Automatic Generation of Object-Oriented Tests with a Multistage-Based Genetic Algorithm

Ahmed S. Ghiduk
Department of Mathematics, Faculty of Science, Beni-Suef University, Egypt
Department of Computer Science, College of Computers and Information Systems,
Taif University, Saudi Arabia
asaghiduk@tu.edu.sa

***Abstract*—The widespread use of the object-oriented programs (*OOPs*) makes the requirement for tests-generation strategies for testing the *OOPs* increases from day to day. In this paper, we present a multi-stage genetic algorithm (*MSGA*) to generate a suite of tests for testing the *OOPs*. *MSGA* includes two optimization stages. The first stage concentrates on finding test cases (sequences of called methods), which satisfy a given test criterion. The second stage focuses on generating test data (values of the arguments of the called methods). In addition, we introduce a new chromosome representation, which consists of two concatenated one-dimensional arrays. Each array contains set of homogeneous genes. In addition, we introduce set of strategies for encoding and decoding the tests. Furthermore, we present set of new genetic operators and the required pre- and post-conditions for applying these operators. In order to determine the applicability and practicability of *MSGA*, we introduce a new testing tool by implementing *MSGA*. Also, we conduct a case study by the new tool to assess the efficiency of *MSGA* in data-flow testing of *OOPs*.**

***Index Terms*— Object-Oriented Testing, Test Generation, Genetic Algorithms, Class Control-Flow Graph, Dominance.**

I. Introduction

Test generation for structural programs is the process of identifying test data, which execute the program under test and satisfy a given test criterion. *Test generation for the object-oriented programs* involves generating: 1) test cases, which are sequences of methods issue on an object of the class under test (*CUT*) and satisfy a given test criterion, and 2) test data, which is a set of values for the arguments of the called methods.

Pargas, Harrold, and Peck [1] presented a generic genetic algorithm, which is successfully applied to generate tests for the structural programs. Buy, Orso, and Pezzè [2] employed symbolic execution and a deduction technique for generating test cases to cover the def-use pairs of *CUT*. Martena, Orso, and Pezzè [3] extend their previous work for automatic generation of test cases for interaclass testing (i.e., test of interactions among methods of the *CUT*) to address the problem of interclass testing (i.e., test of interactions among classes). This technique does not handle the problem of test-data generation. Jiménez et al., [4] presented a test-data generation technique, which uses an algebraic model to represent *CUT*. The technique randomly generates an initial set of test data. Then, the technique applies the cross product on this set of data to find the arguments of the invoked constructors and methods of *CUT*. The limitations of this technique are: 1) the algebraic model cannot handle the complex data type such as object type, and 2) it does not address with finding test cases. Tonella [5] applied the traditional genetic algorithm to generate test data for unit testing of classes. He represented the chromosome as an array of constructors and methods, and their arguments. He introduced some genetic operators. The fitness of an individual is the overlap between the execution traces and control/call dependences leading to currant target. He used all-branches as a test criterion. Data-flow testing of classes is beyond the scope of this work. Further, the representation causes changing the state of the target object throughout the generations. Wappler and Lammermann [6] used the traditional genetic algorithms for testing object-oriented programs. They presented a chromosome representation, which contains information about the called methods, and their return values and parameters. They suggested strategies for encoding and decoding the three components of the chromosome to genes. This technique does not handle data-flow testing of the *OOPs*. Chromosome's representation delimits the diversity of the population's individuals. Cheon, Kim, and Perumandla [7] proposed a technique that combines JML (Java Modeling Language) and the traditional genetic algorithms to unit testing of object-oriented software. They randomly generate the test data. They suggested a fitness function based on the specification of the class under test to guide the genetic algorithm to generate the test data [8]. They proposed a fitness function based on assertions such as method preconditions to find feasible sequences of called methods [9]. Testing intra- and inter-class are out the scope of this work.

Traditional genetic algorithms optimize one problem at a time. Testing the *OOPs* requires optimizing two problems (the called methods and the values of their arguments) simultaneously. Therefore, many modifications in the structure of the traditional genetic algorithms are required to generate tests for the *OOPs*. These modifications include the fitness function, the representation of the chromosome, and genetic operators (crossover and mutation).

doi:10.4304/jcp.5.10.1560-1569

To overcome the limitations of the previous work, we present in this paper a multistage genetic algorithm (*MSGA*) to generate at once a suite of tests for the *OOPs*. *MSGA* contains two optimization stages. The first stage concentrates on finding the test cases, which covers a given test criterion. The second stage focuses on generating the test data. In addition, we introduce a chromosome representation, which consists of two distinct parts of homogeneous genes. We suggest strategies for encoding and decoding tests to chromosomes. We present set of genetic operators to work with the new representation and the pre- and post-requisites to apply these operators.

The rest of the paper is organized as follows. Section 2 gives some basic concepts and definitions. The description of our multistage genetic algorithm *MSGA* is presented in Section 3. Section 4 describes the new chromosome representation and the encoding and decoding strategies. The description of the genetic operators is provided in section 5. Section 6 presents a fitness function to evaluate the generated tests. Section 7 describes the architecture of the implementation of *MSGA* for data-flow testing. Section 8 presents a case study to evaluate *MSGA*. Section 9 gives the conclusions and future work.

II. BACKGROUND

This section gives the concepts and definitions of the class control-flow graph, data-flow testing of classes, dominance, and genetic algorithms.

A. The class control-flow graph (CCFG)

For an individual method, a directed control-flow graph $G = (N, E)$ with a unique entry node (n_0) and a unique exit node (n_k) consists of a set N of nodes, where each node represents a statement (or a group of consecutive statements), and a set E of directed edges, where a directed edge is an ordered pair of adjacent nodes.

For testing the *OPPs*, the structure of the class under test (*CUT*) represents by a model called the class-call graph. A class-call graph is a directed graph in which nodes represent methods, and edges represent procedure calls between methods.

The class control-flow graph is a class call graph in which each node replaced with the control-flow graph of the corresponding method such that each call site in it replaces with call and return nodes and the individual control-flow graphs of all methods of the class are connected together by edges [13]. Table 1 gives the edges and nodes of the class *CoinBox*, which is given in Figure 3.

