Constructing Automated Test Oracle for Low Observable Software

Meysam Valueian¹,², Niousha Attar¹,³, Hassan Haghighi¹,³, and Mojtaba Vahidi Asl¹,³,⁴

¹Faculty of Computer Science and Engineering, Shahid Beheshti University, G.C, Tehran 1983963113, Iran.
²valueian@ce.sharif.edu
³{n_attar, h_haghighi, mo_vahidi}@sbu.ac.ir
⁴Corresponding author: Mojtaba Vahidi Asl, Tel: +982129904131+989123867040

Abstract

Using machine learning techniques for constructing automated test oracles have been successful in recent years. However, existing machine learning based oracles have deficiencies when applied to software systems with low observability, such as embedded software, cyber-physical systems, multimedia software programs, and computer games. This paper proposes a new black box approach to construct automated oracles which can be applied to software systems with low observability. The proposed approach employs an Artificial Neural Network algorithm which uses input values as well as corresponding pass/fail outcomes of the program under test, as the training set. To evaluate the performance of the proposed approach, we have conducted extensive experiments on several benchmarks. The results manifest the applicability of the proposed approach to software systems with low observability as well as its higher accuracy in comparison to a well-known machine learning based method. We have also assessed the effect of different parameters on the accuracy of the proposed approach.

Keywords—Software Testing, Test Oracle, Machine Learning, Artificial Neural Network, Software Observability.

1 Introduction

Due to considerable size, complexity, distribution, pervasiveness, and criticality of recent software systems, producing fault-free software seems to be an
unattainable dream. Furthermore, the increasing demand for software programs and competition in software industry lead to a short time-to-market which increases the likelihood of shipping faulty software. These facts highlight the significance of quality assurance activities in improving software quality and reliability, among which software testing plays a vital role.

Software testing is known as a labor-intensive and expensive task. The huge cost, difficulty, and inaccuracy of manual testing have motivated researchers to seek for automated approaches which aim at improving the accuracy and efficiency of the task. As a consequence, there has been a burgeoning emergence of software testing methodologies, techniques, and tools, to support testing automation, in recent years.

Among different testing activities, test data generation is one of the most effective ones which aims at creating an appropriate subset of input values to find out whether a program produces the intended outputs. The input values should satisfy some testing criterion such that the testers have a dependable estimation of the program reliability [1]. To reduce laboriousness, inaccuracy and intolerable costs of manual test data generation, a significant amount of research has been dedicated to automate the process of test data generation [2,3].

Despite great advances in different software testing domains, e.g. test data generation, one important challenge has been overlooked by academia and industry. The question arises, who will evaluate the correctness of the outcome/behavior of a software program according to the given test input data. The mechanism of labeling program outcomes, as pass or fail, subjected to specific input values is referred to as "test oracle" [4].

Providing an accurate and precise test oracle is the main prerequisite for achieving robust and realistic software testing techniques and tools. This essential aspect of software testing, left as an open problem in many works, is fronting serious challenges. Typically, in real world applications, there might be no common test oracle except human assessment on pass or fail outcome of a program run [5]. One primary challenge for manual test oracles is the diversity and complexity of software systems and platforms, each of them having various types of input parameters and output data. Also, due to demanding business expansion and thanks to advances in communications technologies, software components may be produced simultaneously by numerous developers, perhaps located in far apart places. This makes the manual judgment for the correctness of diverse and complicated components inaccurate, incomplete and expensive.

The mentioned challenges emphasize the need for automatic test oracles. A typical automatic test oracle contains a model or specification by which the outcome of a Software Under Test (SUT) could be assessed. Fig. [1] illustrates the structure of a conventional automated test oracle. As shown in the figure, the test inputs are given to the both SUT and automated test oracle. Their outputs are compared with an appropriate comparator which decides whether the outcome of the SUT, subjected to the given test inputs, is correct.

An automated test oracle can be derived from the formal specification of the SUT. Usually, there is a lack of full formal specification about the features of the
SUT. In these situations, we can use a partial test oracle which can assess the program behavior with respect to a subset of test inputs \[4\]. One approach for producing a partial oracle is using metamorphic testing, i.e., building the test oracle based on known relationships between expected behaviors of the SUT.

Recently, Machine Learning (ML) methods have been employed to build test oracles. A typical machine learning based test oracle involves two main steps: choosing an appropriate learning algorithm for constructing a learning model and preparing an appropriate training dataset from SUT’s historical behavior, represented as input-output pairs; this process is illustrated in Fig. 2. According to the figure, the constructed model reflects the correct behavior of the SUT assuming that the model has 100% precision.

Most of ML-based approaches model the relationships between a limited number of inputs and the corresponding output values, as the training set. The idea behind these approaches is that, the ML-based oracles generalize the limited relationships, involved in the training set, to the whole behavior of the SUT according to its input space. In other words, they produce expected results for the remaining inputs based on what they learned in the training phase. In the testing phase, the inputs of a test set are given to both SUT and test oracle. The expected results, generated by the oracle, is compared with the actual outputs of the SUT. If the results are similar or close together (based on a predefined threshold), the outcome is labeled passed, and otherwise, the execution is considered failed. The choice of the machine learning method may affect the precision of the test oracle.

Some of ML-based works are based on classifying the software behavior by using input/output pairs \[6\], and sometimes, along with execution traces \[6–8\] as input features of the classifier. Neither of the mentioned methods are applicable for low observable software systems. The reason is that, in these systems, the expected results and actual outputs are not easily representable. To be more precise, it is not easy to observe and understand the behavior of low observable systems in terms of their outputs, effects on the environment and other hardware and software components. Sometimes, even if the outputs are observable, their precise encoding or representation, which is a prerequisite for comparison, is difficult. Therefore, in cases categorized below, comparing the expected and actual results is inaccurate or even impossible.

1. Embedded software testing, in which the software has a direct effect on a hardware device(s) which limits the observability of the software \[9\]. In these situations, assuming that the hardware works correctly, most of the testers prefer to evaluate an embedded software by observing the behavior of the hardware.

2. Multimedia software programs, e.g., image editors, audio and video players, and computer games, where typically there is no scalar (and even structured) output. In these programs, the types of outputs are commonly unstructured or semi-structured \[10\]. Therefore, encoding these outputs is mainly difficult, inaccurate or even infeasible while labeling them with pass/fail is usually feasible.
3. GUI based programs, in which users are faced with various graphical fields instead of neat values. Unlike the previous case, these programs have structured outputs. But their encoding to numerical values is a challenging process. In these situations, testers usually label the outputs with pass/fail through taking the opinion of domain experts.

