# Evolving a Test Oracle in Black-Box Testing

Farn Wang[1,2], Jung-Hsuan Wu[1], Chung-Hao Huang[1,2], and Kai-Hsiang Chang[1]

[1] Dept. of Electrical Engineering, [2] Graduate Institute of Electronic Engineering
National Taiwan University, Taiwan, ROC
farn@cc.ee.ntu.edu.tw

**Abstract.** Software testing is an important and expensive activity to the software industry, with testing accounting for over 50% of the cost of software. To ease this problem, test automation is very critical to the process of software testing. One important issue in this automation is to automatically determine whether a program under test (PUT) responds the correct(expected) output for an arbitrary input. In this paper, we model PUTs in black-box way, i.e. processing and responding a list of numbers, and design input/output list relation language(IOLRL) to formally describe the relations between the input and output lists. Given several labelled test cases(test verdicts are set), we use genetic programming to evolve the most distinguishing relations of these test cases in IOLRL and encode the test cases into bit patterns to build a classifier with support vector machine as the constructed test oracle. This classifier can be used to automatically verify if a program output list is the expected one in processing a program input list. The main contribution of this work are the designed IOLRL and the approach to construct test oracle with evolve relations in IOLRL. The experiments show the constructed test oracle has good performance even when few labelled test cases are supplied.

**Keywords:** test oracle, input/output list relation language, genetic programming, support vector machine, black-box testing.

## 1 Introduction

Software testing is an important and expensive activity to the software industry, with testing accounting for over 50% of the cost of software [6]. To ease this situation, test automation is very critical to the process of software testing [4,5]. One important issue in the test automation is an automated test oracle. A test oracle is a mechanism to determine whether an output is the expected output of a program under test (PUT) against an input. Once the test oracle can be automated, the correctness of program outputs can be verified without human intervention. Therefore the efficiency of software testing process can be enhanced.

In this paper, we investigate the problem of test oracle automation in black-box approach. The behavior of a PUT is modelled to process a list of numbers and respond a list of numbers. We design input/output list relation language(IOLRL) to describe the relations between the input and output lists and propose an approach to construct a test oracle with labelled test cases by incorporating the techniques of genetic programming and support vector machine. With genetic programming, relations in IOLRL are evolved to describe the behaviors of a PUT and used to encode the labelled test cases into bit patterns. With these bit patterns, an SVM classifier can be trained and used to verify if an output list is the expected one of a PUT in processing an input list. Hence an automated test oracle is constructed.

The remainder of this paper is structured as follows: Section 2 reviews the related work. Section 3 reviews the background materials of software testing, genetic programming and support vector machine. Section 4 introduces the input/output list relation language(IOLRL). Section 5 presents the procedures of our proposed approach. Section 6 reports the experiments. Section 7 is the conclusion.

## 2 Related Work

In the literature, there have been several researches addressing on the issue of test oracle automation. Brown, et al, [1] use white-box approach to automatically generate a test oracle from formally specified requirements. Peters, et al, [8] constructs a test oracle from program documentations. Chen, et al, [3] constructs a test oracle with metamorphic testing which needs to identify the PUT properties first. In contrast to these approaches, out approach is black-box based and constructs test oracle from test cases which are generally easier for users to supply.

The approach proposed by Vanmali, et al, [10] also construct a test oracle from test cases with neural network technique. Compare to their approach, our approach emphasizes on the relation of the input and output data of a PUT and automatically figures out the relations to infer the PUT's behavior with these data. The experiments in section 6 shows that our approach performs better with fewer supplied test cases. In some cases with large amount of supplied test cases, our approach can also performs better.

## 3 Background

In this section, we first introduce the model of a PUT in this paper and review the definitions of test oracle and test case. Then the background knowledge about genetic programming and support vector machine are introduced.

### 3.1 Definitions

The considered PUTs in this paper are abstracted into programs that take sequences of integers as their input and respond sequences of integers as their output. We also assume the behaviors of such PUTs can only be observed from the input lists and output lists. Therefore we have the following definitions. For convenience, we use $\mathbb{N}$ to denote the non-negative numbers and $\mathbb{Z}$ to denote the integer numbers.

**Definition 1.** ***(Input list)*** *An* input list $I : \mathbb{N} \to (\mathbb{N} \to \mathbb{Z})$ *is a finite sequence such that each element of I is a finite sequence of integers.* ∎

Two input lists $I$, $I'$ are equal if $I(i)(j) = I'(i)(j)$, $0 \leq i, j$.

**Definition 2.** ***(Output list)*** *An* output list $O : \mathbb{N} \to (\mathbb{N} \to \mathbb{Z})$ *is a finite sequence such that each element of O is a finite sequence of integers.* ∎

Two output lists $O$, $O'$ are equal if $O(i)(j) = O'(i)(j)$, $0 \leq i, j$.

**Definition 3.** ***(PUT)*** *A* program under test *(PUT)* $\pi : (\mathbb{N} \to (\mathbb{N} \to \mathbb{Z})) \to (\mathbb{N} \to (\mathbb{N} \to \mathbb{Z}))$ *is a partial function that maps an input list to an output list.* ∎

Note that we restrict the size of sequences of input and output lists to be finite. This restriction is natural since computer systems are always restricted to finite sets in practice [8]. We use example 1 to explain the formal modelling of the considered PUT.