B. Data-flow testing of classes

Data-flow testing focuses on execution all the interactions between the variables of the unit under test. Data-flow analysis identifies all definition-use associations (*dua*) for any variable v of the unit under test. A *dua* is an order triple (d, u, v), where d is a statement containing a definition of the variable v (i.e., it assigns a value to v) and u is a statement containing a use of the variable v (i.e., it reads the value of v or some memory bound to v) that can reached by d over some paths in the unit under test. A def-clear path from statement d to statement u with respect to a variable v is a sequence of statements from d to u without any redefinition of v. A def-kill path from statement d to statement u with respect to a variable v is a sequence of statements from d to u with a new definition for v. Data-flow testing of classes considers the following definitions for a member method m of the *CUT*:

1. m is called a def-clear method with respect to a variable v if and only if every feasible (executable) path in m is a def-clear path with respect to v.
2. m is called a def-kill method with respect to a variable v if and only if every feasible path in m is a def-kill path with respect to v.
3. m is called semi-def-clear or semi-def-kill method with respect to a variable v if and only if there is at least one feasible path in m contains a definition for v.
4. An inter-method def-clear path with respect to a variable v is concatenation of def-clear paths in a sequence of methods.

For a given def-use (d, u, v) of a class C, a test is a sequence of methods invocations appended by their parameters' values that satisfies the following:

- Begin with an invocation of a constructor of C.
- Contain a call for method m_d that causes directly or indirectly the execution of d.
- Contain a call for method m_u that causes directly or indirectly the execution of u.
- There is an inter-method def-clear path from the invocation of m_d to the invocation of m_u and throughout the sequence of invocations between m_d to m_u.

We used the technique presented by Harrold and Rothermel [13] to build *CCFG* and the technique produced by Pande et al. [14] to find all *dua* of the *CUT*. Table 2 and Table 3 give some examples for *dua* of class *CoinBox*.

C. Dominance

Let $G = (N, E)$ be a directed graph with two distinguished nodes n_0 and n_k. A node n dominates a node m if every path P from the entry node n_0 to m contains n [15].

By applying the dominance relations between the nodes of a directed graph G, we can obtain a tree (whose nodes represent the directed graph nodes) rooted at n_0. This tree is called the dominator tree $DT(G)$. A (rooted) tree $T = (N, E)$ is a directed graph in which one distinguished node n_0, called the root, is the head of no edges; every node n except the root n_0 is a head of just one edge and there exists a (unique) path from the root n_0 to each node n. A dominance path in $DT(G)$ is a sequence of nodes $dom(n_q)=n_1,n_2,\ldots,n_q$ where n_1 is the root and n_i is the parent of n_{i+1} in $DT(G)$ for $i=1,\ldots,q-1$. The dominator tree of method *addQtr* of the example class in Figure 3

contains nodes 13, 14, 15, 16, 17, 18, 19, and 20 and edges (13,14), (14,15), (15,16), (16,17), (16,20), (17,18), and (18,19).

D. The Traditional Genetic Algorithms

The basic concepts of the traditional genetic algorithms (GAs) were developed by Holland [10]. The principle behind GAs is that they create and maintain a population of individuals represented by chromosomes. These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to rules of selection, mutation and reproduction. Each individual in the environment (represented by a chromosome) receives a measure of its fitness in the environment. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The structure of a simple GA is given below.

```
Simple Genetic Algorithm ()
{     initialize population;
      evaluate population;
      while termination criterion not reached
      {     select solutions for next population;
            perform crossover and mutation;
            evaluate population; }
}
```

The previous algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have occurred.

III. Our Multi-Stage Genetic Algorithm

In this section, we present a Multi-Stage Genetic Algorithm (*MSGA*) to generate tests for the object-oriented programs. Figure 1 shows the overall algorithm of *MSGA* which consists of two nested optimization stages. The first-optimization stage finds the required called methods by optimizing *popSize* sequences of methods, where *popSize* is the number of individuals in the population and each sequence starts with a constructor. Second-optimization stage finds values of the arguments by optimizing *n* values, where *n* is the number of arguments in the sequence of called methods.

The inputs of *MSGA* are: 1) *CUT*: an instrumented version of the class under test, and 2) *TestReq*: test requirements, which satisfy a given control- or data-flow test criterion.

The output of *MSGA* is, *Final*, set of tests that covers the test requirements.

MSGA uses local variables *MethodsSeq* (set of called methods), *ParametersSeq* (set of values for the arguments), *CurPopulation* and *NewPopulation* (sets of tests), *T* (a single test requirement), and *Scoreboard* (set of covered test requirements). In addition, *MSGA* uses the following alarm functions 1) *Max_Attempts* returns *true* when maximum number of attempts for *T* has been exceeded, 2) *Max_Gen* returns *true* when maximum number of called methods for *T* has been exceeded, and 3) *Out_of_Time* returns *true* when the time limit is exceeded and *false* otherwise. *MSGA* initializes *Max_Attempts*, *Out_of_Time*, and *Max_Gen* by *false* and *Scoreboard* by zero (line 1).

MSGA executes its three loops as follows:

In each iteration of *the outer while loop* (lines 2 and 3), *MSGA* randomly selects untested *T* from *TestReq*. This loop is iterated until either all test requirements are marked or *Out_of_Time()* is *true*.

In *the middle while loop* (first-optimization stage from line 4 through 7), *MSGA* initializes or updates the sequences of called methods of the population. Each individual of the population consists of two parts 1) *names part*, which contains names of called methods and begins with a constructor of the *CUT* and 2) *parameters part*, which contains the values of the arguments.

If the current iteration is the first attempt for *T*, *MSGA* generates *popSize* sequences of called methods. The generation of each sequence is preformed in two steps: 1) a constructor is randomly selected from the available constructors of the *CUT* to construct an object on which a sequence of methods will issue, and 2) the methods, which contain the target *T* are appended this constructor. If the current iteration is not the first attempt for *T*, *MSGA* updates the *popSize* sequences of methods by applying a randomly selected operator of the genetic operators on *names part* of each individual in the population. Then, the arguments of the methods are randomly initialized by values according to their data types.