4. Compatibility testing: in this case, the goal is to evaluate the compatibility of the SUT’s behavior on different platforms [11]. For example, the compatibility testing of web based applications involves evaluating the rendered web pages on different browsers. Encoding, representing, and comparing actual results on different platforms are too difficult and labor-intensive while labeling them with pass/fail is too much simpler.

There are also some other situations where the existing methods are not applicable. For instance, suppose cases in which explicit historical data, represented as input-output pairs, is not available, while there are documents which inform us about failure-inducing inputs or scenarios. The failure-inducing inputs refer to a subset of input values causing the program to behave incorrectly. Similarly, failure-inducing scenarios address those conditions in which the program generates unexpected results. In many industrial software projects, there exist test reports related to previous iterations of the test execution which indicate pass/fail scenarios. These reports involve historical information and therefore can be useful to construct appropriate test oracles, although, they do not include input-output pairs. As another example, consider documents produced by end users during the beta testing process of a software product, which contain valuable pass/fail scenarios.

As an idea, in all the mentioned situations, we can model the relationships between inputs and corresponding pass/fail behaviors (instead of the corresponding outputs) of the program. In this way, without using concrete output values, we may achieve a test oracle which can be used to predict the pass/fail behavior of the program for given test inputs, during the testing phase.

Based on the above idea, in the conference paper [12], we employed a learning based approach which merely requires input values, and the corresponding pass/fail outcomes/behaviors, as the training set. The training set is used to train a binary classifier which serves as the program’s test oracle. Later, during the testing phase, several input parameters for which the corresponding execution outcome is unknown, are given to the classifier. The classifier labels the outcome as pass or fail. Fig. 3 illustrates an overview of our approach using Artificial Neural Network (ANN) as the binary classifier.

Besides solving the mentioned problems, the presented approach in [12] has the following advantages:

- It is based on black-box testing. This means that we do not need neither the program’s source code nor the design documents of the SUT to generate an automated oracle.

- Regarding the testing phase of the proposed approach, shown in Fig. 3, there is no need to execute the SUT to compare its output with the oracle’s
output; as mentioned, only the input data is required in the testing phase. This is advantageous specifically when there is no access to the SUT or when software execution is a time-cost consuming or risky work. These considerations are usually seen in many low observable and safety-critical systems.

• The test oracles presented in [1] and [13] are approximators rather than classifiers. They assume that each of the output components in the input-output pairs given to the SUT at the training phase is correct with respect to the corresponding input, or in other words, the oracle model constructed by the given inputs in the test set, reflects the intended behavior of the SUT. However, this assumption introduces a considerable limitation when preparing historical data for the training phase since those input values inducing unexpected results must be discarded. In contrast, since our proposed approach considers both passing and failure-inducing inputs for model construction, it does not suffer from this limitation.

It should be mentioned that this paper is an extended version of [12] and some parts of this paper have already been presented in [12]. In comparison to [12], the extensions in this paper are:

• We conduct various experiments to evaluate the proposed approach in terms of the following parameters: percentage of passing test cases in the training dataset, Code Coverage Percentage (CCP) of the training dataset, training dataset size, and configuration parameters of ANN. According to the experiments, these parameters have major impacts on the accuracy of the constructed oracle.

• In [12] we only considered benchmarks with integer input values. However, there are many situations where low observable systems (e.g. embedded software and cyber-physical systems) accept signals instead of ordinary input values such as integers. In these situations, binary files play the role of inputs for the embedded software. These kinds of inputs are not suitable for the classifier and make it unreasonably complex with respect to the ANN size. In this paper, we decrease this complexity by reducing the size of binary files for the classifier. In order to demonstrate the capability of our approach for programs with input signals, we added three related programs to our benchmarks.

• Literature review is updated according to the researches that have been done in recent years.

The remaining parts of the paper are organized as follows. Section two is dedicated to the review of related works and a brief background of neural networks needed to read the paper. Section three describes the details of the proposed approach. In section four, the experimental results and analysis are presented. Section five includes conclusion and some directions for future work.
2 Literature Review

2.1 Related Works

Different approaches have been proposed for automating test oracles based on available software artifacts. Formal oracles use the existing formal specification of the systems’ behavior, typically based on the mathematical logic. Formal specification languages can be roughly categorized into two groups:

1. Model-based specification languages, which include states and operations where pre-conditions and post-conditions constrain the operations. Each operation may limit an input state by some preconditions while post-conditions define effects of the operation on the program state. Peters et al. have proposed an algorithm to generate a test oracle from program documentation [14]. In their approach, the documentation is written in fully formal tabular expressions.

2. State transition systems, which have graphical syntax, including states and transitions between them. Here, states are abstract sets of concrete states of the modeled system. Gargantini et al. used Abstract State Machines (ASMs) as an oracle model to predict the expected outputs of the SUT [15].

Although, formal specification based oracles are highly accurate and precise, their applicability is limited. Generally, for most of the software products, there exists no formal specification to construct an adequate and complete test oracle. Furthermore, in most situations, it is costly and difficult to write documentations of an SUT in a formal way.

Implicit oracles are generated using some implicit knowledge to evaluate the behavior of the SUT. Implicit oracles can be used to detect anomalies such as buffer overflow, segmentation fault, etc, which may cause programs to crash or show an execution failure [16–18]. Therefore, there is no need for any formal specification to generate this kind of oracle and it can be used for almost all programs. For example, Walsh et al. [19] proposed an automated technique to detect some types of layout failures in responsive web pages using the implicit knowledge of common responsive failure types. In this approach, they distinguish between intended and incorrect behavior of a layout by checking elements’ positions relative to each other in different viewport widths. For example, if two elements of a layout always overlap in different viewports, the effect is considered intended. If the elements overlap infrequently, it may occur a responsive layout failure. According to the empirical study in [19], their approach detected 33 distinct failures in 16 of 26 real-world web pages.

There are some approaches which use semi-formal or non-formal documents, data sets collected during system executions, properties of SUT, etc. to produce an oracle model. Carver and Lei [20] proposed a test oracle for message-passing concurrent programs, using Labeled Transition Systems (LTSs). The main challenge of using stateful techniques to generate oracle for these kinds of programs
is the state explosion problem. Therefore, [20] proposed a stateless technique for generating global and local test oracles from LTS specification models. Local oracles are used to test individual threads, without testing the system as a whole. However, a global test oracle tests a global relation between the model of the system and its implementation using test inputs generated from a global LTS model of the complete system. Therefore, using local test oracles decreases the number of global, executed test sequences.