*Example 1.* Consider a list-processing program that searches for an integer out of a sequence of numbers. If the sequence contains this integer, the index of this integer in the sequence will be returned, otherwise −1 will be returned. With the above formal notation, let $I = \{\{3, 10, 36, 5\}, \{5\}\}$ be an input list of this program. If this program responds $O = \{\{3\}\}$, then $\{I \to O\}$ is one of the mappings of this program.

**Definition 4.** ***(Test oracle)*** *A* test oracle *of a PUT is a total function* $\omega : (\mathbb{N} \to (\mathbb{N} \to \mathbb{Z}), \mathbb{N} \to (\mathbb{N} \to \mathbb{Z})) \to \{pass, fail\}$. *Given an input list I and an output list O,* $\omega(I, O) = pass$ *if O is the correct output of this PUT in processing I. Otherwise* $\omega(I, O) = fail$. *Ideally, the mapping of a test oracle is always correct.* ∎

Intuitively, a test case is a description to describe the output list of a PUT in processing an input list. This test case is also labelled with a *verdict* bit to indicate whether the described output list is the correct response. We formally define a test case as follows.

**Definition 5.** ***(Test case)*** *A* test case $t = (I, O, v)$ *is a triple where*

- *I is an input list,*
- *O is an output list, and*
- $v \in \{pass, fail\}$ *is a verdict bit to label this test case.* $v = pass$ *indicates O is a correct response against I. Otherwise O is not a correct response.*

*For convenience, we denote* $t^I$ *the input list of t,* $t^O$ *the output list of t and* $t^v$ *the verdict bit of t.* ∎

For example, with the PUT illustrated in example 1, a test case $(\{\{3, 10, 36, 5\}, \{5\}\}, \{\{-1\}\}, fail)$ describes "−1" is not the expected output list in the searching for 5 out of the sequence $\{3, 10, 36, 5\}$. Another test case $(\{\{3, 10, 36, 5\}, \{5\}\}, \{\{3\}\}, pass)$ describes "3" is expected in the searching for 5.

### 3.2 Genetic Programming

*Genetic programming* (GP) [7,12,9] is an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance. It follows the Darwin's theory that transforms populations of *individuals* into new populations of individuals generation by generation. GP, like nature, is a random process, and it can never guarantee results. GP's essential randomness, however, can lead it to escape traps which deterministic methods may be captured by. Like nature, GP has been very successful at evolving novel and unexpected ways of solving problems [9].

GP starts with a population of randomly generated individuals. Each individual is scored with a *fitness function*. Then, these individuals will *reproduce* itself, *crossover* with another individual and *mutate* into a new population. Individuals with higher scores will perform these operations more frequently. Then the average fitness value of the new evolved population will, almost certainly, be better than the average of the previous population. The individuals in the new population are evaluated again according to the fitness function. Operations such as reproduction, crossover and mutation are again applied to this new population. After sufficient generations, the best individual in the last population would be an excellent solution to the posed problem.

The key components of GP are as follows:

- A *syntax* to represent individuals. In GP, individuals are usually expressed as syntax trees.
- A *fitness function* to evaluate how well each individual can survive in the environment.
- A method to initialize a population. The *full*, *grow* and *ramped half-and-half* methods are common in this regard. In both the full and grow methods, the initial individuals, i.e. syntax trees, are generated so that they do not exceed a user specified maximum depth and the tree nodes are randomly taken. The difference is the syntax trees generated by the full method are complete and the syntax trees generated by the grow method are not necessary to be complete. In the ramped half-and-half method, it simply generates half of the population by the full method and half of the population by the grow method to help ensure the generated trees having a variety of sizes and shapes.
- A genetic operator to probabilistically select better individuals based on fitness. The most commonly employed method is *tournament selection*. In tournament selection, a number of individuals are chosen at random from the population. Then these chosen individuals are compared with each other and the top $n$ are chosen to be the parents ($n$ is decided according to the number of parents it needed in the successive genetic operator). Note that the number of randomly chosen individuals should not be too large in order to keep the diversity of individuals in each generation.
- Genetic operators to generate offspring individuals from the parent individuals. The common used operator are *subtree crossover*, *subtree mutation* and

*reproduction*. Given two parent individuals, subtree crossover randomly selects a node in each parent individual and swaps the entire subtrees. Given a parent individual, the subtree mutation randomly selects a node and substitutes the subtree rooted there with a randomly generated subtree. Given a parent individual, the reproduction operator simply inserts a copy of it into the next generation.
- A termination criterion to stop evolving new population. Usually the termination criterion is a number specified by user to determine the maximum generation to evolve.

Due to the page limit, we refer the interesting readers to [7,12,9] for the details of these components.

The syntax should have an important property call *Sufficiency* which means it is possible to express a solution to the problem using the syntax. Unfortunately, sufficiency can be guaranteed only for those problems where theory, or experience with other methods, tells us that a solution can be obtained with the syntax [9]. When a syntax is insufficiency, GP can only develop individuals that approximate the desired one. However, in many cases such an approximation can be very close and good enough for the user's purpose.

