Scientia Iranica D (2020) 27(3), 1333–1351

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
http://scientiairanica.sharif.edu

Constructing automated test oracle for low observable software

M. Valueian[1], N. Attar, H. Haghighi, and M. Vahidi-Asl*

Faculty of Computer Science and Engineering, Shahid Beheshti University, G.C, Tehran, P.O. Box 1983963113, Iran.

Received 27 July 2018; received in revised form 26 January 2019; accepted 10 August 2019

KEYWORDS
Software testing;
Test oracle;
Machine learning;
Artificial neural network;
Software observability.

Abstract. The application of machine learning techniques for constructing automated test oracles has been successful in recent years. However, existing machine learning based oracles are characterized by a number of 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 that can be applied to software systems with low observability. The proposed approach employs an Artificial Neural Network algorithm that uses input values and corresponding pass/fail outcomes of the program under test as the training set. To evaluate the performance of the proposed approach, extensive experiments were carried out on several benchmarks. The results manifest the applicability of the proposed approach to software systems with low observability and its higher accuracy than a well-known machine learning based method. This study also assessed the effect of different parameters on the accuracy of the proposed approach.

1. Introduction

Due to the considerable size, complexity, distribution, pervasiveness, and criticality of recent software systems, producing faultless software seems to be an unattainable dream. Furthermore, the increasing demand for software programs and competition in software industry lead to 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 to improve 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 generations that aims to create an appropriate subset of input values to determine whether a program produces the intended outputs. The input values should satisfy some testing criteria 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

1. *Present address: Department of Computer Engineering, Sharif University of Technology, Tehran, Iran.*
*. *Corresponding author. Tel.: +98 21 29904131*
E-mail addresses: valueian@ce.sharif.edu (M. Valueian); n_attar@sbu.ac.ir (N. Attar); h_haghighi@sbu.ac.ir (H. Haghighi); mo_vahidi@sbu.ac.ir (M. Vahidi-Asl)

doi: 10.24200/sci.2019.51494.2219

research has been dedicated to automating 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 facing 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 of manual test oracles is the diversity and complexity of software systems and platforms, each of which has various types of input parameters and output data. Moreover, 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 on 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. Figure 1 illustrates the structure of a conventional automated test oracle. As shown in the figure, the test inputs are given to 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 the full formal specification of the SUT features. In these situations, this can use a partial test oracle that can assess the program behavior with respect to a subset of test inputs [4]. One approach to producing a partial oracle is the application of metamorphic testing, i.e., building the test oracle based on known relationships between expected behaviors of the SUT.

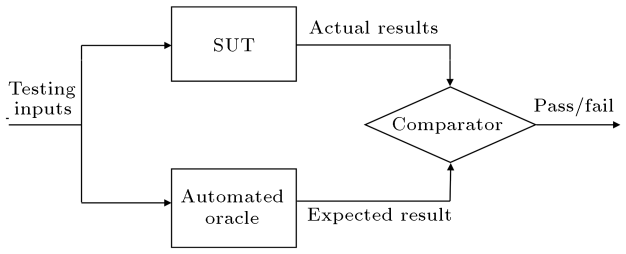

Figure 1. The overall view of a conventional test oracle.

Recently, Machine Learning (ML) methods have been employed to build test oracles. A typical ML-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 Figure 2. According to the figure, the constructed model reflects the correct behavior of the SUT assuming that the model is of 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, are 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; otherwise, the execution is considered to be failed. The choice of the ML 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 is applicable to 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 the 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) that limits the observability of the software [9]. In these

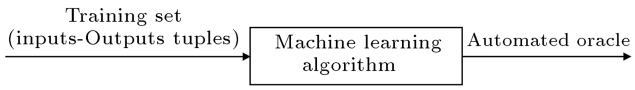

Figure 2. A block diagram of a Machine Learning (ML) based test oracle.

situations, assuming that the hardware works correctly, most of the testers prefer to evaluate 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. Graphical User Interface (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. However, their encoding to numerical values is a challenging process. In these situations, testers usually label the outputs with pass/fail by 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 much simpler.

There are also some other situations where the existing methods are not applicable. For instance, consider the cases in which explicit historical data, represented as input-output pairs, are not available, while there are documents that inform us about failure-inducing inputs or scenarios. The failure-inducing inputs refer to a subset of input values that cause 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 that 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, containing valuable pass/fail scenarios.

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

Based on the above idea, in a conference paper [12], a learning based approach is employed, 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 that 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. Figure 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] is characterized by the following advantages:

- It is based on black-box testing. This means that one needs neither the program source code nor the design documents of the SUT to generate an automated oracle;
- Regarding the testing phase of the proposed approach, shown in Figure 3, there is no need to execute the SUT to compare its output with the oracle's output; as mentioned earlier, 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 and cost consuming or risky work. These considerations are usually seen in many low observable and safety-critical systems;
- The test oracles presented in [1,13] are approximators rather than classifiers. They assume that each of the output components in the input output pairs given to the SUT in 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

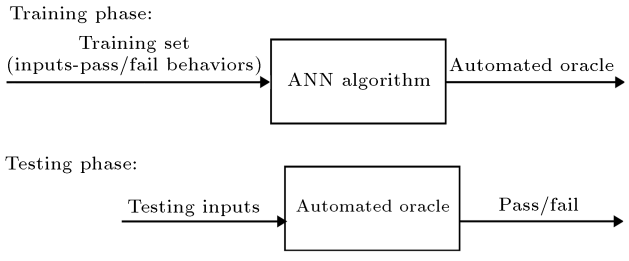

Figure 3. The process of building and using an automated test oracle based on the proposed approach.

contrast, since the 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 given below:

- This study conducts 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], benchmarks are only considered 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, this complexity decreases by reducing the size of binary files for the classifier. In order to demonstrate the capability of the proposed approach to programs with input signals, three related programs are added to our benchmarks;
- Literature review is updated according to the studies that have been done in recent years.