Metamorphic testing is used in some approaches to produce a partial oracle. This method utilizes metamorphic relations. For instance, if function \( f(x) = \sin(x) \) is implemented as a part of the SUT, a metamorphic relation would be \( \sin(\pi - x) = \sin(x) \) that must be held across multiple executions. The metamorphic relations are not necessarily limited to arithmetic equations. For example, Zhou et al. proposed an approach for testing search engines, e.g. *Google* and *Yahoo!,* using metamorphic relations [21]. They built metamorphic relations in terms of the consistency of search results. Discovering metamorphic relations is an important step to construct a test oracle. Automating this step is the most challenging part of the metamorphic testing. Simon has constructed a lazy systematic unit-testing tool called JWALK [22], in which the specification of a system is lazy and eventually learned by interacting between JWALK and the developer. This might be a convenient tool to extract metamorphic relations.

Goffi proposed a white-box method to evaluate software behavior in his thesis [23]. Also, he and his colleagues proposed two approaches to generate test oracles [24–27]. In the first work, oracle is generated from software redundancy [24, 26, 27], which is considered as a specific application of metamorphic testing [28]. They used the notion of cross-checking oracle. The idea behind this oracle is that, two similar sequences of method calls are supposed to behave equivalently, but their actual behaviors might be different because of a fault in the implementation. Therefore, if an equivalent check of two similar sequences fails, it is concluded that there is a fault in the code. In order to find identical sequences in the level of method calls, they proposed a search-based technique to synthesize sequences of method invocations that are equivalent to a target method within a finite set of execution scenarios [29]. By considering 47 methods of 7 classes taken from the Stack Java Standard Library and the Graphstream library, they automatically synthesized 123 equivalent method sequences of 141 manually identified sequences. It means that their approach generates 87% of equivalent method sequences, automatically. They improved their previous work by synthesizing more equivalent method calls for relevant components of the *Google Guava* library [30].

In the second approach, Goffi et al. [25] proposed a technique that automatically creates test oracles for exceptional behaviors from Javadoc comments. This method utilizes natural language processing and run-time instrumentation, and is supported by a tool called ‘Toradocu’.

Some of program behaviors can be automatically evaluated against the extracted/detected invariants. Program invariants are properties which must be held during program execution. For example, a loop invariant is a condition
that is true at the beginning, end, and at each iteration of the loop. Invariants can participate in test oracles considering that invariant detection is an essential part of this method. Ernst et al. proposed a machine learning based technique to detect invariants \cite{31}. They implemented the Daikon tool for this purpose \cite{32}.

Elyasov et al. \cite{33} proposed a new type of automated test oracles, called Execution Equivalence (EE) invariants. These invariants can be extracted from software logs, which include events and states of the software. They presented a tool called LOPI (LOg-based Pattern Inferencer) for mining EE-invariants. They also compared Daikon with LOPI on some benchmarks. According to the experiments, the effectiveness of EE-invariants is competitive with the ones found by Daikon.

N-Version programming can be applied to produce an oracle model, where software is implemented in different ways by different development teams but with the same functionality. Each of the software versions may be seen as a test oracle for the others. This solution is expensive because each version must be implemented by a different development team. Furthermore, there is no guarantee that one implementation is fault free to be a reference oracle model for the other versions. In order to decrease the cost of N-Version programming, Feldt proposed a method to generate multiple software versions, automatically, using genetic programming \cite{34}.

As another solution to the oracle problem, the notion of decision table has been used in \cite{35} just for Web Applications (WAs). In this work, a WA is considered composed of pages as a test model for both client and server side. Decision table is a representation of WA behavior. It contains two main parts: Condition Section (which is the list of combinational conditions according to inputs) and Action Section (which is the list of responses to be produced when corresponding conditions are true). Table 1 illustrates a decision table template. In this table, the Input Section demonstrates conditions related to Input Variables, Input Actions and State Before Test, while in the Output Section, the actions associated with each condition is described by Expected Results, Expected Output Sections and Expected State after Test. At the end of the execution of each test case, a comparator compares actual results against the expected values of output variables, output actions, exceptions, and the environment state obtained after the test execution.

Memon et al. applied AI planning to generate oracle models for Graphical User Interfaces (GUIs) \cite{36}. Their models contain two parts: Expected-state Generator and Verifier. In order to generate expected-state, they used a formal model of GUI, which is composed of GUI elements and actions, in which actions are displayed by preconditions and effects. This model is derived from specifications. Therefore, this approach is feasible if there exist appropriate specifications. When a test case runs, the actual values of properties for an element or elements are known. At this moment, the verifier can compare these values against the expected values, to determine if they are equal. Therefore, the verifier is a process that compares the expected state of the GUI with the
actual state and returns a verdict of equal or not equal.

Last et al. demonstrated that data mining models can be employed for re-
covering system requirements, and evaluating software outputs [37]. To prove
the feasibility of the approach, they applied Info Fuzzy Network (IFN) as a data
mining algorithm.

Zheng et al. proposed a method to construct an oracle model for web search
engines [38]. They collected item sets which include queries and search results.
Then, they applied the association analysis technique to extract rules from the
items. The derived rules play the role of a pseudo test oracle, which means that
by giving new search results, the mentioned approach detects the search results
that violate the mined rules, and presents them to testers for manual judging.

Singhal et al. employed the Decision Tree algorithm to create oracle models
[39]. They have utilized code predicates to recognize inputs’ features to con-
struct a decision tree. They applied this approach to the triangle benchmark
program where the leaves of the tree are labeled with the triangle classes (equi-
lateral triangles, isosceles triangles, scalene triangles, and invalid triangles).

Wang et al. utilized Support Vector Machine (SVM), as a supervised ma-
chine learning algorithm, to train an oracle model [40]. They annotated the
program code of SUTs by Intelligent Test Oracle Library (InTOL) to collect
test traces according to procedure calls. They extracted features from each test
trace as an input for the SVM algorithm and then used the constructed SVM
model as the test oracle.

Vanmali et al. proposed an oracle using ANN to test a new version of soft-
ware [41]. Their methodology is based on black-box testing. It is assumed
that the functions existing in the previous version are still preserved in the new
version. Their training set includes inputs-outputs mappings from the previous
version of the software. They used predefined thresholds to compare the actual
result of the new version and the estimated expected output of the ANN.

Zhang et al. proposed an oracle to test SUTs that work as classifiers [42].
For example, the PRIME program, which is used in their evaluations, is a pro-
gram that works as a classifier. It determines whether the input number is a
prime number. Therefore, the output of the program is a member of the set
\{PRIM, NOT\_PRIM\}. They used a probabilistic neural network as an oracle
to test these kinds of SUTs. It is worth mentioning that they limited SUT to
classification problems while our method can be used for different types of soft-
ware.

