9
0.1499	2	Knox	AH-64	Attack	SA-9	6

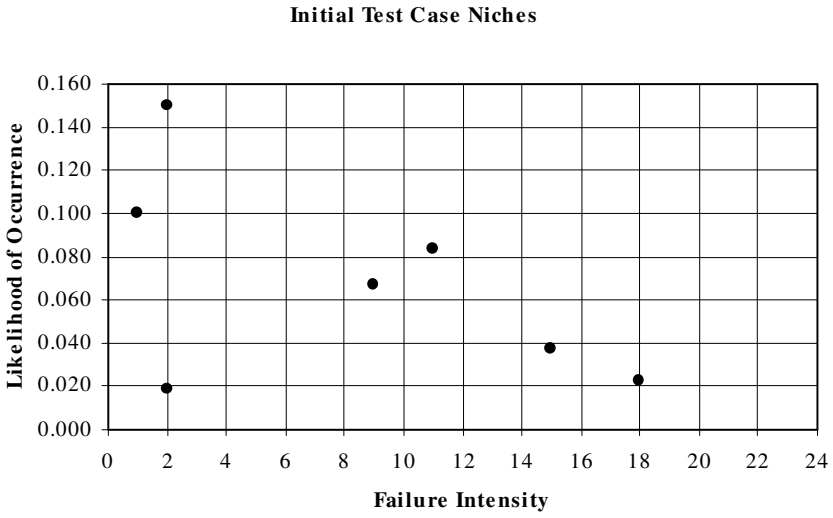

Figure 11. Test Case Niches for User Profile 2 after Generation 1 with scaling factor of 2.

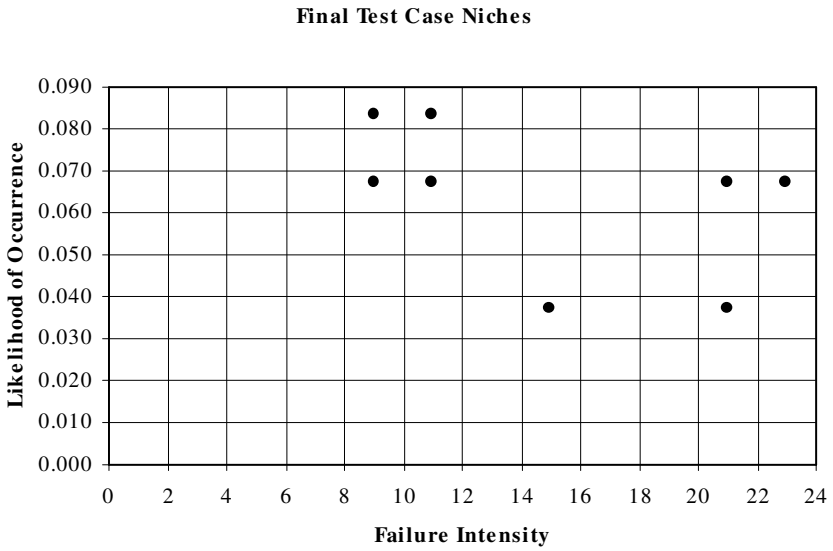

Figure 12. Test Case Niches for User Profile 2 after Generation 30 with scaling factor of 2.

Table 13. Number of individuals for test case niches shown in Figure 11

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0185	2	Knox	AH-64	Recon	SA-9	8
0.0223	18	NTC	M3	Transport	SAW Gunner	8
0.0370	15	Knox	AC-130	Ingress	SA-15	7
0.0669	9	NTC	M1A1	Assault	T-80	8
0.0833	11	Knox	M1A1	Assault	T-72	8
0.0999	1	Knox	AC-130	Attack	SA-9	8
0.1499	2	Knox	AH-64	Attack	SA-9	53

Table 14. Number of individuals for test case niches shown in Figure 12

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0370	15	Knox	AC-130	Ingress	SA-15	5
0.0370	21	Knox	AC-130	Ingress	T-80	7
0.0669	9	NTC	M1A1	Assault	T-80	2
0.0669	11	NTC	M1A1	Assault	T-72	9
0.0669	21	NTC	M1A1	Assault	T-80	30
0.0669	23	NTC	M1A1	Assault	T-72	31
0.0833	9	Knox	M1A1	Assault	T-80	6
0.0833	11	Knox	M1A1	Assault	T-72	10

7. CONCLUSIONS

This paper introduces a genetic algorithm approach to software usage testing that is used to explore the space of input data and identify and focus on regions that cause failures. Analysis of the examples in this paper demonstrates that genetic algorithms can be used as a tool to help a software tester search, locate, and isolate failures in a software system. The use of genetic algorithms supports automated testing and helps to identify those failures that are most severe and likely to occur for the user.