Generating an initial population needs initializing the sequences of called methods and then initializing their arguments. Initializing a sequence of called methods is performed by randomly selecting one of the constructors of *CUT* after that adding method/s contains the target to be tested. If these initial sequences have arguments, *MSGA* initializes them by random values. This population will pass, one individual at a time, to the inner while loop to optimize the parameters. This while loop is iterated until either the target is satisfied by a test or *Max_Gen()* returns true.

The inner while loop (second-optimization stage lines 8 to 14) finds the parameters (i.e., the values of the arguments of the called methods).

In line 9, *MSGA* uses a fitness function to evaluate the generated tests. The fitness function depends on the test criterion. For control-flow criteria, *MSGA* can use the fitness function of Pargas et al. [1]. For data-flow criteria, it can use the fitness function of Ghiduk et al. [11].

MSGA sorts *CurPopulation* according to the fitness values and selects tests to be parents of the new population (line 10) by a roulette wheel [12] with slots sized according to fitness. Then, the genetic operators are applied on the *parameters* of the selected parents to generate *NewPopulation* (line 11) see section 5.2.

MSGA uses each individual in *NewPopulation* to execute the *CUT* (line 12), and updates *Scoreboard* accordingly (line 13). If the target is satisfied by at least

```
Algorithm: MSGA
Input: CUT is a class to be tested and TestReq is a list of test requirements, which satisfies a coverage criterion c.
Output: Final is a set of tests for covering the given coverage criterion c.
Declarations:
    CurPopulation, NewPopulation: set of tests. Each test has the form {MethodsSeq, ParametersSeq}.
    MethodsSeq: sequence of methods to be called.
    ParametersSeq: sequence of parameters of the called methods.
    T: a test requirement for which a test case is to be generated
    Scoreboard: record of satisfied test requirements.
    Max_Attempts(): function that returns true when maximum number of attempts for a single T has been exceeded.
    Max_Gen(): function that returns true when maximum number of called methods for T has been exceeded.
    Out_of_Time(): function that returns true when the time limit is exceeded and false otherwise.
begin
 /*step1 Initialize and set up*/
[1]     Create and Initialize Scoreboard and the alarm functions
/* Step 2: Generate tests*/
[2]          while((some of TestReq are unmarked) and not(Out_of_Time()))do
[3]              Select unmarked T from TestReq.
[4]              while(T not marked and not(Max_ Attempts()))do
[5]                  if(first attempt) {Initialize the MethodsSeq of CurPopulation.}
[6]                  else {Apply one of the methods' operations on the current MethodsSeq.}
[7]                  Initialize randomly the ParametersSeq.
[8]                  while(T not marked and not Max_Gen() and nParameters>0)do
[9]                      Compute fitness values of CurPopulation.
[10]                     Select parents of NewPopulation from CurPopulation according to fitness values.
[11]                     Apply crossover and mutation on the parents to generate the NewPopulation.
[12]                     Execute Program on each member of NewPopulation
[13]                     Update Scoreboard and mark TestReq to reflect those test requirements that are satisfied.
[14]                 endwhile
[15]             endwhile
[16]         endwhile
   /* Step 3: Clean up and return*/
[17]  Final= tests that satisfy TestReq
[18]  Return(Tests, TestReq)
[19]  end
```

Figure 1. The overall algorithm of *MSGA*.

one test, the algorithm marked the target in *TestReq*, then the inner loop stop and subsequently the middle loop stop, and the algorithm attempts to find a new target. Otherwise, the *MSGA* replaces the *CurPopulation* with *NewPopulation* and attempts again satisfying the target. If the target consumed its all attempts without covering, the inner loop stop and the called methods (*names part*) of each individual is updated using the genetic operators see section 5.1. Then, the inner loop is repeated again. If the maximum attempts of the *names part* are exceeded, the middle loop stop and *MSGA* selects a new target. The inner two loops are repeated until either all test requirements are covered or time limit is exceeded, at which the outer loop stop. Finally, *MSGA* assigns all tests which satisfy *TestReq* to *Final* (line 17) and returns *Final* and *TestReq* (line 18).

IV. Representation, Encoding and Decoding

This section presents the strategies that are used by *MSGA* to represent, encode, and decode the tests.

A. Chromosomes Representation

The representation of tests for testing object-oriented programs is not just a sequence of values, but it is a sequence of methods and a sequence of values. Tonella [5] represented the tests as two concatenated parts separated by the character '@'. The first part is a sequence of methods invocations. The second part is the set of actual input values. Wappler and Lammermann [6] proposed a representation contains information about the called methods and their parameters and return values.

A shortcoming of the representation that is suggested by Tonella is that the state of the target object is changeable from generation to generation subsequently the fitness values are changeable and undependable. The suggested representation of Wappler and Lammermann caused in presence of inconvertible-individuals problems, which are, delimit the diversity of the population's individuals. Therefore, we suggested the following new representation to overcome these shortcomings.

Figure 2 shows the syntax of our represented chromosome. In this syntax, we use the character '&' as separator for the two main parts of the chromosome. The plus '+' character indicates that its two operands are concatenated. The square braces "[ ]" denote an array of elements, and the angle braces "< >" denote a single element. The curly braces "{}" used for optional repetition.

```
Chromosome = [methodsSeq] & [values]
[methodsSeq] = <constructor>+[called methods]
<constructor> = <single constructor randomly
                selected from the constructors of CUT>
[called methods]= [method {,method}]
[values]        = [value{,value}]
```

Figure 2: Syntax of chromosome.

We consider the chromosome as two separated parts, which encode into two separated array. The first part is a sequence of names of methods and the second part is a sequence of values. The sequence of names of methods consists of a constructor and a set of methods (may be empty) of *CUT*. The constructor is used to create an object of *CUT* on which the rest methods will issue. The set of methods is state changing method(s) or method(s) containing a tested target. The sequence of values is an array of values (may be empty) assigns to the required arguments for the sequence of methods. The type of values may be basic types (i.e., integer, real, character, boolean) or object type.