Almaghairbe and his colleagues [43] proposed test oracles by clustering fail-
ures. They utilized anomaly detection techniques using software’s input/output
pairs. They did the experiments based on the assumption that failures tend to
be in small clusters. In the next work [44], they extracted dynamic execution
traces using Daikon and used them along with related input/output pairs in
order to improve the accuracy of the approach. Almaghairbe et al. also used
classification methods to evaluate the behavior of the software [6]. In another
work, LO et al. [7] proposed a method to classify software behavior by extract-
ing iterative patterns from execution traces. Also, Yilmaz et al. [8] proposed
a hybrid instrumentation approach that uses hardware counters for collecting
program spectra to train a decision tree classifier in order to classify the software behavior.

Shahamiri et al. exerted Single-Neural-Network to build oracle models aiming at testing program’s decision-making structures [45] and verifying logical modules [46]. Both decision rules and logical modules were modeled using the neural network technique.

In [13], an idea was presented to use multi neural networks for constructing test oracles. The proposed scheme includes an I/O relationship analysis to create the training dataset, training ANNs and testing the constructed ANN as an automated oracle. In [1], Shahamiri et al. showed how multi neural network could improve the performance of a test oracle on more complicated software, with multiple outputs, compared to their previous work. To this end, a separate ANN is constructed for each output. Therefore, the complexity of the SUT is distributed over several ANNs, and the ultimate oracle is composed of these ANNs.

The main difference between the idea in [1] and ours is the approach of composing the elements of datasets, an essential issue that has a significant impact on the ANN learning process and its precision. In our approach, there is no need to employ multi neural networks because the software’s concrete output values are not included in our datasets. For each input vector, a single value is assigned, which is the passing or failing state of the software after running the SUT with that input vector.

Some of the existing machine learning based oracles such as [1, 13] are constructed based on test sets containing a large number of test cases, each of which includes inputs and expected outputs. The oracle learns how to produce the desired outputs. The next step is comparing the desired outputs with the actual outputs, generated by the SUT, according to predefined thresholds. If the difference is less than the predefined threshold, the actual output will be considered as correct. Otherwise, it would be labeled fail. Also, some of the existing approaches, such as [6–8], consider outputs as input features of the machine learning method to classify the software behavior. Although, these proposed oracles can work on specific programs, they suffer from the following weak points:

- These methods cannot be applied to systems with low observability.
- Some works such as [1, 13] that generate actual outputs and compare them with expected results, cannot guarantee that the predefined thresholds are appropriate. This limitation highly affects the assessment of the constructed test oracle.
- Most existing approaches assume that the data samples given in the training phase are all correct. Based on this assumption, the constructed oracle only reflects the correct behavior of the SUT. Since they do not consider failing test cases, they have no idea about failing patterns of the code. Therefore, due to the lack of failing patterns, their model’s approximations are likely to be imprecise.
The mentioned approaches have to execute the SUT in order to achieve outputs required for evaluating the software behavior, but the proposed method in this paper does not need to execute the SUT at least in the testing phase.

Our proposed approach overcomes these deficiencies: it applies to systems which have low observability and/or produce unstructured or semi-structured outputs. Furthermore, since we consider pass/fail outcome/behavior of an SUT rather than its expected output values to train a binary classifier, there would be no need for thresholds or comparators. This significantly increases the accuracy of the constructed test oracle. In addition, we do not use outputs of the SUT in order to evaluate it. So, there is no need to execute the software, which is a costly and sometimes risky operation. Moreover, our test oracle is built according to both the correct and incorrect behavior of the SUT.

2.2 Artificial Neural Network

In recent years, a wide variety of machine learning methods have been proposed that can be used to discover hidden knowledge from data. Among many categories of industrial-strength machine learning algorithms, ANN has attracted a considerable amount of attention over past few years. According to [47], an Artificial Neural Network is a computational system that consists of elements, called units or nodes, whose functionality is based on biological neurons. As illustrated in Fig. 4, each neural cell consists of two important elements: (1) Cell body, which includes the neuron’s nucleus. It is worth noting that the computational tasks take place in this element. (2) Axon, which can be seen as a wire that passes an activity from one neuron to others. Furthermore, each neuron receives thousands of signals from other neurons. Eventually, these incoming signals reach to the cell body. They are integrated together in the cell body. If the resulting signal is more than a threshold, this neuron will fire and send signal to the forward neurons [47].

The main idea of the ANN algorithm is inspired by biological neural systems. Each unit or node in ANN is equivalent to a biological neuron. Each node receives inputs from other nodes or external resources. Also, each input is associated with some weight, which indicates the importance of that input. The node applies a function to the weighted sum of its inputs. This function is called ”Activation Function” and the result of this function is the output of the node. The structure of each node in ANN is illustrated in Fig. 5. The purpose of the activation function is to add a non-linearity feature to the node’s output. Since most of the real world data are non-linear and we intend to learn these non-linear representations, we use activation functions. Fig. 6 illustrates some activation functions that are useful in practice.

ANN can solve complex mathematical problems, such as stochastic and non-linear problems, using simple computational operations. Another feature of ANN is its self-organizing capability. This feature enables ANN to use the
knowledge of previous solutions in order to solve current problems. The ability of ANN in comparison with old mathematical methods are: (1) Parallel and high speed processing, (2) Learning and adapting with the environment of the problem [48]. Therefore, we use ANN as our classifier in the rest of this paper. In the following subsection, we describe the type of the neural network we use in the experiments.

2.2.1 Feed-Forward Neural Network

The Feed-Forward neural network is the simplest type of ANNs. It contains multiple nodes arranged in different layers. The nodes in each layer are fully connected to the nodes in the next layer. All connections are associated with some weights. Fig. 7 illustrates an example of a feed-forward neural network. A feed-forward neural network consists of three types of layers:

- **Input Layer**, which includes nodes that their inputs are provided from outside resources.

- **Hidden Layer**, which includes nodes that perform computational tasks on the outputs of the input layer. A feed-forward neural network has a single input and a single output layer, but zero or more hidden layers.

- **Output Layer**, which includes nodes that are responsible for computational tasks as well as transferring information from the network to the outside world.

It is worth noting that feed-forward neural networks with no hidden layers and at least one hidden layer are called Single Layer Perceptron (SLP) and Multi Layer Perceptron (MLP), respectively.

As mentioned in the previous section, the weights that are associated with inputs of a specific node illustrates the importance of each input to that node. Therefore, the main challenge of using ANN is how to calculate optimal weights of connections between nodes in order to produce desired output. The process of assigning optimal weights to the connections is called the training phase of ANN. In the next subsection, we introduce the algorithm that is utilized to train ANN in this paper.