For GP to work effectively, most non-terminal nodes in a syntax tree are also required to have another important *closure* property [7] which can be further broken down into the *type consistency* and *evaluation safety* properties. Since the crossover operator can mix and join syntax nodes arbitrarily, it is necessary that any subtree can be used in any of the argument positions for every non-terminal node. Therefore it is common to require all the non-terminal nodes to be type consistent, i.e. they all return values of the same type, and each of their arguments also have this type. An alternative to require the type consistency is to extend the crossover and mutation operators to explicitly make use of the type information so that the offsprings they produce do not contain illegal type mismatches. The other component of closure is evaluation safety. Evaluation safety is required because individuals may fail in an execution. In order to preserve the evaluation safety property, modifications of normal behaviors are common. An alternative is to reduce the fitness of individuals that have such errors. In section 5, we will describe how to handle the closure and evaluation safety properties in our application.

### 3.3 Support Vector Machine

The support vector machine (SVM) [11,13] is a technique motivated by statistical learning theory and has been extensively used to solve many classification problems. It is a method to learn functions from a set of labelled training data points. The idea is to separate two classes of the labelled training data points with a decision boundary which maximizes the margin between them. In this paper, we will only consider a binary classification task with a set of $n$ labelled training data points: $(\boldsymbol{x_i}, y_i)$, where $\boldsymbol{x_i} \in \mathbb{R}^N$ and $y_i \in \{+1, -1\}$ indicates its corresponding class. The SVM technique tries to separate the training data points with a hyperplane ${\boldsymbol{x_i}}^T \boldsymbol{w} + b = 0$ with the maximum margin where $\boldsymbol{w} \in \mathbb{R}^N$ is normal to the hyperplane and $b \in \mathbb{R}$ is a bias. The *margin* of such hyperplane is defined by the sum of the shortest distance from the hyperplane to the closest positive training data points and the closest negative training data points. Due to the page limit, we refer the interesting readers to [11,13] for the details.

One of the advantages of SVM is that the learning task is insensitive to the relative numbers of training data points in positive and negative classes. Another advantage is SVM can achieve a lower false alarm rate. Most learning algorithm based on Empirical Risk Minimization will tend to classify only the positive class correctly to minimize the error over data set. Since SVM aims at minimizing a bound on the generalization error of a model in high dimensional space, so called Structural Risk Minimization, rather than minimizing the error over data set, the training data points that are far behind the hyperplane will not change the support vectors. Therefore, SVM can achieve a lower false alarm rate.

## 4 Input/Output List Relation Language(IOLRL)

In this section, we present the *input/output list relation language* (IOLRL) to formally describe the relations between the input and output lists of test cases. The syntax of the language IOLRL in BNF form is as follows:

$$
\begin{aligned}
< \mathit{Rule} > \; &:= < \mathit{Rule} >< \mathit{BOP} >< \mathit{Rule} > \\
&\mid \neg \; ( < \mathit{Rule} > ) \\
&\mid \exists < \mathit{qvar} > ( < \mathit{Rule} > ) \\
&\mid \forall < \mathit{qvar} > ( < \mathit{Rule} > ) \\
&\mid < \mathit{Elem} >< \mathit{ROP} >< \mathit{Elem} > \\
&\mid < \mathit{Elem} >< \mathit{InOP} >< \mathit{Elems} >; \\
< \mathit{Elems} > \; &:= < \mathit{List} > [ < \mathit{nv} > ]; \\
< \mathit{Elem} > \; &:= < \mathit{Math} > ( < \mathit{List} > [ < \mathit{nv} > ] ) \\
&\mid < \mathit{List} > [ < \mathit{nv} > ][ < \mathit{nv} > ] \\
&\mid < \mathit{nv} > + < \mathit{nv} > \\
&\mid < \mathit{nv} > - < \mathit{nv} > \\
&\mid < \mathit{nv} >; \\
< \mathit{Math} > \; &:= \mathit{max} \mid \mathit{min} \mid \mathit{average} \mid \mathit{sum}; \\
< \mathit{BOP} > \; &:= \wedge \mid \vee; \\
< \mathit{ROP} > \; &:= \; = \mid \neq \mid \leq \mid > \mid < \mid \geq; \\
< \mathit{InOP} > \; &:= \; \in \; ; \\
< \mathit{qvar} > \; &:= < \mathit{var} > ; \\
< \mathit{nv} > \; &:= < \mathit{num} > \mid < \mathit{var} >; \\
< \mathit{num} > \; &:= 0 \mid 1 \mid 2 \mid \ldots ; \\
< \mathit{var} > \; &:= i \mid j \mid \ldots ; \\
< \mathit{List} > \; &:= \mathit{InputList} \mid \mathit{OutputList};
\end{aligned}
$$

Note that for simplicity, the syntax rule $< \mathit{num} >$ is limited from 0 to 9 and the syntax rule $< \mathit{var} >$ is limited to $i$ and $j$ in the demonstration experiments in section 6.

The syntax of IOLRL are self-explained of its semantics. We use three example relations to explain the IOLRL semantics.

- $\exists i(\forall j(OutputList[0][j] \in InputList[i]))$. This relation describes there exists a sequence $InputList[i]$ such that every element of the sequence $OutputList[0]$ appears in $InputList[i]$.
- $\forall i(OutputList[0][i] \leq OutputList[0][i+1])$. This relation describes the elements in the sequence $OutputList[0]$ are in non-decreasing order.
- $sum(InputList[0]) \in OutputList[0]$. This relation describes the summation of the sequence $InputList[0]$ is in the sequence $OutputList[0]$.

For convenience, we denote $R$ the set of relations described in IOLRL. For a relation in $R$, we use it to test if the input/output lists of a test case satisfy this relation. Therefore we have the following definition.