The strategy presented in this paper relies on a technique that not only helps the tester to isolate failure clusters, but also provides the developer with more information concerning the faults in the software and the input values that triggered them. The developer can then use this information to search, locate, and isolate the faults that caused the failures. The result can improve efficiency of both the testing and the development teams and can support subsequent improvements in the software development process.

The examples discussed in this paper raise a number of new ideas and issues for future consideration, such as the use of a global parallel genetic algorithm, different representation scheme, restrictive mating, and genetic algorithm parameter sensitivity to different user profiles. For example, current testing practice involves several testers working on different test cases at the same time. For the example application discussed in this paper, the fitness evaluation lends itself readily to parallelism. A global parallel genetic algorithm could take advantage of this parallelism. Such an approach could provide automated support to the current testing practice of distributed work effort. While each of these areas for future consideration could be further investigated with respect to applicability for software testing, as demonstrated by the examples of this paper, the simple genetic algorithm approach presented in this paper provides in itself a useful contribution to the selection of test cases and a focused examination of test results. Thus, application of this approach can support reasoning about test results to support quality system assessment and/or debugging activities.

ACKNOWLEDGEMENT

This work was funded in part by NAWC-TSD Contract N61339-01-D-002.

REFERENCES

Chambers, L., Ed. (2000), *The Practical Handbook of Genetic Algorithms: Applications, Second Edition*, Chapman & Hall / CRC.

Coello Coello, C.A., D.A.V. Veldhuizen, G.B. Lamont (2002), *Evolutionary Algorithms for Solving Multi-Objective Problems*, Kluwer Academic Publishers, New York, NY.

Coley, D. A. (2001), *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific, River Edge, NJ.

Deb, K. (1999), "Multi-objective genetic algorithms: Problem difficulties and construction of test problems", *Evolutionary Computation Journal*, 7,3, 205-230.

Dickinson, W., D. Leon, and A. Podgurski (2001), "Pursuing Failure: The Distribution of Program Failures in a Profile Space." In *Proceedings of the 9th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 246 – 255.

Drake, T (2000), "Testing Software Based Systems: The Final Frontier." *Software Tech News*, Vol. 3, No. 3.

Fonseca, C. M. and P. J. Fleming (1995), "An overview of evolutionary algorithms in multiobjective optimization", *Evolutionary Computation Journal*, 3,1, 1-16.

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Goldberg, D.E. (1990), "Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking," Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois, Tech. Rep.90001.

Haupt, R. L., and S. E. Haupt (1998), *Practical Genetic Algorithms* John Wiley & Sons, Inc. New York, NY.

Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

Horn, J. (1997), "The Nature of Niching: Genetic Algorithms and the Evolution of Optimal, Cooperative Populations", PhD Thesis,

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

IEEE Std. 829-1998, IEEE Standard for Software Test Documentation.

Karr, C. L., and L. M. Freeman, Ed. (1999), *Industrial Applications of Genetic Algorithms*, CRC Press, New York, NY.

Mahfoud, S. W. (1995), "Niching Methods for Genetic Algorithms." PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Mitchell, M. (1996), *An Introduction to Genetic Algorithms*, MIT Press.

Myers, G.J. (1976), *Software Reliability*, John Wiley & Sons, Inc., New York.

Myers, G.J. (1979), *The Art of Software Testing*, John Wiley & Sons, Inc., New York.

Schultz, A. C., J. J. Grefenstette, and K. A. De Jong (1992), "Adaptive testing of controllers for autonomous vehicles." In *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology*, pp. 158 – 164.

Zeller, A. (2001), "Automated Debugging : Are We Close?" *IEEE Computer,* 34, 11, 26-31.