2.2.2 Back-Propagation Algorithm

Back-propagation is one of the several ways to train ANN. It falls in the category of supervised learning algorithms, which means that it learns from labeled training data. This means, we know the expected labels for some input data.

Initially, the weights of all edges are randomly assigned. Then, for each input vector in the training dataset, ANN calculates the corresponding output. This output is compared with the desired label in the training set and the error is propagated back to the previous layer. The weights are adjusted according to the propagated error. This process is repeated until the error is less than a
predefined threshold. When the algorithm terminates, the ANN is learned. At this point, ANN is ready to produce output labels for new inputs.

3 The Proposed Method

To resolve the deficiencies of the existing machine learning based test oracles, we have applied a binary classifier to two classes of passing and failing input test data. The constructed oracle models the relationships between the inputs and the corresponding pass/fail outcomes of a given program.

Our training data can be provided from different resources which have already labeled a subset of program’s inputs as pass/fail according to the program’s outcome/behavior. For example, these resources could be human oracles, and documents or reports indicating passing/failing scenarios in the previous software versions. Since the cost of some of these resources could be non-trivial, we attempt to train the model with as small as possible dataset.

To provide an appropriate dataset, we seek for the available training resources. For example, we may build the dataset from both the available regression test suite and human oracles. For the latter case, we run SUT with some random input data and ask human oracle(s) or domain experts to label the outcomes as pass/fail. Typically, in real-world systems, the number of failing runs is less than the passing ones. Nevertheless, in our experiments, we consider various datasets with various ratios of passing and failing test data to analyze the sensitivity of the proposed approach in terms of different ratios.

The proposed approach has three main phases which are detailed in the remaining parts of this section. The flowchart of the approach is shown in Fig. 8.

3.1 Data preparation

The collected dataset includes inputs and the corresponding execution outcomes. Suppose the SUT has $n$ input parameters. The inputs can be organized as an input vector $X$, shown by Equation 1, where feature $x_i$ represents the $i$th input parameter of the SUT with $n$ input parameters.

$$X = < x_1, x_2, x_3, ..., x_n >$$

The set of all possible input vectors, represented by $T$ in Equation 2, can be shown as a Cartesian product of every input parameter domain, $D(x_i)$ (for $i : 1..n$).

$$T = D(x_1) \times D(x_2) \times ... \times D(x_n)$$

The number of possible input vectors is the product of the domain size of the input parameters as Equation 3.

$$|T| = |D(x_1)| \times |D(x_2)| \times ... \times |D(x_n)| = \prod_{i=1}^{n} |D(x_i)|$$
The SUT is executed by each input vector, and the resulting outcomes/behaviors of the execution is labeled with the members of set $C$ as Equation 4.

\[ C = \{ \text{Fail, Pass} \} \quad (4) \]

The training dataset is defined as a partial function from $T$ to $C$. For an SUT with $n$ input parameters and $m$ input vectors, the dataset includes two matrices:

1. An $m \times n$ input data matrix, where $x^i_j$, $1 \leq i \leq n$ and $1 \leq j \leq m$, is the value of input parameter $i$ from input vector $j$, as shown below.

\[
\begin{pmatrix}
  x^1_1 & x^2_1 & \ldots & x^n_1 \\
  x^1_2 & x^2_2 & \ldots & x^n_2 \\
  \vdots & \vdots & \ddots & \vdots \\
  x^1_m & x^2_m & \ldots & x^n_m
\end{pmatrix}
\]

2. An $m \times 1$ outcome vector, where $c_k \in \{0, 1\}$, $1 \leq k \leq m$, and $c_k = 1$ indicates that the result of the SUT subjected to the input data vector $<x^1_k, x^2_k, \ldots, x^n_k>$ is fail, otherwise pass. The vector is as follows:

\[
\begin{pmatrix}
  c_1 \\
  c_2 \\
  \vdots \\
  c_m
\end{pmatrix}
\]

3.2 Training a binary classifier with the training dataset

We have utilized a multilayer perceptron neural network for generating the oracle model. The output of each layer is fed into the next layer as input. In multilayer perceptron neural networks, the neurons are fully-connected, which means the output from each neuron is distributed to all of the neurons of the layer. At the beginning of the training phase, each connection is initialized with a random weight. As mentioned in section 3.1, we have produced a set of inputs and their corresponding pass/fail labels as the training dataset in order to apply to this network. A multilayer perceptron NN has three types of layers:

- Input layer: the inputs of the SUT are mapped to neurons of the input layer. Therefore, the number of neurons in this layer equals to the number of SUT’s inputs.
- Hidden layer: outputs of the input layer are fully-connected to the hidden layer as inputs. We applied ‘tan-sigmoid’, as an activation function, to this layer.
- Output layer: outputs of the hidden layer are fully-connected to inputs of this layer. We applied ‘sigmoid’, as an activation function, to this layer. Outputs of this layer are considered as the outputs of the created oracle model.
There are various parameters that determine how much the ANN should be trained. One of the most important parameters is called ‘epoch’, which is the number of iterations that training samples pass through the learning algorithm. In other words, epoch is the number of times that all of the training data are used once to update the weights of ANN. We considered the constant number of 1000, parameter Max in Fig. 8 for the epoch value in our experiments. The other parameter is ‘error’, which means that training will continue as long as the error of the training phase falls below this parameter value. In our experiments, if during the training process, the error rate is less than 1%, the Predefined value in Fig. 8 then the training phase will be stopped. Otherwise, it will continue until the number of iterations meet the epoch number. As illustrated in Fig. 8, in order to reduce the error of the model, the ANN’s iterative algorithm gradually changes the weights of connections between neurons based on the epoch value. For this purpose, a random weight is assigned to each connection at the beginning of the training phase. Afterward, we apply the training dataset on ANN. At the end of each iteration, the algorithm compares the output of the network to SUT’s corresponding pass/fail labels, which exist in the training dataset. If the error in the comparison process is less than a default value or the number of iterations becomes more than a specific threshold, the algorithm stops and the network is considered as the oracle model. Otherwise, in order to improve the model, the algorithm uses the back-propagation method and changes the weight of connections between neurons.