The remaining parts of the paper are organized as follows. Section 2 presents a review of related works and a brief background of neural networks needed to read the paper. Section 3 describes the details of the proposed approach. Section 4 gives the experimental results and analysis. Section 5 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 postconditions define effects of the operation on the program state. Peters and Parnas 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 include abstract sets of concrete states of the modeled system. Gargantini and Riccobene applied 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. that may cause programs to crash or show 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 distinguished between intended and incorrect behaviors 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 produce a responsive layout failure. According to the empirical study in [19], their approach detected 33 distinct failures in 16 out of 26 real-world web pages.

There are some approaches that 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 an oracle for these kinds of programs is the state explosion problem. Therefore Carver and Lei [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 through 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. Simons has constructed a lazy systematic unit-testing tool called JWALK [22], in which the specification of a system is lazy and eventually learned by the interaction 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]. Moreover, 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; however, their actual behaviors might be different because of a faulty 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 were equivalent to a target method within a finite set of execution scenarios [29]. 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 is implied 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 runtime 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 that must be held during program execution. For example, a loop invariant is a condition that is true at the beginning and end and in each iteration of the loop. Invariants can be included in test oracles considering that invariant detection is an essential part of this method. Ernst et al. proposed a ML-based technique to detect invariants [31]. They implemented the Daikon tool for this purpose [32].

Elyasov et al. [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 compared to 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 for one implementation to be 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 [34].

As another solution to the oracle problem, the notion of decision table was used in [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 are 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 the 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 automated planning to generate oracle models for GUIs [36]. Their models include two parts: Expected-state generator and verifier. In order to generate expected-state, they used a formal

Table 1. Decision table template [55].

Variant					
Input section			Output section		
Input variables	Input actions	State before test	Expected results	Expected output sections	Expected state after test
...			...		

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 could be employed for recovering 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 that 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 utilized code predicates to recognize input features to construct a decision tree. They applied this approach to the triangle benchmark program, where the leaves of the tree are labeled with the triangle classes (equilateral triangles, isosceles triangles, scalene triangles, and invalid triangles).

Wang et al. utilized Support Vector Machine (SVM) as a supervised machine 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 software [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 input-output 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 program 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 the method can be used for different types of software.

Almaghairbe and his colleagues [43] proposed test oracles by clustering failures. They utilized anomaly detection techniques using software's input/output pairs. They did the experiments based on the assumption that failures tended 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. Moreover, Almaghairbe and Roper applied 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 extracting iterative patterns from execution traces. In addition, Yilmaz and Porter [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 to test a program's decision-making structures [45] and verify 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, train ANNs, and test the constructed ANN as an automated oracle. In [1], Shahamiri et al. [46] 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 was constructed for each output. Therefore, the

complexity of the SUT was distributed over several ANNs, and the ultimate oracle was 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 ML-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 to be correct. Otherwise, it would be labeled as failure. In addition, some of the existing approaches, such as [6–8], consider outputs as input features of the ML 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; however, the proposed-method in this paper does not need to execute the SUT at least in the testing phase.

The proposed approach overcomes these deficiencies: It applies to systems that have low observability and/or produce unstructured or semi-structured outputs. Furthermore, since we considered 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. Therefore, there is no need to perform the software, which is a costly and sometimes risky operation. Moreover, the proposed test oracle is built according to both the correct and incorrect behaviors of the SUT.

2.2. Artificial Neural Network (ANN)

In recent years, a wide variety of ML methods have been proposed that can be used to discover hidden knowledge from data. Among many categories of industrial-strength ML algorithms, ANN has attracted much attention over past few years. According to [47], an ANN is a computational system that consists of elements, called units or nodes, whose functionality is based on biological neurons. As illustrated in Figure 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 another.

Furthermore, each neuron receives thousands of signals from other neurons. Eventually, these incoming signals reach 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. Moreover, 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", the result of which is the output of the node. The structure of each node in ANN is illustrated in Figure 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,

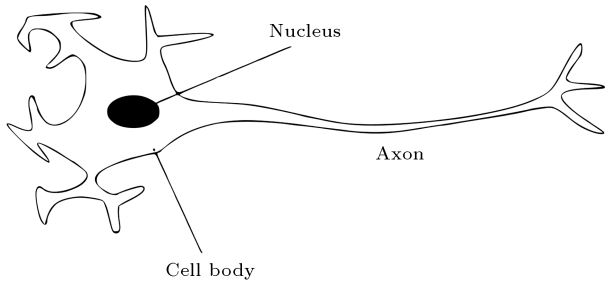

Figure 4. A biological neural cell.