**Definition 6.** ***(Evaluation)*** *Given a test case $t$ and a relation $r \in R$, an* evaluation $\eta_r : (\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z}), \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})) \rightarrow \{0,1\}$ *is a function such that*

$$\begin{cases} \eta_r(t^I, t^O) = 1 \text{ if } t \text{ satisfies } r \\ \eta_r(t^I, t^O) = 0 \text{ if } t \text{ does not satisfy } r \end{cases} \qquad \blacksquare$$

We introduce IOLRL and the evaluation in this paper for two main reasons.

1. We assume that an encoding with emphasis on the relation between the input and output lists can enhance the performance of the constructed test oracle.
2. Since we use SVM to construct the test oracle and the training data points processed by SVM are numbers, we need an encoding mechanism from a test case into numbers. The evaluation of test cases for each relation provides such transformation.

## 5 Implementation

This work is implemented with the auxiliaries of the genetic programming library *GPC++* [14] and the SVM library *libsvm* [2]. Given a set of labelled test cases as the training data and a PUT, the basic idea is to use GP to evolve several relations to well separate the *pass* and $fail$ test cases. Here the individuals of GP are relations in IOLRL syntax. Then with the evolved relations, we train an SVM model based on the evaluations and labelling of these test cases. This trained SVM model is thus our constructed test oracle.

Figure 1 is the flowchart to construct a test oracle. We first explain the procedures as follows and the detail procedures in the next.

**Step 1.** Create an initial population with $n$ relations by the ramped half-and half method. The maximum tree depth is arbitrarily set to 8.

**Step 2.** Evaluate each labelled test case with the $n$ relations.

**Step 3.** Increase *gen* by 1. If *gen* does not exceed the user-defined maximum generation, calculate the fitness for each relation and go to step 4. Otherwise, go to step 5.

**Step 4.** Repeat to generate new offspring relations until $n$ relations are generated. By setting the probability to 90% of performing crossover operation, Approximately 90% of the new population are generated with crossover operator. The rest of the population are generated with reproduction operator. Go to step 2. Note that 90% of crossover rate is set here because this is a typical setting [9]. We also simply eliminate the mutation operator because the probability to apply a mutation operator is relatively small (typically 1% rate).

**Step 5.** Train an SVM model as the constructed test oracle according to the evaluation of the training data. The used SVM kernel is radial basis function (RBF).

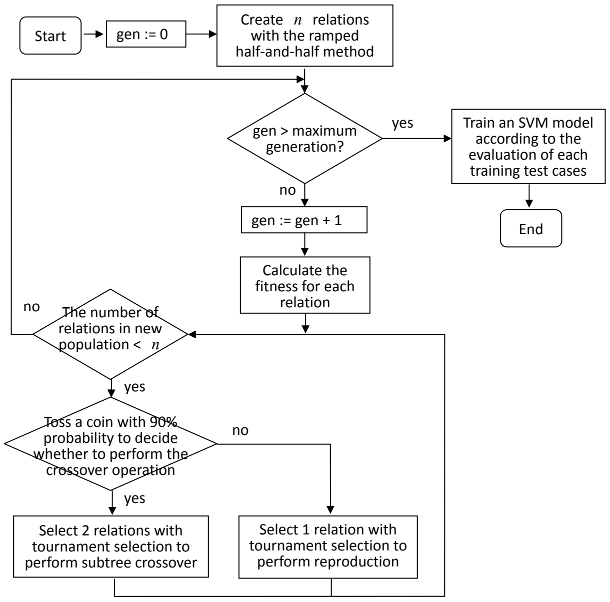

Start
gen := 0
Create n relations with the ramped half-and-half method
gen > maximum generation?
yes
Train an SVM model according to the evaluation of each training test cases
End
no
gen := gen + 1
Calculate the fitness for each relation
no
The number of relations in new population < n
yes
Toss a coin with 90% probability to decide whether to perform the crossover operation
no
yes
Select 2 relations with tournament selection to perform subtree crossover
Select 1 relation with tournament selection to perform reproduction

**Fig. 1.** The flowchart of the proposed method

Intuitively, a good relation should be able to distinguish test cases that are labelled to different verdicts and should not be able to distinguish test cases that are labelled to the same verdict. Therefore for those test cases that are labelled to *pass*, all the evaluations of these test cases should be either 0 or 1. And for those test cases that are labelled to *fail*, all the evaluations should be in the opposite side. The procedure to calculate the fitness of a relation in step 3 is described as follows. Note that the less returned number of this procedure means the better of the relation.

```
relation_fitness
input A list of test cases $T$ and a relation $r$.
output A real number.
1: Let $m$ and $v$ be the mean and variance of $\{\eta_r(t^I, t^O) | t^v = pass \text{ and } t \in T\}$.
2: Let $m'$ and $v'$ be the mean and variance of $\{\eta_r(t^I, t^O) | t^v = fail \text{ and } t \in T\}$.
3: Return $v + v' - |m - m'|$.
```

To preserve the type consistency property in the generated relations, we implement the GP as *strongly typed GP* [9]. Therefore in an IOLRL syntax tree, every terminal symbol has a type and every non-terminal symbol has types for each of its arguments and a type for its return value. The process that generated the initial random individuals and the crossover operator are implemented under the type constraints. In our experiment, each left-hand-side symbol of IOLRL grammar rules has its own type. The procedure to create an individual in step 1 can only randomly pick a symbol of the demanded type to expand a syntax tree. The crossover operation in step 4 can only swap subtrees that return the same type.