Consider the class *CoinBox* in Figure 3, which is taken from the work of Buy et al. [2]. Class *CoinBox* includes three integer variables (*totalQtrs*, *curQtrs*, and *allowVend*), one constructor (*CoinBox()*), and three methods (*returnQtrs()*, *addQtr()*, and *vend()*). We

augmented the method *vend* by an integer argument *x*. According to our representation, *CoinBox, addQtr, vend* & 4 is an example for the chromosome.

```
// CoinBox.cpp
1.  # include <iostream.h>
2.  # include "coinbox.h"
3.  CoinBox::CoinBox()
4.  { totalQtrs = 0;
5.    curQtrs = 0;
6.    allowVend = 0;
    }
7.  void CoinBox::returnQtrs()
8.  { curQtrs = 0; }
9.  void CoinBox::addQtr()
10. {curQtrs = curQtrs + 1;
11. if(curQtrs > 1)
12. {allowVend = 1;}
13. }
14. void CoinBox::vend (int x){
15.  if(x==5&&allowVend != 0)
16.  {totalQtrs = totalQtrs + curQtrs;
17.    curQtrs = 0;
18.    allowVend = 0;}}
```

```
// CoinBox.h
class CoinBox
{ // data members
private:
  int totalQtrs;
  int curQtrs;
  int allowVend;
public:
CoinBox();//constructor
 void returnQtrs();
 void addQtr();
 void vend(int);};
```

Figure 3. Test cluster for experiment.

B. Chromosome encoding and decoding

In this section, we describe the strategies for encoding and decoding the tests to chromosome.

- *Methods encoding and decoding:*

To encode the called method into chromosome, all methods and constructors of CUT are numbered in serial way such that numbering starts by one and constructors annotated before methods. For class CoinBox, we refer to constructor CoinBox by 1, method returnQtrs by 2, addQtr by 3 and vend by 4. Next, we replace each name of method or constructor in the names part by its number. Consequently, the first part of the chromosome (names part) is converted to a numerical array of length at least one and the value of any element of this array identifies a constructor or a method of CUT. This array contains integers in the interval from one to the total number of constructors and methods. Thus, the example chromosome CoinBox, addQtr, vend & 4 encodes to 1, 3, 4 & 4.

The decoding of names part is the inverse operation for the encoding. Each element in the encoded array indicates to an index of a method or a constructor of *CUT*. Therefore, we replace each index by the corresponding constructor or method.

- *Arguments encoding and decoding:*

Arguments data-types may be object-types or basic data-types such as integer, float, or character. We use a new binary array to represent the parameters of the called methods.

The length of the array depends on the required precision and the domain length for each parameter (which are assigned by the user). Arguments are encoded into binary array as follows:

Real or integer data-types: user assigns the precision and the domain length for each parameter. We use the method of Michalewicz [12] to map these data-types to binary number and vice versa.

Arguments of character type are converted into binary number according to its ASCII value.

Arguments of boolean type are encoded by one bit with value 1 (true) or 0 (false).

Arguments of object type, suppose one parameters is an object *B* created by a constructor of a class *C* which has two constructors and one method. We encode this object into binary number using its serial number. We are arranged the set of constructors of *C* and numbered them serially from one to three. Suppose that object *B* was created by constructor number two. So, this parameter encodes into the binary number '10'. Decoding converts the binary number to integer number. This integer number indicates to the index of the constructor in the arranged set of constructors of class *C*. If the integer value is *V* greater than the total number of constructors of class *C*. We subtract *nM* from *V* such that *V-nM* is less than or equal *M*, where *M* is the total number of constructors of class *C* and *n* is the smallest number such that $V\text{-}nM \leq M$. The final representation of the example chromosome will be 1, 3, 4 & 100. To execute this test, we call the following sequence of code:

```
CoinBox B;
B.addQtr();
B.vend(4);
```

V. Genetic Operators

We divide genetic operators into two categories: 1) methods genetic operators, and 2) arguments genetic operators.

A. Methods Genetic Operators

Methods genetic operators are applied repeatedly in the first-optimization stage on the *names part* of the chromosome.

We present prerequisites (conditions must be satisfied to apply an operator), postrequisites (modifications have to do after applying an operator) and stop constrained (conditions prevent applying an operator).

- *Constructor CHANGE operator:*

CHANGE operator replaces the current constructor with another randomly selected constructor from the same class.

$CHANGE([c_1,m_1,..,m_n])=[c_2,m_1,..,m_n]$ where c_2 is a randomly selected constructor from the *CUT*.

Example 1: Suppose a *CUT* has three constructors and three methods. The constructors are serially numbered from one to three and methods from four to six. Suppose that the called methods are *[1,4,6]*.

Then *CHANGE([1,4,6])= [3, 4, 5]* where *rand(1,3)* results in constructor *3*.

Prerequisites:

1. New and old constructors must be of same class.
2. Old constructor must not contain any target.
3. In data-flow testing, new constructors must not contain killing to the target.

Postrequisites:

a) Remove arguments of old constructor and insert arguments of the new one into chromosomes.
b) Replace traces of the old constructor by the new.

- *Method INSERT operator:*

INSERT operator randomly selects method of *CUT* and inserts it in a randomly selected position in the sequence of called methods.

INSERT ($[c_1, m_1, m_2, m_n]$) = $[c_1, m_1, \mathbf{m_4}, m_2, m_n]$. *INSERT* operator invokes *rand*(m_F, m_L) and *rand(2, length)*. Function *rand*(m_F, m_L) selects a method (e.g., m_4) randomly from the set of methods, m_F is the first method and m_L is the last method. Function *rand(2,length)* randomly selects an insertion position from 2 to *length* of called methods sequence (e.g., position *3*)

Example 2: for the given class in example 1. *INSERT([1,4, 6])* invokes *rand* function twice as follows: *rand(4, 6) = 5*, and *rand(2, 3) = 3*. Therefore, *INSERT([1, 4, 6])= [1, 4, **5**, 6]*.

Stop conditions:

The length of called methods sequence exceeds the maximum length determined by the user.

Prerequisites:

a) The new method must be related to the CUT.
b) The insertion position must be after the constructor and before the target method.
c) In data-flow testing, new method must not contain a killing to target.

Postrequisites:

a) Insert the parameters of the new method in the corresponding positions in the chromosomes.
b) Increase the length of methods sequence by 1.

- *Method REMOVE operator:*

REMOVE operator randomly selects and deletes a method from a sequence of called methods.

REMOVE($[c_1, m_1, m_2, \ldots, m_n]$) invokes *rand(2, length)* to select a deletion position from 2 to *length* of the sequence of called methods (e.g., location *2*). It deletes the method, which exists in the deletion position.