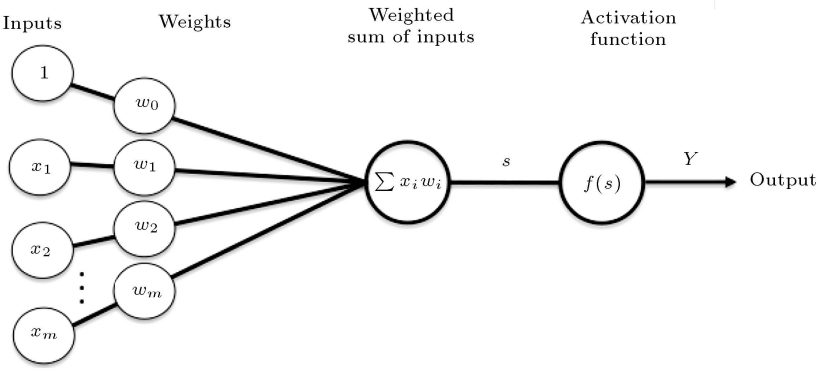

Figure 5. Node structure in Artificial Neural Network (ANN).

activation functions are applied. Figure 6 illustrates some activation functions that are useful in practice.

ANN can solve complex mathematical problems such as stochastic and nonlinear 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, ANN is used as our classifier in the rest of this paper. In the following subsection, the type of the neural network used in the experiments is described.

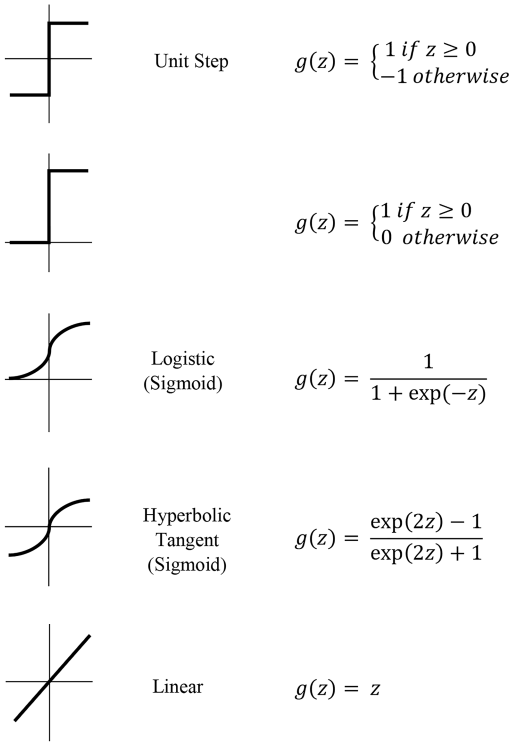

Figure 6. Commonly used activation functions.

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. Figure 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 whose 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 and 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

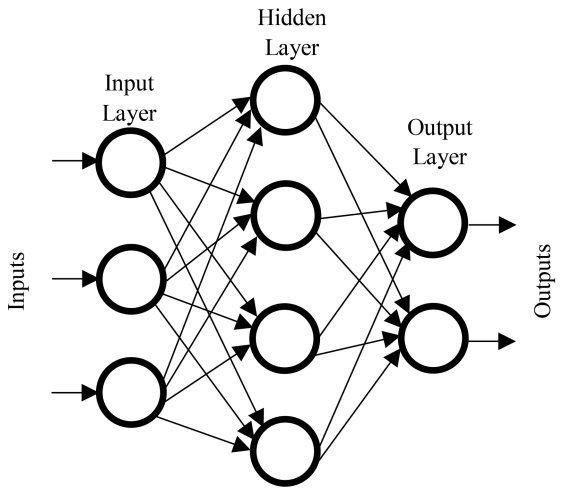

Figure 7. An example of feed-forward neural network.

are called single layer perceptron and multi layer perceptron, respectively.

As mentioned in the previous section, the weights that are associated with inputs of a specific node illustrate 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 the desired output. The process of assigning optimal weights to the connections is called the training phase of ANN. In the next subsection, the algorithm that is utilized to train ANN is introduced 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, meaning that it learns from labeled training data. This means that 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 the 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 ML-based test oracles, a binary classifier has been applied 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 that have already labeled a subset of program inputs as pass/fail according to the program 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, attempt is made 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, various datasets are considered 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 that are detailed in the remaining parts of this section. The flowchart of the approach is shown in Figure 8.

3.1. Data preparation

The collected dataset includes inputs and the corresponding execution outcomes. Suppose that the SUT has n input parameters. The inputs can be organized as an input vector X, shown by Eq. (1), where feature x_i represents the ith input parameter of the SUT with n input parameters:

$$X =< x_1, x_2, x_3, \cdots, x_n > . \tag{1}$$

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

$$T = D(x_1) \times D(x_2) \times \cdots \times D(x_n). \tag{2}$$

The number of possible input vectors is the product of the domain size of the input parameters as:

$$|T| = |D(x_1)| \times |D(x_2)| \times \cdots \times |D(x_n)| = \prod_{i=1}^{n} |D(x_i)|. \tag{3}$$

The SUT is executed by each input vector, and the resulting outcomes/behaviors of the execution are labeled with the members of set C as:

$$C = \{Fail, Pass\}. \tag{4}$$

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_j^i, $1 \le i \le n$ and $1 \le j \le m$, is the value of input parameter i from input vector j, as shown below:

$$\begin{pmatrix} x_1^1 & x_1^2 & \cdots & x_1^n \\ x_2^1 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \ddots & \vdots \\ x_m^1 & x_m^2 & \cdots & x_m^n \end{pmatrix}.$$