As for the evaluation safety property, we can see that a generated relation may not be able to be evaluated. For example, $\forall i(\exists i(OutputList[0][j] < 9))$ is unable to be evaluated for any test case. Therefore for those relations that are unable to be evaluated, their fitness are set to the worst value among the population.

Since we construct a test oracle with SVM in step 5 and the input of SVM is a set of labelled numeric points, we need to prepare an input for SVM from the evolved relations and training data.bbvv Therefore we have the following procedure. Then the training procedure of SVM is inferred to [2].

```
prepare_training_data
input A list of relations $R$ and a list of test cases $T$.
output a set of training data points.
1: Let $D$ be an empty set.
2: for $i = 1$ to $|T|$ do
3:    Let $t$ be the $i$th relation of $T$.
4:    Let $M$ be a zero vector of size $|R|$.
5:    for $j = 1$ to $|R|$ do
6:       Let $r$ be the $j$th relation of $R$.
7:       $m_j = \eta_r(t^I, t^O)$. /* $m_j$ is the $j$th element of $M$. */
8:    end for
9:    if $t^v$ is $pass$ then $label = 1$ else $label = -1$ end if
10:   $D = D \cup (M, label)$.
11: end for
12: Return $D$.
```

## 6 Experiment

We use the following three benchmarks(PUTs) to demonstrate our technique.

– **Binary search.** The binary search program takes a numeric sequence and a number as its input and should respond the index of the number in the sequence. If the number is not in the sequence, $-1$ is responded. An example test case is $(\{\{1, 4, 5, 6, 7, 7\}, \{0\}\}, \{\{-1\}\}, pass)$.
– **Quick sort.** The quick sort program takes a numeric sequence as its input and should respond the sequence in non-decreasing order. An example test case of this program is $(\{\{4, 5, 6, 8, 9, 4, 7, 10, 21\}\}, \{\{4, 4, 5, 6, 7, 8, 9, 10, 21\}\}, pass)$.
– **Set intersection.** The set intersection program takes 2 numeric sequences as its input and should respond a sequence of the intersected numbers. In order to perform the operation efficiently, set intersection programs generally require the input sequences to be ordered and therefore the intersected sequence is also ordered. The benchmark here requires that the input and output sequences are in non-decreasing order. An example test case is $(\{\{1, 5, 7, 8, 9\}, \{1, 5, 9\}\}, \{\{1, 5, 9\}\}, pass)$.

For each benchmark, we generate a data set with the following strategies.

– The half of the data set are test cases with *pass* verdict and the others are with *fail* verdict.
– For each test case, the numbers in the input list are generated at random between 1 to 50 and the size of the sequences in the input list are decided at random between 1 to 20.
– For test cases with *pass* verdict, the output lists are computed according to the correct behavior of the benchmark.
– For test cases with *fail* verdict, the output lists are generated with the following types of fault at uniform distribution.
  • **Binary search.** The output can be a random number other than the index of the searched number, the searched number itself rather than the index, and a sequence of randomly generated numbers.
  • **Quick sort.** The output can be a sequence of randomly generated numbers with different/the same size of the sequence in the input list, and an ordered sequence of randomly generated numbers other than the correct sequence.
  • **Set intersection.** The output can be a sequence of randomly generated numbers with random size/minimal size of sequences in the input list, and an unordered sequence of intersected numbers.

Note that the domain of randomly generated numbers and the size of sequences are limited for simplicity.

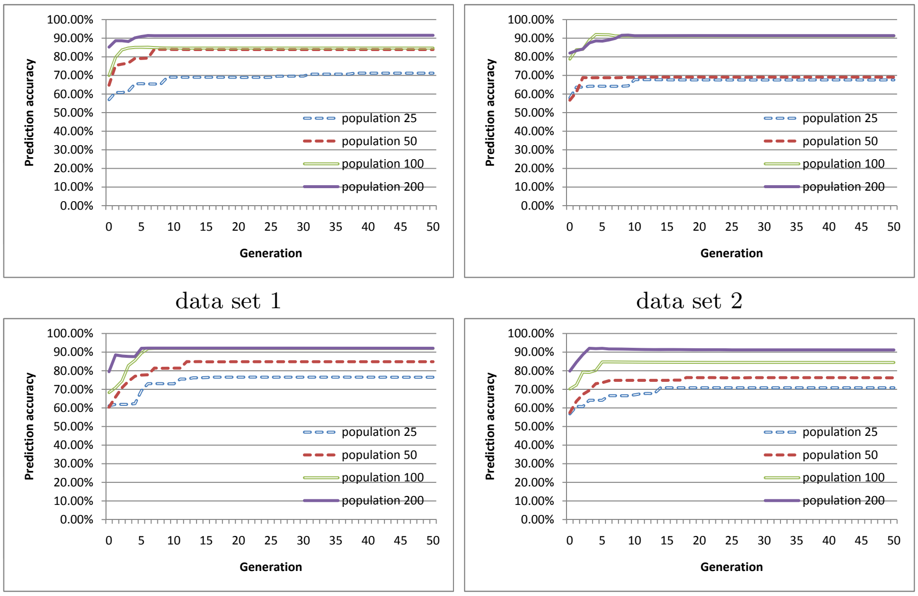
The experiments are conducted in two parts. The first part presents the performance data of our approach with different parameters and the next part presents the comparison data of the neural network approach in [10] and our approach with GP evolved/user specified relations.