3.3 Evaluating the accuracy of the constructed model

To assess the accuracy of the constructed model, we have carried out various experiments by giving inputs from the parameters’ domain of the SUT which are not included in the training set. To this end, the training dataset is divided into several groups. Each group is selected such that the ratio of pass/fail input data in that group is different from other groups. This difference is required for the sensitivity analysis of the constructed model. The sensitivity analysis was done such that in each experiment, the impact of a particular parameter on the accuracy of the oracle was examined while the remaining parameters were unchanged. In addition to the pass/fail ratio in the training dataset, the studied parameters are the code coverage percentage of the training dataset, the size of the training dataset, and configuration parameters of ANN.

4 Evaluation

In this section, we have evaluated the proposed method on different types of software programs, especially embedded software. In the following, first we describe the experimental setup, and then we present and analyze the results of the experiments.
4.1 Experiment Setup

To evaluate the proposed method, we have used the neural network toolbox of MATLAB software version R2018a+update3 for building the neural network model [49]. Experiments were conducted on "Intel(R) Core i7-7500U CPU 2.70GHz up-to 2.90GHz, 8.0 GB RAM", and the operating system was "Windows 10 Pro 64-bit".

We have considered five benchmarks with different features and characteristics. For each benchmark, several faulty versions were generated. Three of five benchmarks, so called DES [50], ITU-T G718.0 [51], and GSAD [52], fall into the category of embedded software. The reason we have considered these benchmarks as embedded programs is according to the definition presented by Lee in [53]. According to this definition, embedded programs could have interactions with physical devices. They are not necessarily execute on computers and can be executed on cars, airplanes, telephones, audio devices, robots, etc.

The aim of choosing these three benchmarks is to show the applicability of the proposed method on programs with low observability as well as programs with unstructured or semi-structured outputs. Two other benchmarks, namely Scan and TCAS [54], have numerical inputs and outputs. These two benchmarks have been chosen to compare the proposed method with a known machine learning based method, proposed in [1]. From this point forward, we call the method in [1] as the baseline method. It is worth noting that the baseline method is not applicable for low observable programs and programs with non-numerical outputs. Table 2 illustrates the features of the selected benchmarks.

The application of each benchmark is as follows:

- **Scan** benchmark is a scheduling program for an operating system.

- **TCAS** benchmark, which is an abbreviated form of *Traffic alert and Collision Avoidance System*, is used for aircraft traffic controlling to prevent aircrafts from any midair collision.

- **DES** benchmark, which is an abbreviated form of *Data Encryption Standard*, is a block cipher (a form of shared secret encryption) that was selected by the National Bureau of Standards as an official Federal Information Processing Standard (FIPS) for the United States in 1976 [50]. It is based on a symmetric-key algorithm that uses a 56-bit key.

- **ITU – T** is one of the three sectors of the International Telecommunication Union (ITU); it coordinates standards for telecommunications. ITU-T G718.0 is a lossless voice signal compression software which is used to compress G.711 bitstream. The purpose of the compression is mainly for transmission over IP (e.g., VoIP). The input and output of the benchmark is a binary file indicating the original and compressed voice signal, respectively [51].

- **GSAD** benchmark, which is an abbreviated form of *Generic Sound Activity Detector*, is an independent front-end processing module that can be
used to detect whether a transmission voice line is busy or not. In other words, it indicates whether the input frame is a silence or an audible noise frame. The input format of this benchmark is also a binary file.

In order to evaluate the constructed model and compare the results with a similar approach, we use the Accuracy criterion, which is calculated as equation 5, where $TP$, $TN$, $FP$, and $FN$ are True Positive, True Negative, False Positive and False Negative, respectively.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN} \times 100$$  \hspace{1cm} (5)

### 4.2 Experiment Results and Discussion

In this section, we investigate the impact of different parameters on the accuracy of the constructed oracles. These parameters are the percentage of passing test cases in the training dataset, the code coverage percentage of the training dataset, the size of the training dataset, and the configuration parameters of ANN. Samples of input data, the related outcomes as training set and the results of the algorithm can be accessed from [http://ticksoft.sbu.ac.ir/upload/samples.zip](http://ticksoft.sbu.ac.ir/upload/samples.zip).

#### 4.2.1 Percentage of Passing Test Cases in Training Dataset

The training dataset in our approach contains input values and the corresponding pass/fail outcome of the SUT, which is used to train the ANN classifier. The number of passing test cases in comparison to the number of all test cases of the training dataset may affect the accuracy of the classifier during the testing phase. We use the term ‘ratio’, defined in equation 6, as the under study parameter in this section.

$$\text{ratio} = \frac{\text{No. of passing test cases}}{\text{No. of test cases}} \times 100$$  \hspace{1cm} (6)

In order to carry out the experiment for each benchmark, we have generated different training datasets with different ratios. Then for each benchmark, we have constructed ANN classifiers using the existing training datasets. In the testing phase, new test cases are randomly generated to evaluate the accuracy of the classifier as test oracle. Fig. 9 illustrates the accuracy of the test oracle for each benchmark, over training datasets with different ratios.

According to the diagrams of Fig. 9, the highest accuracy belongs to the training datasets with ratio of 50%, which means half of the training dataset contains pass labels. It could be seen that, if the portion of passing test cases in the training dataset is more than failing ones, or vice versa, the accuracy decreases. The reason is, when the pass test cases are more than the fail ones, $TP$ has high value but $TN$ has small value in equation 5. Therefore, the accuracy
decreases overall. The same thing occurs when the failing test cases are more than the passing ones.

In real-world systems more test cases are typically labeled as pass rather than fail. Therefore, in order to show the usefulness of the proposed approach in real-world systems, we have carried out the rest of the experiments using training datasets which contain 90% pass labels.

4.2.2 Code Coverage Percentage of Training Dataset

Code Coverage Percentage (CCP) is a measure that shows the percentage of the executed code when a particular set of test data is given to the program. In test data generation, which is one of the main activities in the software testing process, CCP is a criterion for assessing the adequacy of the generated test data. In this paper, we define CCP as the percentage of visited statements to all statements (see equation \(7\)). Since generating our classifier-based oracle severely depends on the test input dataset, it is expected that the CCP of the training dataset is an appropriate indicator of the adequacy of the constructed oracle. In this section, we investigate the impact of CCP of the training dataset on the accuracy of the constructed oracles.

\[
CCP = \frac{\text{No. of visited statements}}{\text{No. of all statements}} \times 100 \quad (7)
\]

In order to do the experiment for each benchmark, we generate different training datasets with different CCPs. Then for each benchmark, we create ANN classifiers using training datasets. In the testing phase, test data are generated randomly to evaluate the accuracy of the classifier as test oracle. Fig. 10 illustrates the accuracy of test oracle for each benchmark, over training dataset CCP.