2. An $m \times 1$ outcome vector, where $c_k \in \{0,1\}$, $1 \le k \le m$, and $c_k = 1$, indicates that the result of the SUT subjected to the input data vector $< x_k^1, x_k^2, \cdots, x_k^n >$ is failure; otherwise, it is a pass. The vector is given as follows:

$$\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}.$$

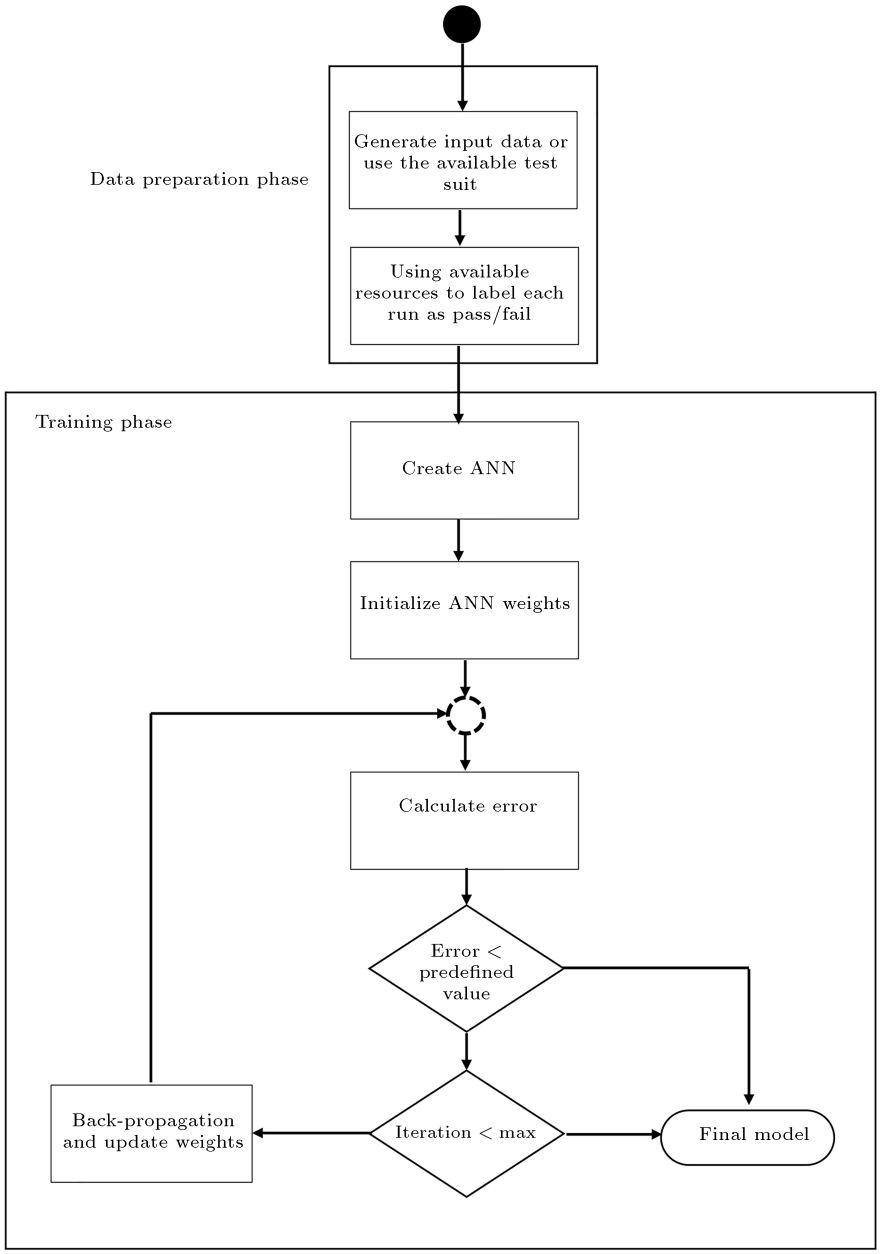

Figure 8. The flowchart of the proposed approach.

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, meaning that 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 it to this network. A multilayer perceptron ANN 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 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 Figure 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 Figure 8, then the training phase will be stopped. Otherwise, it will continue until the number of iterations meet the epoch number. As illustrated in Figure 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, the training dataset is applied to 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 CCP 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, the experimental setup is described and, then, the results of the experiments are presented and analyzed.

4.1. Experiment setup

To evaluate the proposed method, this study 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 2.70 GHz up-to 2.90 GHz, 8.0 GB RAM", and the operating system was "Windows 10 Pro 64-bit".

Five benchmarks with different features and characteristics have been considered. 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 these benchmarks have been considered as embedded programs is according to the definition presented by Lee [53]. According to this definition, embedded programs could have interactions with physical devices. They are not necessarily executed 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 to programs with low observability and to 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 ML-based method, proposed in [1]. From this point forward, we call the method in [1] the baseline method. It is worth noting that the baseline method is not applicable to low observable programs and programs with non-numerical outputs. Table 2 illustrates the features of the selected

Table 2. Features of benchmarks.

Benchmark	Number of inputs	Number of lines	Programming language
Scan	8	70	Java
TCAS	12	173	C
DES	2	330	C++
ITU-T G718.0	1	356	C and VHDL
GSAD	1	411	C, Verilog and VHDL

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 bench mark represent a binary file that indicates the original and compressed voice signals, 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 [52].