## 6.1 Performance Study

Since our approach only require a set of labelled test cases to construct a test oracle, users may want to know how many test cases are sufficient and how to setup the GP parameters with better performance. In this section, we present the experimental studies of training data size and GP population size. The performance of constructed test oracle is measured with *prediction accuracy* and *time cost*. The prediction accuracy is the percentage of correctly labelled test cases of a testing data set with the constructed test oracle. The time cost is the used time to evolve the relations and train the test oracle. The experimental data are collected on Intel Pentium CPU T4200@2.00 GHz with 3G RAM, running Ubuntu 8.04.

To objectively demonstrate the performance with different population size, we have collected the performance data for each benchmark with 4 different training data sets of size 50. For each benchmark, the prediction accuracy is tested with a testing data set of size 3000. Each experiment is run for 10 times and the average performance data is reported. As we can see in Figure 2, Figure 3 and Figure 4, the prediction accuracy climbs fast and is better with larger population size. The impact of population size is significant because greater diversity in the population can result in higher chance to evolve good relations. The time cost is summarized in Table 1, Table 2 and Table 3. The tables indicate our approach constructs a test oracle in reasonable time with good accuracy.

To objectively demonstrate the effect of different training data size, we have collected the performance data for each benchmark with 4 different populations. Each population consists of 400 relations and the maximal generation is 50. For each benchmark, the prediction accuracy is tested with a testing data set of 3000