Then *REMOVE* ($[c_1, \mathbf{m_1}, m_2, \ldots, m_n]$)=$[c_1, m_2, \ldots, m_n]$.

Example 3: REMOVE([1, 4, 6] = [1, 4]. Where, *rand(2, 3)* generates *3*.

Stop conditions:

Length of called methods sequence is less than two.

Prerequisites:

a) The deleted method must not contain a target.
b) Deletion must be after position of constructor.

Postrequisites:

a) Delete the parameters of the deleted method.
b) Decrease the *length* of methods sequence by 1.

- *Method mMUTATION operator:*

mMUTATION operator replaces a randomly selected method from a sequence of called methods with another randomly selected method from the same class.

mMUTATION($[c_1, m_1, \ldots, m_n]$) = $[c_1, m_3, \ldots, m_n]$. This operator calls *rand(2, length)* and *rand*(m_F, m_L). *rand*(m_F, m_L) randomly selects method from the methods of *CUT* (e.g., m_3). *rand(2, length)* randomly selects replacement position from 2 to length of methods sequence (e.g., position *2)*. The length of methods sequence does not change.

*Example 4: mMUTATION([1,**4**,6])=[1,**5**,6]* where, *rand(2,length=3)* generates 2 and *rand*(m_F= *4*, m_L=*6*) generates 5. Then we replace method 4 by 5.

Prerequisites:

a) New and old methods must be of the same class.
b) Old method must not contain any target.
c) In data-flow testing, new method must not contain a killing to the target.

Postrequisites:

a) Delete parameters of the old method from the chromosome and insert parameters of the new.
b) Replace all traces of the old method by the new.

- *One-point mCROSSOVER operator:*

mCROSSOVER operator cuts at randomly selected position two sequences and swaps the cut parts.

mCROSSOVER($[c_1, m_1, m_3, \ldots, m_n]$,$[\mathbf{c_2}, \mathbf{m_2}, \mathbf{m_4}, \ldots, \mathbf{m_m}]$) invokes *rand(1,length))* to randomly generates a cutting position (e.g., 2). Therefore, *mCROSSOVER* swaps the two chromosomes at position 2 to get the new sequences of methods $[c_1, m_1, \mathbf{m_4}, \ldots, \mathbf{m_m}]$ and $[\mathbf{c_2}, \mathbf{m_2}, m_3, \ldots, m_n]$.

*Example 5: mCROSSOVER ([1,4,6],[**2,6,5**])=[1,**6, 5**], [**2,** 4, 6]*. Where *rand(1, 3)* generates 1.

Prerequisites:

a) The swapped methods must be of the same class.
b) Swapped methods must not contain any target.

Postrequisites:

a) Parameters values are cut and swapped.
b) Delete parameters values do not use any longer.
c) Insert any necessary parameters.

B. Arguments Genetic Operators

Arguments genetic operators are applied on the *parameters part* of the chromosome. This set of operators is applied repeatedly in the second-optimization stage of *MSGA*. We adapt the traditional genetic operators to use in *MSGA*.

- *Parameters pCROSSOVER operator:*

During *pCROSSOVER*, two parents (chromosomes) exchange sub string information (genetic material) at a random position in the parameters sequence to produce two new strings. If the two parameters sequence are not of the same length then we take the smallest length as length of the two parameters sequences or when the position of swapping exceed the smallest length no swapping occur.

- *Parameters pMUTATION operator:*

pMUTATION operator is performed on a bit-by-bit basis. *pMUTATION* always operates after the *pCROSSOVER*, and flips a randomly selected bit from 0 to 1 or vice versa.

VI. FITNESS FUNCTION

MSGA uses a new fitness function to evaluate the generated tests. This function utilizes the concepts of the dominance relations between nodes of the class's control flow graph. *MSGA* evaluates each test by executing the sequence of methods of *CUT* with its parameters, and recording the set of executed nodes (statements) of the *CUT*. *MSGA* computes also the dominance paths of the target structure for the given test criterion (e.g., def and use nodes for all def-use criterion, i.e, *dom_d* and *dom_u*).

The fitness function is the summation of two terms. The first term is the ratio of the number of covered nodes of the dominance path of the def node (*nCoverDom_d*) to the total number of nodes of the dominance path of the def node (*nDom_d*). The second term is the ratio of the number of covered nodes of the dominance path of the use node (*nCoverDom_u*) to the total number of nodes of the dominance path of the use node (*nDom_u*). The fitness value $ft(v_i)$ for each chromosome v_i *(i = 1, ..., popSize)* is calculated by the following formula:

$$ft(v_i) = \frac{1}{2} \times \frac{nCoverDom\_d}{nDom\_d} + \frac{1}{2} \times \frac{nCoverDom\_u}{nDom\_u}.$$

The fitness value is the only feedback from the problem for the GA. A test that is represented by the chromosome v_i is optimal if its fitness value $ft(v_i) = 1$.

VII. Prototype Implementation

In order to determine the applicability of *MSGA*, an object-oriented prototype in C++ that provides the data-flow coverage of object-oriented software was implemented. A case study was performed by this prototype involving class *CoinBox* in Figure 3. This section describes the implementation of the prototype and the next section gives the case study.

Figure 4 gives the overall diagram of the architecture of the implementation of the algorithm *MSGA*, called *OOTGen*.

Our prototype *OOTGen* consists of two major modules: *Class Analyzer* and *Test Generator*.

The *Class Analyzer* performs the following tasks on the class under test (*CUT*):

- Parsing the source code: reads the source code of the *CUT* and classifies each statement (i.e., determining its type).
- Reformatting source code: reformats some statements of the source code to facilitate building the class control-flow graph of *CUT*.
- Analyzing source code: analysis the source code of the *CUT* to collect information about the components of it (e.g., instance variables and methods) and all used variables. The collected information about the methods is their categories (constructor or destructor or ordinary method), their returned data type, their associated classes, their access levels, and so on. The information collected about variable is name, data type, scope (global or local), fixed or dynamic, temporary or parameter, the set of actions occurred on this variable (def or use action), and so on.
- Building *CCFG*: builds the class control-flow graph using the technique described by Harrold and Rothermel in [13].
- Computing *Def-Uses*: computes all definition-uses associations in the class under test using the approach of Pande, Landi, and Ryder described in [14]. The prototype finds reaching definitions in the case of no pointers. The case of pointers is quite similar.
- Building *Dom-Trees*: builds the dominators trees of the class' methods. It uses for this task the algorithm presented by Lengauer and Trajan in [15]. This algorithm is repeated for each method of the class under test to build its dominator tree.
- Instrument: probes are instrumented in the class' methods to record the executed statements of them, which used in the calculation of fitness function.