In order to evaluate the constructed model and compare the results with a similar approach, the *Accuracy* criterion is used, which is calculated through Eq. (5), where *TP*, *TN*, *FP*, and *FN* are *True Positive*, *True Negative*, *False Positive*, and *False Negative*, respectively:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \times 100. \quad (5)$$

4.2. Experiment results and discussion

In this section, the impact of different parameters on the accuracy of the constructed oracles is investigated. These parameters include the percentage of passing test cases in the training dataset, the CCP 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 the link: (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. The term 'ratio', defined in Eq. (6), is used as the under study parameter in this section.

$$ratio = \frac{No.\ of\ passing\ test\ cases}{No.\ of\ test\ cases} \times 100. \quad (6)$$

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

According to the diagrams of Figure 9, the highest accuracy belongs to the training datasets at a ratio of 50%, which means that 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 that when the pass test cases are more than the fail ones, TP has a high value, but TN has a small value in Eq. (5). Therefore, the accuracy decreases overall. The same 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, the rest of the experiments have been carried out using training datasets that contain 90% pass labels.

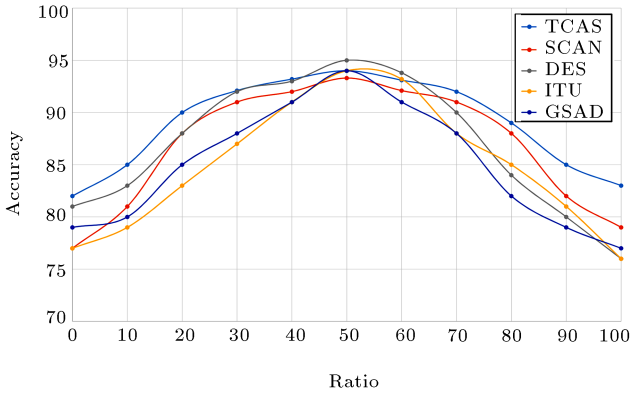

Figure 9. The accuracy of the constructed oracle over different ratios of the training dataset.

4.2.2. Code Coverage Percentage (CCP) of training dataset

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, CCP is defined as the percentage of visited statements to all statements (see Eq. (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, the impact of CCP of the training dataset on the accuracy of the constructed oracles is investigated.

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

In order to do the experiment for each benchmark, different training datasets with different CCPs have been generated. Then, for each benchmark, ANN classifiers are created using training datasets. In the testing phase, test data are generated randomly to evaluate the accuracy of the classifier as a test oracle. Figure 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 Figure 10, the diagrams start at different points. By applying training dataset with CCP of 20% to 40%, the accuracy of the proposed oracle ranges between 65% and 80%, which is reasonably acceptable.

Nevertheless, as is clear in Figure 10, the higher the CCP of the training dataset is, the more accurate the test oracle will be. This is because the training dataset covers more parts of the SUT and, consequently, the oracle can model more parts of the SUT.

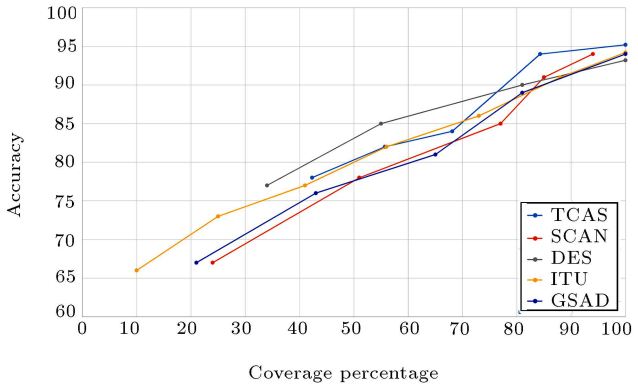

Figure 10. The accuracy of the constructed oracle over Code Coverage Percentage (CCP) of the training dataset.

Therefore, training datasets with maximum possible CCP measure for each benchmark are applied to 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 the maximum code coverage.

4.2.3. The size of training dataset

One of the other factors that has impact on the accuracy of the test oracle is the training dataset size, which means the number of test cases used in the training dataset.

In order to carry out the experiment for each benchmark, we have generated different training datasets of 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, ANN classifiers are created using training datasets.

In the testing phase, test cases are generated randomly to evaluate the accuracy of the classifier as a test oracle. Figure 11 illustrates the accuracy of the test oracle for each benchmark over the training dataset sizes. According to the diagrams, the higher the number of test cases is in the training dataset, the more accurate the oracle will be, 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 decrease. This is because the model is biased to the training dataset and, therefore, the testing dataset is classified in a wrong way, which is usually called 'overfitting'.

4.2.4. Configuration parameters of ANN

Oracles constructed by the proposed approach are ANN, 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 and 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. However, 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