data set 1 data set 2

data set 3 data set 4

**Fig. 2.** Prediction accuracy with different population size for binary search benchmark

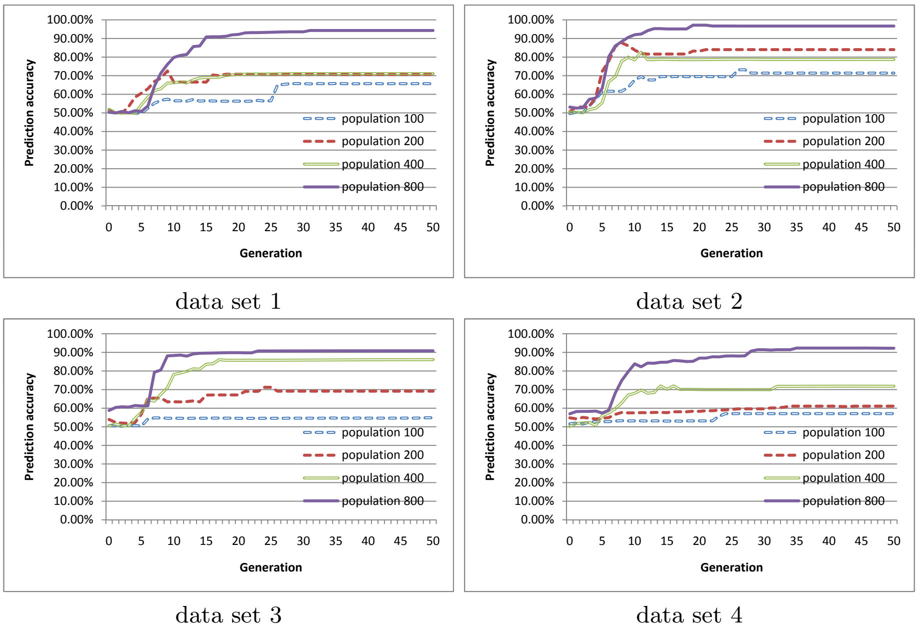


**Fig. 3.** Prediction accuracy with different population size for quick sort benchmark

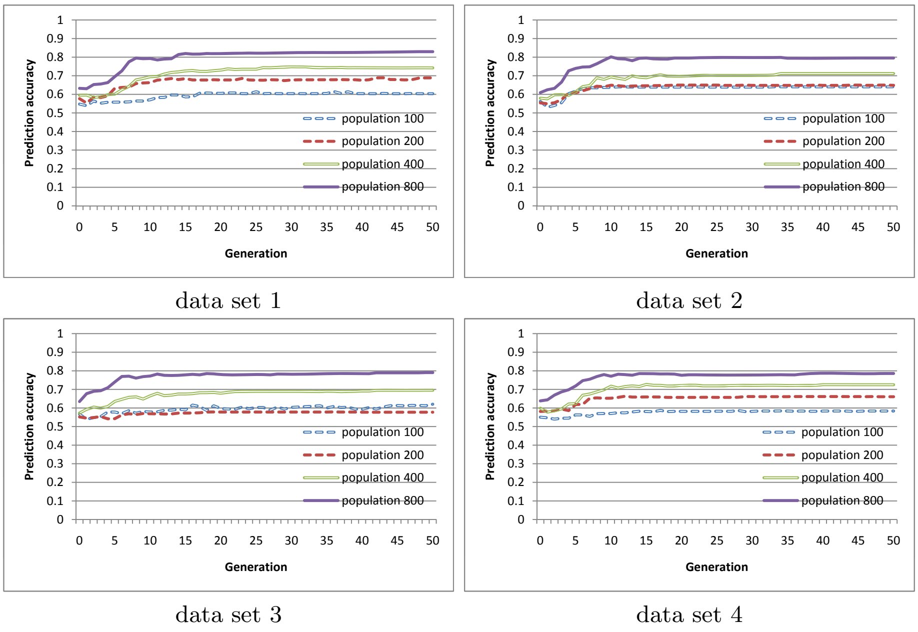


**Fig. 4.** Prediction accuracy with different population size for set intersection benchmark

**Table 1.** Performance data of binary search benchmark

| Population size | 25 | | | | 50 | | | | 100 | | | | 200 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| data set | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| accuracy(%) | 71.1 | 67.6 | 76.5 | 70.8 | 83.9 | 69.1 | 84.9 | 76.2 | 84.7 | 91.2 | 92.0 | 84.4 | 91.6 | 91.4 | 92.1 | 91.2 |
| time cost(s) | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.2 | 0.2 | 0.3 | 1.4 | 0.6 | 1.0 | 0.8 | 3.2 | 1.9 | 3.5 | 2.6 |

**Table 2.** Performance data of quick sort benchmark

| Population size | 100 | | | | 200 | | | | 400 | | | | 800 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| data set | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| accuracy(%) | 65.7 | 71.4 | 54.9 | 57.1 | 70.8 | 84.0 | 69.1 | 61.1 | 71.1 | 78.7 | 86.1 | 71.8 | 94.3 | 96.7 | 90.9 | 92.3 |
| time cost(s) | 0.5 | 2.2 | 1.4 | 4.4 | 3.1 | 4.4 | 1.7 | 4.2 | 7.9 | 13.2 | 7.2 | 21.6 | 33.5 | 41.9 | 64.3 | 43.9 |

**Table 3.** Performance data of set intersection benchmark

| Population size | 100 | | | | 200 | | | | 400 | | | | 800 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| data set | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
| accuracy(%) | 60.3 | 64.0 | 62.0 | 58.4 | 68.9 | 64.8 | 57.7 | 66.1 | 74.3 | 71.2 | 69.5 | 72.5 | 83.0 | 79.5 | 79.0 | 78.6 |
| time cost(s) | 7.9 | 8.5 | 2.1 | 2.1 | 5.5 | 10.3 | 5.3 | 19.1 | 43.6 | 67.3 | 18.1 | 20.7 | 138.6 | 183.3 | 58.9 | 287.5 |

test cases. Each experiment is run for 10 times and the average performance data is reported. As we can see in Table 4, the training data size does not effect the prediction accuracy very much.

The parameter studies of the population size and training data size have demonstrated the key point of our approach is to have well-evolved relations to describe the behaviors of benchmarks rather than lots of training data. This is especially useful in applications that are lack of training data.

## 6.2 Compare with Manually Specified Relations and Neural Network Approaches

In the perfect condition of our approach, the GP evolved relations can completely distinguish the test cases of *pass* and *fail* verdicts. Therefore we show the prediction accuracy of our approach by replacing the GP evolved relations with manually specified relations in IOLRL. The manually specified relations describe the expected behaviors of the three benchmarks. For those test cases that are labelled *pass*, they will be evaluated to be *true* with these relations, and vice versa. Therefore, they can completely distinguish the *pass* and *fail* test cases. This experiment is to show the sufficiency of IOLRL as the input/output list relation specification language for the three benchmarks and the performance of the constructed test oracle by SVM. We also compare the performance data to our original approach to show the effectiveness of the GP procedures. At last, we compare with the neural network based approach proposed by [10].

**Table 4.** Prediction accuracy with different training data size

| Benchmark | population | Training data size | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 10 | 20 | 50 | 100 | 200 | 400 | 800 | 1600 |
| Binary search | 1 | 86.1 | 89.7 | 79.1 | 87.8 | 92.1 | 91.5 | 91.5 | 91.7 |
| | 2 | 90.8 | 91.9 | 91.0 | 92.1 | 91.6 | 92.3 | 92.2 | 91.9 |
| | 3 | 91.8 | 91.8 | 92.2 | 91.9 | 92.3 | 92.0 | 92.0 | 92.3 |
| | 4 | 89.0 | 89.8 | 91.8 | 91.7 | 91.8 | 91.9 | 91.9 | 92.0 |
| Quick sort | 1 | 60.0 | 56.7 | 63.3 | 63.7 | 60.8 | 82.2 | 69.5 | 65.5 |
| | 2 | 63.5 | 61.2 | 59.1 | 61.9 | 57.5 | 58.7 | 75.4 | 70.7 |
| | 3 | 59.3 | 64.2 | 52.3 | 63.7 | 62.4 | 61.4 | 83.2 | 73.9 |
| | 4 | 62.9 | 66.2 | 68.4 | 55.6 | 68.9 | 59.9 | 65.4 | 69.2 |
| Set intersection | 1 | 54.7 | 61.3 | 54.8 | 56.7 | 58.9 | 56.7 | 57.6 | 58.9 |
| | 2 | 53.9 | 58.3 | 57.0 | 59.9 | 64.3 | 57.9 | 59.2 | 61.8 |
| | 3 | 57.8 | 53.6 | 57.1 | 65.3 | 59.9 | 58.2 | 53.3 | 64.1 |
| | 4 | 53.6 | 59.1 | 52.8 | 57.2 | 55.3 | 57.5 | 57.5 | 54.0 |