It is worth noting that we are not able to have arbitrary CCP for the training dataset, because some statements of the code are always executed and consequently for each benchmark in Fig. 10 the diagrams start from different points. By applying training dataset with CCP of 20% to 40%, the accuracy of the proposed oracle is between 65% and 80%, which is reasonably acceptable.

Nevertheless, as we can see in Fig. 10 the higher the CCP of the training dataset is, the more accurate test oracle is generated. This is because the training dataset covers more parts of the SUT, and consequently, the oracle can model the more parts of the SUT. Therefore, we use training datasets with maximum possible CCP measure for each benchmark in the rest of the experiments. This decision is acceptable because in the test data generation phase of real-world projects, it is tried to generate test data with maximum code coverage.

4.2.3 The Size of Training Dataset

One of the other factors that have impact on the accuracy of the test oracle is the training dataset size, which means the number of test cases used in the training
In order to carry out the experiment for each benchmark, we have generated different training datasets in different sizes but with a fixed ratio and coverage according to the consequences in sections 4.2.1 and 4.2.2. The ratio has been set to 90% for all training datasets, and the CCP measure has been set to 98.8%, 100%, 100%, 100% and 100% for benchmarks Scan, TCAS [54], DES [50], ITU-T G718.0 [51] and GSAD [52], respectively. Then, for each benchmark, we created ANN classifiers using training datasets.

In the testing phase, test cases are generated randomly to evaluate the accuracy of the classifier as test oracle. Fig. 11 illustrates the accuracy of the test oracle for each benchmark, over the training dataset sizes. According to the diagrams, the more the number of test cases is in the training dataset, the more accurate is the oracle because the weights of the ANN edges are adjusted more precisely. It is worth mentioning that when the size of the training dataset becomes larger than a specific value, the accuracy of the constructed oracle will be decreased. This is because the model is biased to the training dataset, and therefore, classifies the testing dataset in a wrong way, which is usually called 'overfitting'.

4.2.4 Configuration Parameters of ANN

Oracles constructed by the proposed approach are Artificial Neural Networks, which model the behavior of SUTs. Therefore, the topology of ANNs has a remarkable impact on the accuracy of the constructed oracles. The number of hidden layers as well as the number of neurons in each layer are the most important parameters of ANN, which are studied in our experiments. Choosing larger values for these parameters, by considering appropriate epoch value, may lead to more accurate ANN. But the training time will increase in this condition. Therefore, it is necessary to select appropriate values for these parameters in order to save time and cost, while achieving the best accuracy at the same time. Nevertheless, it is obvious that for a more complex SUT, a larger ANN is required (in terms of the number of hidden layers and the number of neurons in each layer).

By fixing the other parameters based on the results obtained in sections 4.2.1, 4.2.2 and 4.2.3, we have conducted the experiments with different number of hidden layers and various number of neurons in each layer. Fig. 12(a) and 12(b) illustrate the accuracy of the constructed oracle for each benchmark, over the number of hidden layers and the number of neurons in each layer, respectively. As we can see in both figures, when the value of each parameter is small, the accuracy is low because the size of the ANN is not large enough to model the complexity of the SUT as required; and when the value of each parameter exceeds from a specific point, the accuracy approaches a certain value, which is not necessarily maximum.

Generally, when the number of hidden layers and the number of neurons in each layer increase, the weights of network edges are not changed significantly
with the epoch of 1000 (According to section 3.2). Therefore, the accuracy de-
creases. This is a kind of trade off between time and accuracy, as mentioned
above. The configuration of ANN for each benchmark is illustrated in Table 3.

It is worth mentioning that the input type of the GSAD and ITU-T G718.0
benchmarks is binary file. Therefore, each binary character is considered as
a single input. In order to reduce the number of neurons in the input layer
of the ANN, we separated the binary values 4 by 4 bytes. To this end, we
converted the binary file into a set of integer inputs. These inputs were fed to
the ANN instead of binary characters, which consequently reduced the size of
the constructed models.

4.2.5 Comparison with the Baseline Method

In this section, we compare the accuracy of the oracles generated by the proposed
approach with those constructed by a known approximation based method, sug-
gested by Shah-Amiri et. al [1] which generates oracles by modelling the rela-
tionship between program’s inputs and output values using neural networks.
These oracles are only appropriate for particular types of software that have
crude numerical outputs.

To compare these two types of oracles, the experiments were conducted on
Scan and TCAS benchmarks. We compare the accuracy of the two types of
oracles over different sizes of the training dataset. Fig. 13 illustrates the effect
of the dataset’s size on the accuracy of the two oracle types.

In this figure, the gray lines show the accuracy of our oracles when we have
the best training dataset (in terms of the pass ratio and CCP), and the blue
lines show the accuracy of our oracles when the training dataset is the same as
the one used in [1]. The blue line in Fig. 13(a) shows the accuracy of oracles
constructed by our approach when the CCP measure and the pass ratio of the
training dataset are 61% and 90%, respectively. The gray line is for the time
that the CCP measure and the pass ratio of the training dataset are 98.8% and
50%, respectively. The blue line in Fig. 13(b) shows the accuracy of oracles
constructed by our approach when the CCP measure and the pass ratio of the
training dataset are 50% and 90%, respectively. The gray line is for the time
that the CCP measure and the pass ratio of the training dataset are 100% and
50%, respectively.

The decrement of accuracy in our oracles from a specific point is because
of the overfitting problem. Nevertheless, in general, the accuracy of oracles
generated by the method in [1] is less than ours according to Fig. 13. In
addition, when the size of the training dataset is zero (which means without
having the training phase), our proposed approach works randomly with the
accuracy about 50%, because we have two classes of pass and fail. In contrast,
the accuracy of the method in [1] is zero, since it is an approximator rather than
a classifier, which means it is not able to produce any result even randomly.
5 Conclusion and Future Work

Building automated oracles is challenging for embedded software which has low observability and/or produces un-structured or semi-structured outputs. In this paper, we have proposed a classifier based method using ANNs, which addresses the mentioned issue. In the proposed approach, oracles need input data tagged with two labels of "pass" and "fail" rather than outputs and any execution trace. Also, unlike some oracles such as \[1,13\], the comparison between expected results and actual outputs is not required in our approach. Therefore, it can be applied to a wide range of software systems, including embedded software. The experimental results on five benchmarks, which three of them are categorized in embedded software systems, manifest the capability of the proposed approach in constructing accurate oracles for such systems.

As the other results of the experiments, we achieved the following items:

- When the percentage of pass labels in the training dataset is 50%, the accuracy of the constructed oracle reaches to its highest value in comparison with other percentages. Since having training datasets with pass ratio of 50% is unreasonable for real-world systems, we considered pass ratio of 90%. It is worth noting that in this situation the constructed oracles indicated acceptable accuracy, as well.