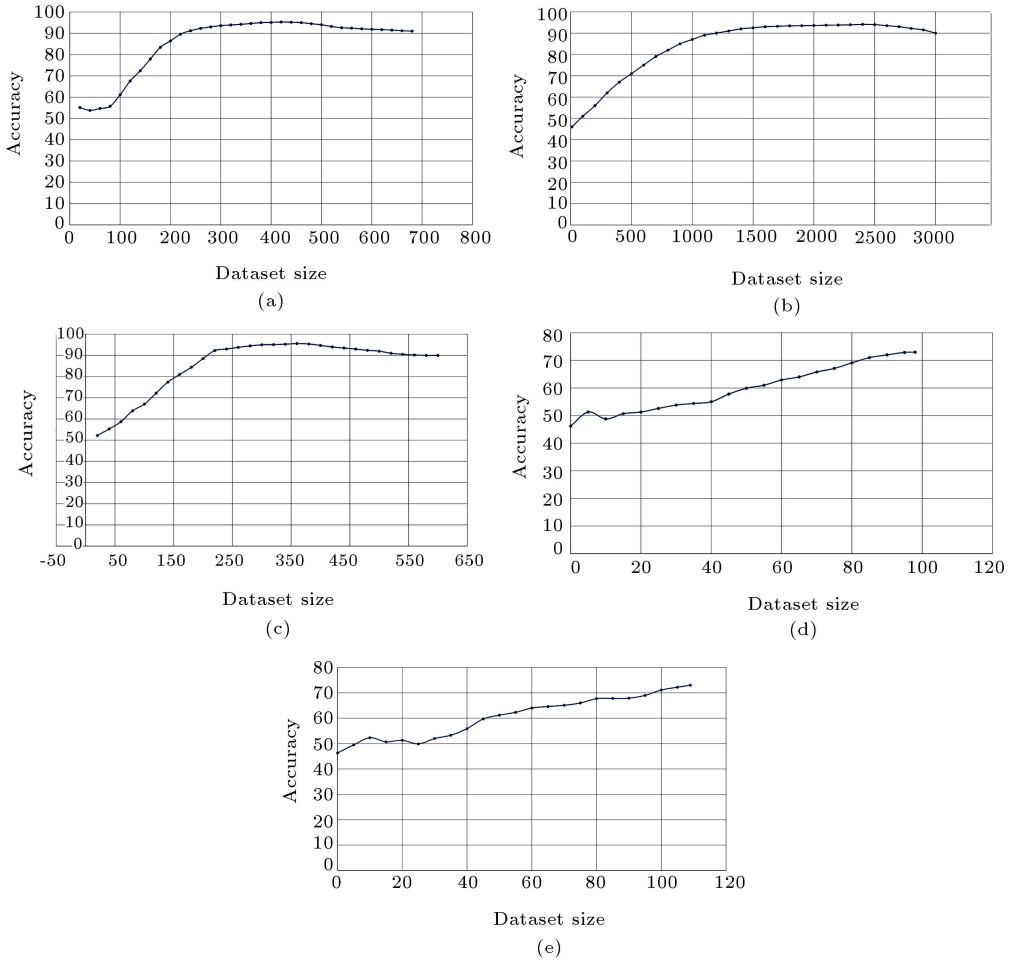

Figure 11. The impact of the size of the training dataset on the accuracy of the constructed oracle for each benchmark: (a) SCAN, (b) TCAS, (c) DES, (d) ITU-T G718.0, and (e) GSAD.

results obtained in Sections 4.2.1, 4.2.2, and 4.2.3, the experiments with different numbers of hidden layers and various number of neurons in each layer have been conducted.

Figure 12(a) and (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 is clear in both figures, when the value of each parameter is low, the accuracy is low, because the size of the ANN is not large enough to model the complexity of the SUT as required; when the value of each parameter exceeds 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 do not change significantly with the epoch of 1000 (according to Section 3.2). Therefore, the accuracy decreases. 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 a binary file.

Table 3. Configuration of Artificial Neural Network (ANN) for each benchmark.

Benchmark	Number of neurons in the input layer	Number of hidden layers	Number of neurons in each hidden layer	Number of neurons in the output layer
Scan	8	9	7	1
TCAS	12	13	7	1
DES	2	5	8	1
ITU-T	121	4	11	1
GSAD	145	4	11	1

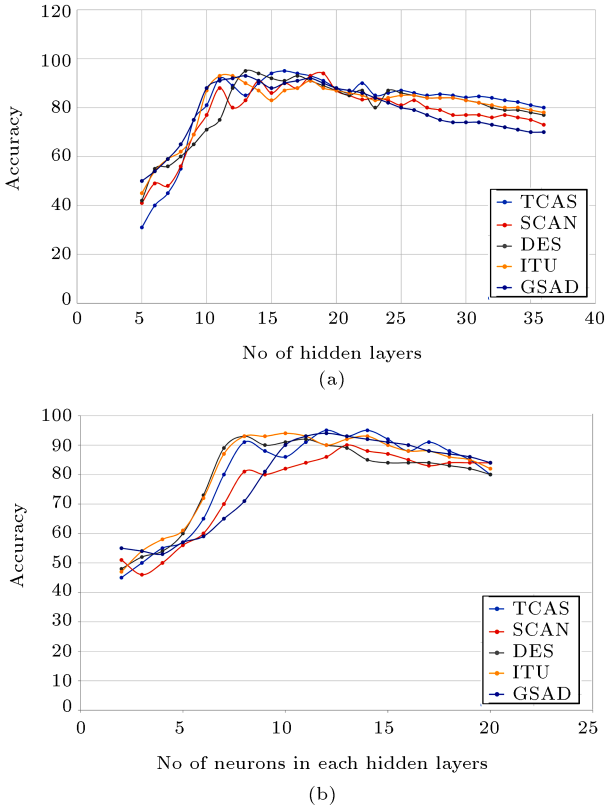

Figure 12. The impact of the parameters of Artificial Neural Network (ANN) configuration on the accuracy: (a) The accuracy of the constructed oracle for each benchmark over the number of hidden layers and (b) the accuracy of the constructed oracle for each benchmark over the number of neurons in each layer of the constructed oracle.

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, the binary values are separated 4 by 4 bytes. To this end, the binary file is converted 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

This section compares the accuracy of the oracles generated by the proposed approach with those constructed by a known approximation-based method, suggested by Shahamiri et al. [1], that generates oracles by modeling the relationship between program inputs and output values using neural networks. These oracles are only appropriate for particular types of software that produce concrete numerical outputs.