**Table 5.** Comparison of prediction accuracy with 50 training data

| Benchmark | data set | Our approach | Our approach with manually specified relations | NN approach | NN approach with 400 training data |
|---|---|---|---|---|---|
| Binary search | 1 | 91.6 | 99.90 | 77.13 | 95.47 |
| | 2 | 91.4 | | 79.71 | 94.35 |
| | 3 | 92.1 | | 75.39 | 94.72 |
| | 4 | 91.2 | | 75.85 | 94.80 |
| Quick sort | 1 | 94.3 | 99.90 | 58.98 | 65.57 |
| | 2 | 96.7 | | 56.23 | 66.33 |
| | 3 | 90.9 | | 55.32 | 64.86 |
| | 4 | 92.3 | | 56.00 | 67.01 |
| Set intersection | 1 | 83.0 | 98.44 | 75.93 | 86.19 |
| | 2 | 79.0 | | 79.5 | 84.63 |
| | 3 | 79.5 | | 79.0 | 86.51 |
| | 4 | 83.0 | | 78.6 | 88.14 |

The manually specified relations are listed as follows.

– **Binary search**

$$\begin{aligned} &\forall i(InputList[0][i] \leq InputList[0][i+1]) \\ \wedge\ &(OutputList[0][0] < 0) \vee (InputList[0][OutputList[0][0]] = InputList[1][0]) \\ \wedge\ &(OutputList[0][0] \geq 0) \vee (\neg(InputList[1][0] \in InputList[0]) \end{aligned}$$

– **Quick sort**

$$\begin{aligned} &\forall i(OutputList[0][i] \leq OutputList[0][i+1]) \\ \wedge\ &\forall i(InputList[0][i] \in OutputList[0]) \\ \wedge\ &\forall i(OutputList[0][i] \in InputList[0]) \end{aligned}$$

– **Set intersection**

$$\begin{aligned} &\forall i(OutputList[0][i] \in InputList[0] \wedge OutputList[0][i] \in InputList[1]) \\ \wedge\ &\forall i(InputList[0][i] \in OutputList[0] \vee \neg(InputList[0][i] \in InputList[1])) \\ \wedge\ &\forall i(InputList[1][i] \in OutputList[0] \vee \neg(InputList[1][i] \in InputList[0])) \\ \wedge\ &\forall i(OutputList[0][i] \leq OutputList[0][i+1]) \\ \wedge\ &\forall i(InputList[0][i] \leq InputList[0][i+1]) \\ \wedge\ &\forall i(InputList[1][i] \leq InputList[1][i+1]) \end{aligned}$$

The comparison data is shown in Table 5. 800 population size and 50 maximal generation are set in our approach. The experimental data shows that our approach performs better than neural network approach with small training data size and evenly with larger training data size. It also shows that the IOLRL can well-described the behaviors of the three benchmarks and the evolved relations can approximate the performance of the manually specified relations.

## 7 Conclusion

In this paper, we model PUTs in black-box manner and design IOLRL to formally describe the relations between the input and output list of PUTs. We use genetic programming to evolve relations in IOLRL that can well separate *pass* and $fail$ test cases. With these relations and some labelled test cases, we build an SVM model as a test oracle. The advantage of our approach is only few test cases are needed to construct a test oracle. This is because we increase the variable dimensions of the test cases by supplying large set of GP population to evaluate the test cases. The potential problem is our designed IOLRL is not sufficient to effectively describe the I/O relations that can well separate the *pass* and $fail$ test cases. However the experiments show the designed IOLRL is sufficient for the three benchmarks. The performance gap between the evolved relations and the manually specified relations can be compensated with better configurations of the GP procedure. In summary, this paper proposes a novel approach to evolve a test oracle with test cases and the experimental data shows our constructed test oracle performs well.

## References

1. Brown, D.B., Roggio, R.F., Cross II, J.H., McCreary, C.L.: An automated oracle for software testing. IEEE Transactions on Reliability 41(2), 272–280 (1992)
2. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines (2001), Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
3. Chen, T.Y., Ho, J.W.K., Liu, H., Xie, X.: An innovative approach for testing bioinformatics programs using metamorphic testing. BMC Bioinformatics 10 (2009)
4. Dustin, E., Rashka, J., Paul, J.: Automated software testing: introduction, management, and performance. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)

5. Edvardsson, J.: A survey on automatic test data generation. In: Proceedings of the Second Conference on Computer Science and Engineering in Linköping, October 1999. ECSEL, pp. 21–28 (1999)
6. Jones, E.L., Chatmon, C.L.: A perspective on teaching software testing. In: Proceedings of the seventh annual consortium for computing in small colleges central plains conference on The journal of computing in small colleges, USA, pp. 92–100. Consortium for Computing Sciences in Colleges (2001)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
8. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering 24(3), 161–173 (1998)
9. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming (2008), Published via, `http://lulu.com`, and freely available at `http://www.gp-field-guide.org.uk` (With contributions by J. R. Koza)
10. Vanmali, M., Last, M., Kandel, A.: Using a neural network in the software testing process. Int. J. Intell. Syst. 17(1), 45–62 (2002)
11. Vapnik, V.N.: Statistical Learning Theory. Wiley Interscience, Hoboken (1998)
12. Walker, M.: Introduction to genetic programming (2001)
13. Wang, Y.-C.F., Casasent, D.: A hierarchical classifier using new support vector machine. In: ICDAR, pp. 851–855. IEEE Computer Society Press, Los Alamitos (2005)
14. Weinbrenner, T.: Genetic programming techniques applied to measurement data. Diploma Thesis (February 1997)