- When the code coverage percentage (CCP) of the training dataset is high, more accurate test oracle is generated. The reason is that, the training dataset covers more parts of the SUT. Fortunately, in the test data generation phase of real-world projects, it is tried to generate test data with maximum code coverage. Therefore, the results are promising to achieve an acceptable accuracy in real situations.

- When the size of the training dataset increases, the accuracy of the constructed oracle also increases because the weights of the edges in the ANN become more accurate by using a larger dataset. However, when the size becomes more than a specific value, the accuracy will be decreased because the model is biased to the training dataset, or in other words, overfitting occurs.

- The configuration parameters of ANN depends on the code complexity of the SUT. The larger the ANN is, the more the time is needed for its convergence. So, selecting parameters is a trade off between time and the accuracy of the oracle. We used the method of 'try and error' to achieve appropriate values for the ANN parameters per benchmark program.

The proposed approach is based on the black-box testing. Therefore, we do not require the program's source code, execution traces or design documents of the SUT to generate an automated oracle. In addition, unlike the mentioned oracles in section \[2\] there is no need to execute the SUT in order to achieve its outputs, which is advantageous specially in testing embedded software and safety-critical applications. Moreover, unlike the majority of machine learning-based oracles,
which do not consider the failing patterns of the code during model construction, our proposed approach considers both passing and failure-inducing inputs for model construction. In this way, our model reflects the whole behavior of the SUT, and thus, it shows more accuracy. At last, the experimental results of the comparison between our approach and the machine learning based oracle proposed in [1], revealed the fact that our approach is at least as good as the approach in [1] in common cases, although our method is applicable in some cases that the method in [1] is not.

In our black-box approach, we have assumed that we have no access to the program’s code. Assuming that we have access to the software code, we can construct more robust oracles. For future works, we are planning to study the impact of the analysis of the source code on the accuracy of the oracle. We would also intend to investigate the effect of different machine learning techniques on the precision of the constructed oracle. As another future work, we will consider different metrics of code complexity to examine the effect of this complexity on the topology of ANN [?].

References


List of Captions

Figure 1: The overall view of a conventional test oracle
Figure 2: A block diagram of an ML-based test oracle
Figure 3: The process of building and using an automated test oracle based on our approach
Figure 4: A biological neural cell
Figure 5: Node structure in ANN
Figure 6: Commonly used activation functions
Figure 7: An example of feed-forward neural network
Figure 8: The flowchart of the proposed approach
Figure 9: The accuracy of the constructed oracle over different ratios of the training dataset
Figure 10: The accuracy of the constructed oracle over CCP of the training dataset
Figure 11(a): SCAN
Figure 11(b): TCAS
Figure 11(c): DES
Figure 11(d): ITU-T G718.0
Figure 11(e): GSAD
Figure 11: The impact of the size of the training dataset on the accuracy of the constructed oracle for each benchmark
Figure 12(a): The accuracy of the constructed oracle for each benchmark, over the number of hidden layers
Figure 12(b): The accuracy of the constructed oracle for each benchmark, over the number of neurons in each layer
Figure 12: The impact of the parameters of ANN configuration on the accuracy of the constructed oracle
Figure 13(a): The accuracy for the Scan benchmark
Figure 13(b): The accuracy for the TCAS benchmark
Figure 13: The comparison between our approach and the method of [1]
Table 1: Decision table template
Table 2: Features of benchmarks
Table 3: Configuration of ANN for each benchmark
List of Figures and Tables

Figure 1

Figure 2

Figure 3

Figure 4
Weighted sum of inputs

Weights

Inputs

Activation Function

Output

\[ f(s) = \sum_{i} w_i x_i \]

Figure 5

Unit Step
\[ g(z) = \begin{cases} 
1 & \text{if } z \geq 0 \\
-1 & \text{otherwise} 
\end{cases} \]

Logistic (Sigmoid)
\[ g(z) = \frac{1}{1 + \exp(-z)} \]

Hyperbolic Tangent (Sigmoid)
\[ g(z) = \frac{\exp(2z) - 1}{\exp(2z) + 1} \]

Linear
\[ g(z) = z \]

Figure 6

Figure 7
Generate input data or use the available test suite

Using available resources to label each run as pass/fail

Data Preparation Phase

Create ANN

Initialize ANN weights

Calculate error

Error < Predefined value

Back-Propagation and update weights

Iteration < Max

Final model

Training Phase

Figure 8
Figure 11
Figure 12
Figure 13

Table 1

<table>
<thead>
<tr>
<th>Variant</th>
<th>Input Section</th>
<th>Output Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input Variables</td>
<td>Input Actions</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

34
Table 2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number Of Inputs</th>
<th>Number Of Lines</th>
<th>Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>8</td>
<td>70</td>
<td>Java</td>
</tr>
<tr>
<td>TCAS</td>
<td>12</td>
<td>173</td>
<td>C</td>
</tr>
<tr>
<td>DES</td>
<td>2</td>
<td>330</td>
<td>C++</td>
</tr>
<tr>
<td>ITU-T G718.0</td>
<td>1</td>
<td>356</td>
<td>C and VHDL</td>
</tr>
<tr>
<td>GSAD</td>
<td>1</td>
<td>411</td>
<td>C, Verilog and VHDL</td>
</tr>
</tbody>
</table>

Table 3

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Neurons in the Input Layer</th>
<th>Number of Hidden Layers</th>
<th>Number of Neurons in Each Hidden Layer</th>
<th>Number of Neurons in the Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>TCAS</td>
<td>12</td>
<td>13</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>DES</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>ITU-T</td>
<td>121</td>
<td>4</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>GSAD</td>
<td>145</td>
<td>4</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>
Meysam Valueian, is Ph.D. student in department of Computer Engineering in Sharif University of Technology. He received his BS and MS degree from Faculty of Computer Science and Engineering at Shahid Beheshti University. Software testing and repair are among his research interests.

Niousha Attar received her BS and MS degree from Faculty of Computer Science and Engineering at Shahid Beheshti University. She is currently PhD student in that faculty. Her research interests are complex networks and software testing.

Hassan Haghighi received his PhD degree in Computer Engineering-Software from Sharif University of Technology, Iran in 2009 and is currently an Associate Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include formal methods in the software development life cycle, software testing, and software architecture.

Mojtaba Vahidi Asl is Assistant Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. He received his BS in Computer Engineering from AmirKabir University of Technology, MS and PhD degree in Software Engineering from Iran University of Science and Technology. His research area includes software testing and debugging and Human computer interaction.