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

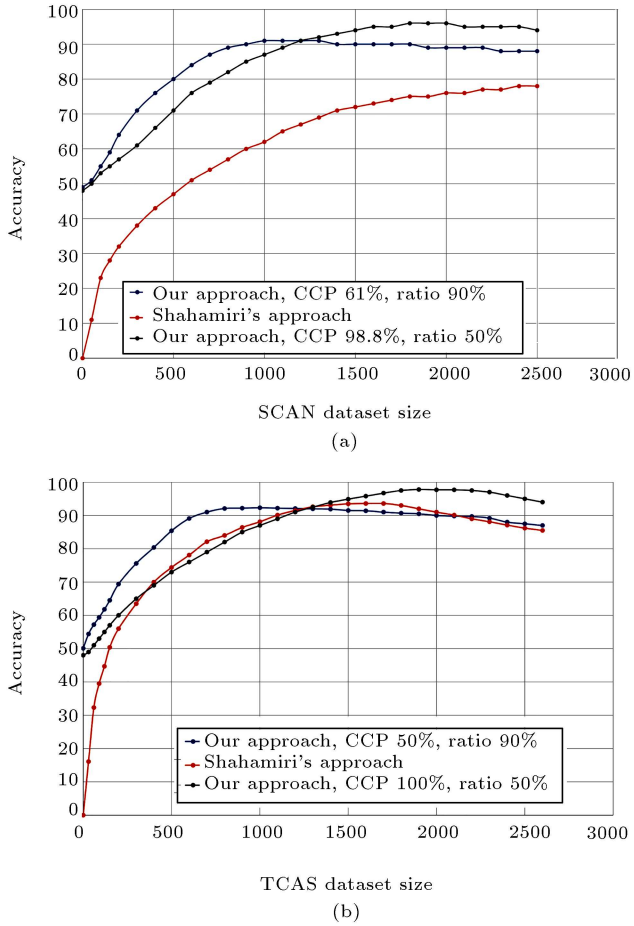

Figure 13. The comparison between our approach and the method of Shahamiri et al. [1]: (a) The accuracy for the Scan benchmark and (b) the accuracy for the TCAS benchmark.

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 Figure 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 considered when the CCP measure and the pass ratio of the training dataset are 98.8% and 50%, respectively. The blue line in Figure 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 considered when 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 Figure 13. In addition, when the size

of the training dataset is zero (which means without having the training phase), the 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 that it is not able to produce any result even randomly.

5. Conclusion and future work

Building automated oracles is challenging for embedded software with low observability and/or produces un-structured or semi-structured outputs. In this paper, a classifier based method using Artificial Neural Networks (ANNs) was proposed, which addressed 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. Moreover, 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 of five benchmarks, three of which 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, the following items are achieved:

- When the percentage of pass labels in the training dataset is 50%, the accuracy of the constructed oracle reaches its highest value in comparison with other percentages. Since having training datasets at a pass ratio of 50% is unreasonable for real-world systems, then this study considered a 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, a more accurate test oracle is generated. The reason is that the training dataset covers more parts of the Software Under Test (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 of achieving an acceptable accuracy in real situations are promising;
- 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 decrease because the model is biased to the training dataset, or in other words, overfitting occurs;
- The configuration parameters of ANN depend on the code complexity of the SUT. The larger the ANN is, the longer the time is needed for its convergence. Thus, selecting parameters is a tradeoff between time and the accuracy of the oracle. This study 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, this study does not require the program 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, especially 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, the proposed approach considers both passing and failure-inducing inputs for model construction. In this way, the proposed model reflects the whole behavior of the SUT and, thus, shows greater 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 this approach is at least as good as the approach in [1] in common cases, although our method is applicable to some cases to which the method in [1] is not.

In the proposed black-box approach, it was assumed that there was no access to the program 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

1. Shahamiri, S.R., Wan-Kadir, W.M., Ibrahim, S., and Hashim, S.Z.M. "Artificial neural networks as multi-networks automated test oracle", *Automated Software Engineering*, **19**(3), pp. 303–334 (2012).
2. Valizadeh, M., Tadayon, M., and Bagheri, A. "Making problem: A new approach to reachability assurance in digraphs", *Scientia Iranica*, **25**(3), pp. 1441–1455 (2018).
3. Rezaee, A. and Zamani, B. "A novel approach to automatic model-based test case generation", *Scientia Iranica*, **24**(6), pp. 3132–3147 (2017).
4. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. "The oracle problem in software testing: A

survey", *IEEE Transactions on Software Engineering*, **41**(5), pp. 507–525 (2015).

5. Ammann, P. and Offutt, J., *Introduction to Software Testing*, Cambridge University Press (2016).

6. Almaghairbe, R. and Roper, M. "Automatically classifying test results by semi-supervised learning", In *27th IEEE Int. Symp. on Software Reliability Engineering*, pp. 116–126 (2016).

7. Lo, D., Cheng, H., Han, J., Khoo, S.-C., and Sun, C. "Classification of software behaviors for failure detection: a discriminative pattern mining approach", In *15th ACM SIGKDD int. Conf. on Knowledge Discovery and Data Mining*, pp. 557–566 (2009).

8. Yilmaz, C. and Porter, A. "Combining hardware and software instrumentation to classify program executions", In *18th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pp. 67–76 (2010).

9. Freedman, R.S. "Testability of software components", *IEEE Transactions on Software Engineering*, **17**(6), pp. 553–564 (1991).

10. Vliegendhart, R., Dolstra, E., and Pouwelse, J. "Crowdsourced user interface testing for multimedia applications", In *ACM Multimedia 2012 Workshop on Crowdsourcing for Multimedia*, pp. 21–22 (2012).