Class control-flow graph of *CUT*, dominators trees of the class' methods, all definition-uses associations, and instrumented version of the class under test are passed to the next module.

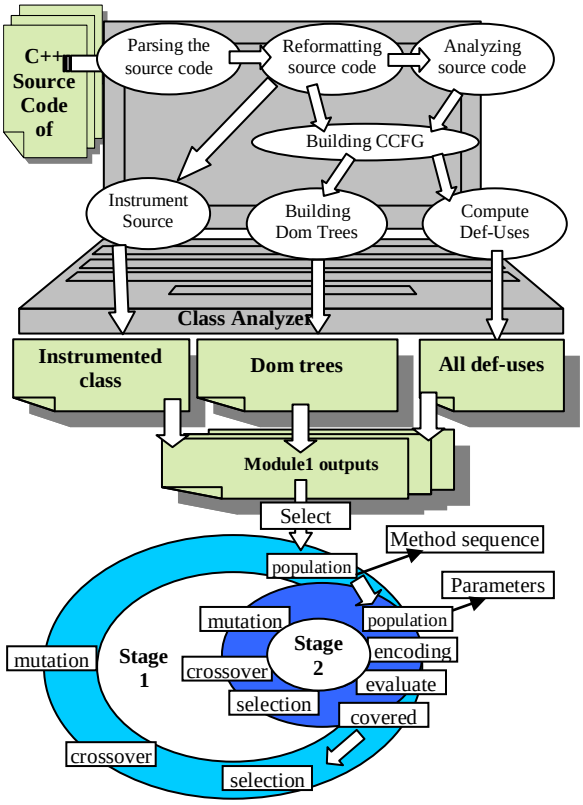

Figure 4: The overall diagram of the prototype *ooTGen*.

The *Test Generator* performs the following tasks on definition-uses associations of the class under test:

- Selecting uncovered def-use association to be covered.
- Creating and evolving populations to cover the selected def-use.
- Reporting the process to find the coverage percentage.

OOTGen handles automatic test data generation for data-flow testing of the object-oriented programs for three level of interactions 1) intra-method, 2) inter-method, 3) intra-class. It can handle the data-flow in the presence of pointers. The *OOTGen* can be extended to handle the inter-class testing.

VIII. Case Study

To determine the practicability of the proposed genetic algorithm *MSGA* and the prototype, a case study was

performed by applying the implemented prototype on class *CoinBox*, which is showed in Figure 3.

The outputs of the first module of our prototype are *CCFG*, an instrumented version of the test cluster in Figure 3 and the set of def-use associations. Table 1 shows the *CCFG's* edges of the test cluster.

From Table 1 we can conclude that edges from *e5* through *e11* represent the control-flow graph of the constructor *CoinBox*, edges from *e12* through *e16* represent the control-flow graph of function *returnQtrs*, edges from *e17* through *e26* represent the control-flow graph of function *addQtr*, edges from *e2* through *e37* represent the control-flow graph of function *vend*. The other edges connect the four control-flow graph together [13].

The set of def-use associations depends on the number of frames, the number of called method in the frame, and which methods will be called. For example: suppose frame1 contains four methods *CoinBox, returnQtrs, vend*, and *addQtr*, then the set of def-use associations for frame1 is shown in table 2. If frame2 contains 3 methods *CoinBox, returnQtrs*, and *vend* then the set of its def-use associations is shown in table 3. Table 2 contains 12 def-use associations and Table 3 includes 8 def-use associations. For example *(18, 23, allowVend)* is a *dua* where variable *allowVend* is defined in statement 18 and used in statement 23.

Figure 5 gives a sample for the output report of the genetic module (Test Generator). This report shows the parameters of the genetic algorithm, encoding of the methods, and the generations to cover the def-use association (27, 23, curQtrs).

From Figure 5 we can conclude the following:

- The parameters of *MSGA* genetic algorithm are population Size = 4, maximum number of methods in the individual = 7, maximum number of parameters generation = 5, parameters crossover probability = 0.50, and parameters mutation probability = 0.15. Therefore, stage 1 will stop if the length of a sequence of method calls is 7 or the target test requirement is covered. Stage 2 will stop if the set of parameters of the sequence of method calls is generated 5 times or the target is covered.
- Stage 1 generates or updates sequences of method calls. For example: in generation 1, it generates the sequence 1, 4, 4, (i.e., *CoinBox(), vend(int), vend(int)*). We can execute this sequence by using the constructor to construct an object of the class *CoinBox* and invoke method *vend* on it as the following code:

```
CoinBox CB;
CB.vend(6);
CB.vend(2);
```

- *MSGA* initializes each sequence by a randomly selected constructor and the methods, which contain the target test requirement. For example: 1, 4, 4 where 1 is the constructor and 4 is method *vend* which contains the definition and use nodes of the *dua* (27, 23, curQtrs).
- Stage 2 performs the traditional genetic algorithm to generate the required parameters.
- The generated test is 1, 3, 3, 4, 4 @ 5, 2 with fitness value = 1 which can execute as follows.

```
CoinBox CB;
CB.addQtr();
CB.addQtr();
CB.vend(5);
CB.vend(2);
```

TABLE 1.
THE SET OF EDGES OF CCFG

e1	(frame_entry,frame_loop)	e24	(addQtr_17,addQtr_18)
e2	(frame_loop,frame_call)	e25	(addQtr_18,addQtr_19)
e3	(frame_loop,frame_exit)	e26	(addQtr_19,addQtr_20)
e4	(frame_return,frame_loop)	e27	(entry_vend_21,vend_21)
e5	(entry_CoinBox_3,CoinBox_3)	e28	(vend_29,exit_vend_29)
e6	(CoinBox_8,exit_CoinBox_8)	e29	(vend_21,vend_22)
e7	(CoinBox_3,CoinBox_4)	e30	(vend_22,vend_23)
e8	(CoinBox_4,CoinBox_5)	e31	(vend_23,vend_24)
e9	(CoinBox_5,CoinBox_6)	e32	(vend_23,vend_29)
e10	(CoinBox_6,CoinBox_7)	e33	(vend_24,vend_25)
e11	(CoinBox_7,CoinBox_8)	e34	(vend_25,vend_26)
e12	(entry_returnQtrs_9,returnQtrs_9)	e35	(vend_26,vend_27)
e13	(returnQtrs_12,exit_returnQtrs_12)	e36	(vend_27,vend_28)
e14	(returnQtrs_9,returnQtrs_10)	e37	(vend_28,vend_29)
e15	(returnQtrs_10,returnQtrs_11)	e38	(frame_call,entry_CoinBox_3)
e16	(returnQtrs_11,returnQtrs_12)	e39	(exit_CoinBox_8,frame_return)
e17	(entry_addQtr_13,addQtr_13)	e40	(frame_call,entry_returnQtrs_9)
e18	(addQtr_20,exit_addQtr_20)	e41	(exit_returnQtrs_12,frame_return)
e19	(addQtr_13,addQtr_14)	e42	(frame_call,entry_addQtr_13)
e20	(addQtr_14,addQtr_15)	e43	(exit_addQtr_20,frame_return)
e21	(addQtr_15,addQtr_16)	e44	(frame_call,entry_vend_21)
e22	(addQtr_16,addQtr_17)	e45	(exit_vend_29,frame_return)
e23	(addQtr_16,addQtr_20)		

TABLE 2.
DEF-USE ASSOCIATIONS OF FRAME1

5	25	totalQtrs
6	15	curQtrs
6	25	curQtrs
7	23	allowVend
15	16	curQtrs
15	15	curQtrs
15	25	curQtrs
18	23	allowVend
25	25	totalQtrs
26	15	curQtrs
26	25	curQtrs
27	23	allowVend

TABLE 3.
DEF-USE ASSOCIATIONS OF FRAME2

5	25	totalQtrs
6	25	curQtrs
7	23	allowVend
7	23	allowVend
11	25	curQtrs
15	16	curQtrs
25	25	totalQtrs
26	25	curQtrs

More experiments are required for determining the efficiency and effectiveness of the proposed algorithm and prototype.

IX. CONCLUSIONS AND FUTURE WORK

Testing the object-oriented programs has been addressed from different viewpoint by many researches, most of them concerned with the problem related to the generation of sequences of method calls to satisfy a set of test requirements. A few researches address the specific problem of test-data generation for classes.

We presented in this paper a multi-stage genetic algorithm *(MSGA)*. *MSGA* contains two optimization stages. *MSGA* has the ability to generate sequences of method calls and data at the same time. We used *MSGA* for generating tests for the object-oriented programs. In the paper, we introduced a new chromosome representation to be used by *MSGA*. The strategies for encoding and decoding the tests to chromosomes are discussed in the paper. In addition, a set of genetic operators to evolve the current population of tests are presented.

The Class Under Test(CUT) is : coinbox

Population Size: 4
Maximum Number of Methods in the individual: 7
Maximum Number of Parameters Generation: 5
Parameters Crossover Probability: 0.80
Parameters Mutation Probability: 0.15

Method	Category	Encode
CoinBox	constructor	1
returnQtrs	method	2
addQtr	method	3
vend	method	4

** **Note**:
** individual 1,2,3 & 10110111000 is
** methods' encoding,&(separator),parameters' encoding
**** GA Started ****
==========
For the definition use accusation (27, 23, curQtrs)
==========
*** Generation 1: Initial Population
stage1:
Initializing the methods sequences:
* Individual_1 = 1, 4, 4 &---
* Individual_2 = 1, 4, 4 & ---
* Individual_3 = 1, 4, 4 & ---
* Individual_4 = 1, 4, 4 & ---
stage2:
1.Initializing parameters
* Individual_1 = 1, 4, 4 & 6, 2 = 1, 4, 4 & 110010
* Individual_2 = 1, 4, 4 & 1, 3 = 1, 4, 4 & 001011
* Individual_3 = 1, 4, 4 & 3, 4 = 1, 4, 4 & 011100
* Individual_4 = 1, 4, 4 & 4, 5 = 1, 4, 4 & 100101
2. Evaluation of the Population
* FT(v1)=1/2(3/7)+1/2(3/3)=0.72 * FT(v2)= 0.72
* FT(v3)= 0.72 * FT(v4)= 0.72
where:
* Traversed Path is: 3 4 5 6 7 8 , 21 22 23 29 , 21 22 23 29
* dom. path of def node 27 is: 21 22 23 24 25 26 27
* dom. path of use node 23 is: 21 22 23
(27, 23, curQtrs) Not covered
3- Selection: Individual_2 & Individual_3 are the parent
3.1crossover
* Individual_2 = 1, 4, 4 & **0111**11
* Individual_3 = 1, 4, 4 & **0010**00
3.2 mutation
* Individual_2 = 1, 4, 4 & **0111***0*1
* Individual_3 = 1, 4, 4 & ***1*010**00
3.3. the new population:
* Individual_1 = 1, 4, 4 & 110010
* Individual_2 = 1, 4, 4 & **0111***0*1
* Individual_3 = 1, 4, 4 & ***1*010**00
* Individual_4 = 1, 4, 4 & 100101
Return to the evaluation (step 2 in stage 2) and so on

*** Generation 2:
stage1:
Updating the methods sequences by applying *mINSERT*:
* Individual_1 = 1, 3, 4, 4 &---
* Individual_2 = 1, 3, 4, 4 & ---
* Individual_3 = 1, 2, 4, 4 & ---
* Individual_4 = 1, 3, 4, 4 & ---

The same steps as Generation1:

*** Generation 3:
stage1:
Updating the methods sequences by applying *mINSERT*:
* Individual_1 = 1, 3, **3,** 4, 4 &---
* Individual_2 = 1, 3, **2,** 4, 4 & ---
* Individual_3 = 1, 2, **3,** 4, 4 & ---
* Individual_4 = 1, 3, **4,** 4, 4 & ---
stage2:
1.Initializing parameters
* Individual_1 = 1, 3, 3, 4, 4 & 6, 2 = 1, 4, 4 & 110010
* Individual_2 = 1, 3, 2, 4, 4 & 1, 3 = 1, 4, 4 & 001011
* Individual_3 = 1, 2, 3, 4, 4 & 3, 4 = 1, 4, 4 & 011100
* Individual_4=1, 3, 4, 4, 4 & 3,4,5= 1,3,4,4,4& 011100101
2. Evaluation of the Population
* FT(v1)=1/2(3/7)+1/2(3/3)=0.72 * FT(v2)= 0.72
* FT(v3)= 0.72 * FT(v4)= 0.72
where:
* Traversed Path is: 3 4 5 6 7 8**,** 13 14 15 16 20, 13 14 15 16 17 18 19 20, 21 22 23 29, 21 22 23 29
(27, 23, curQtrs) Not covered
3- Selection: Individual_1 & Individual_2 are the parent
3.1crossover
* Individual_1 = 1, 3, 3, 4, 4 & 6, 2 = 1, 4, 4 & **001**010
* Individual_2 = 1, 3, 2, 4, 4 & 1, 3 = 1, 4, 4 & **110**011
3.2 mutation
* Individual_1 = 1, 3, 3, 4, 4 & ***1*01**010
* Individual_2 = 1, 3, 2, 4, 4 & ***0*10**011
3.3. the new population:
* Individual_1 = 1, 3, 3, 4, 4 & ***1*01**010 = 1, 3, 3, 4, 4 & 5, 2
* Individual_2 = 1, 3, 2, 4, 4 & ***0*10**11= 1, 3, 2, 4, 4 & 2, 3
* Individual_3 = 1, 2, 3, 4, 4 & 011100= 1, 2, 3, 4, 4 & 3, 4
* Individual_4=1, 3, 4, 4, 4 & 011100101=1, 3,4, 4,4&3,4,5
Return to the evaluation (step 2 in stage 2)
2. Evaluation of the Population
* FT(v1)=1/2(7/7)+1/2(3/3)=1
where:
* Traversed Path is: 3 4 5 6 7 8**,** 13 14 15 16 20, 13 14 15 16 17 18 19 20, 21 22 23 24 25 26 **27** 28 29, 21 22 **23** 29
(27, 23, curQtrs) covered
**********Final Report **********
** Best Fitness is: 1.000
** No. of Generations = 3
** The Test Requirement (27, 23, curQtrs) is satisfied and
** The Generated Test is: 1, 3, 3, 4, 4 & 5, 2
** see individual 1 and its evaluation.

Figure 5. Sample of the output report of the Prototype.

We also presented the prerequisites, postrequisites, and stop-conditions, which are necessary to apply the new genetic operators.

A prototype has been development for using the proposed algorithm to generate tests for data-flow testing of the object-oriented software. In addition, a case study has been conducted to evaluate the presented algorithm. The results of the case study are encouraged and it showed that *MSGA* represents a potentially powerful approach in the area of automatic test generation for object-oriented testing.

Our future work will focus on conducting an empirical study using the implemented prototype to evaluate the effectiveness and the efficiency of our technique and compare it with the other genetic algorithms based techniques.

ACKNOWLEDGMENT

The author would like to thank the reviewers of the 2nd International Conference on Computer Science and its Applications (CSA 2009) for their valuable comments and suggestions to improve this paper.

REFERENCES

[1] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test Data Generation Using Genetic Algorithms" Journal of Software Testing, Verifications and Reliability, vol. 9, pp. 263-282, 1999.

[2] U. Buy, A. Orso, and Pezzè, "Automated Testing of classes," In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2000), August 2000.

[3] V. Martena, A. Orso, and Pezzè, "Interclass Testing of Object Oriented Software," In Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS2002), 2002.
[4] http://ctp.di.fct.unl.pt/~ja/wituml/polo.PDF.
[5] P. Tonella, "Evolutionary testing of classes," In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2000), pp. 119-128, 2004.
[6] S. Wappler and F. Lammermann, "Using evolutionary algorithms for the unit testing of object-oriented software," In Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO 2005), pp. 1053-1060, 2005.
[7] Y. Cheon, M. Y. Kim, and A. Perumandla, "A Complete Automation of Unit Testing for Java Programs," Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), pp. 290-295, 2005.
[8] Y. Cheon and M. Kim, "A Fitness Function for Modular Evolutionary Testing of Object-Oriented Programs" In Genetic and Evolutionary Computation Conference, pp. 1952-1954, 2006.
[9] M. Y. Kim and Y. Cheon, "A Fitness Function to Find Feasible Sequences of Method Calls for Evolutionary Testing of Object-Oriented Programs," To appear in International Conference on Software Testing, Verification, and Validation, Norway, April 9-11, 2008.
[10] J. Holland, Adaptation in Natural and Artificial Systems, ISBN 0 472 08460 7. University of Michigan Press, Ann Arbor, MI, 1975.
[11] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis, "Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage" Proceeding of 14th Asia-Pacific Software Engineering Conference (APSEC 2007), pp. 41-48, 2007.
[12] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3rd Edition, Springer, 1999.
[13] M. J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In 2nd ACM-SIGSOFT Symposium on the foundations of software engineering, pages 154-163, New Orleans, LA(USA), December 1994.
[14] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations in C programs. IEEE Transactions on Software Engineering, 20(5): 385-403, May 1994.
[15] T. Lengauer and R. E. Trajan, "A Fast Algorithm for Finding Dominators in a Flowgraph" ACM Transactions on programming Languages and Systems, vol. 1, pp. 121-141, 1979.

Ahmed S. Ghiduk is an assistant professor at Beni-Suef University, Egypt. He received the BSc degree from Cairo University, Egypt, in 1994, the MSc degree from Minia University, Egypt, in 2001, and a Ph.D. from Beni-Suef University, Egypt in joint with College of Computing, Georgia Institute of Technology, USA, in 2007. His research interests include software engineering especially search-based software testing, genetic algorithms, and ant colony. Currently, Ahmed S. Ghiduk is an assistant professor at College of Computers and Information Systems, Taif University, Saudi Arabia. One can connect Ahmed S. Ghiduk on asaghiduk@yahoo.com or gamil.com.