11. Jan, S.R., Shah, S.T.U., Johar, Z.U., Shah, Y., and Khan, F. "An innovative approach to investigate various software testing techniques and strategies", *International Journal of Scientific Research in Science, Engineering and Technology*, **2**(2), pp. 2395–1990 (2016).

12. Gholami, F., Attar, N., Haghighi, H., Vahidi-Asl, M., Valueian, M., and Mohamadyari, S. "A classifier-based test oracle for embedded software", In *2018 IEEE Real-Time and Embedded Systems and Technologies*, pp. 104–111 (2018).

13. Shahamiri, S.R., Kadir, W.M.N.W., Ibrahim, S., and Hashim, S.Z.M. "An automated framework for software test oracle", *Information and Software Technology*, **53**(7), pp. 774–788 (2011).

14. Peters, D.K. and Parnas, D.L. "Using test oracles generated from program documentation", *IEEE Transactions on Software Engineering*, **24**(3), pp. 161–173 (1998).

15. Gargantini, A. and Riccobene, E. "Asm-based testing: Coverage criteria and automatic test sequence generation", *Journal of Universal Computer Science*, **7**(11), pp. 1050–1067 (2001).

16. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., and Engler, D.R. "Exe: automatically generating inputs of death", *ACM Transactions on Information and System Security*, **12**(2), p. 10 (2008).

17. Shrestha, K. and Rutherford, M.J. "An empirical evaluation of assertions as oracles", In *4th IEEE Fourth Int. Conf. on Software Testing, Verification and Validation*, pp. 110–119 (2011).

18. Ricca, F. and Tonella, P. "Detecting anomaly and failure in web applications", *IEEE MultiMedia*, **13**(2), pp. 44–51 (2006).

19. Walsh, T.A., Kapfhammer, G.M., and McMinn, P. "Automated layout failure detection for responsive web pages without an explicit oracle", In *26th ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, pp. 192–202 (2017).

20. Carver, R. and Lei, Y. "Stateless techniques for generating global and local test oracles for message-passing concurrent programs", *Journal of Systems and Software*, **136**, pp. 237–265 (2018).

21. Zhou, Z.Q., Zhang, S., Hagenbuchner, M., Tse, T., Kuo, F.C., and Chen, T.Y. "Automated functional testing of online search services", *Software Testing, Verification and Reliability*, **22**(4), pp. 221–243 (2012).

22. Simons, A.J. "Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction", *Automated Software Engineering*, **14**(4), pp. 369–418 (2007).

23. Goffi, A. "Automating test oracles generation", Ph.D. Thesis, Universit'a della Svizzera italiana (2018).

24. Carzaniga, A., Goffi, A., Gorla, A., Mattavelli, A., and Pezzè, M. "Crosschecking oracles from intrinsic software redundancy", In *36th Int. Conf. on Software Engineering*, pp. 931–942 (2014).

25. Goffi, A., Gorla, A., Ernst, M.D., and Pezzè, M. "Automatic generation of oracles for exceptional behaviors", In *25th Int. Symp. on Software Testing and Analysis*, pp. 213–224 (2016).

26. Pezzè, M. "Towards cost-effective oracles", In *10th IEEE/ACM Int. Workshop on Automation of Software Test*, pp. 1–2 (2015).

27. Goffi, A. "Automatic generation of cost-effective test oracles", In *36th ACM Int. Conf. on Software Engineering*, pp. 678–681 (2014).

28. Segura, S., Fraser, G., Sanchez, A.B., and Ruiz-Cortés, A. "A survey on metamorphic testing", *IEEE Transactions on Software Engineering*, **42**(9), pp. 805–824 (2016).

29. Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M., and Tonella, P. "Searchbased synthesis of equivalent method sequences", In *22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pp. 366–376 (2014).

30. Mattavelli, A., Goffi, A., and Gorla, A. "Synthesis of equivalent method calls in guava", In *Int. Symp. on Search Based Software Engineering*, pp. 248–254 (2015).

31. Ernst, M.D., Cockrell, J., Griswold, W.G., and Notkin, D. "Dynamically discovering likely program invariants to support program evolution", *IEEE Transactions on Software Engineering*, **27**(2), pp. 99–123 (2001).

32. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., and Xiao, C. "The daikon system for dynamic detection of likely invariants", *Science of Computer Programming*, **69**(1–3), pp. 35–45 (2007).

33. Elyasov, A., Prasetya, W., Hage, J., Rueda, U., Vos, T., and Condori-Fernández, N. "Ab = ba: execution equivalence as a new type of testing oracle", In *30th Annual ACM Symp. on Applied Computing*, pp. 1559–1566 (2015).

34. Feldt, R. "Generating diverse software versions with genetic programming: an experimental study", *IEE Proceedings-Software*, **145**(6), pp. 228–236 (1998).

35. Di Lucca, G.A., Fasolino, A.R., Faralli, F., and De Carlini, U. "Testing web applications", In *Int. IEEE Conf. on Software Maintenance*, pp. 310–319 (2002).

36. Memon, A.M., Pollack, M.E., and Soffa, M.L. "Automated test oracles for GUIs", In *8th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering: Twenty-First Century Applications*, pp. 30–39 (2000).

37. Last, M., Friedman, M., and Kandel, A. "Using data mining for automated software testing", *International Journal of Software Engineering and Knowledge Engineering*, **14**(4), pp. 369–393 (2004).

38. Zheng, W., Ma, H., Lyu, M.R., Xie, T., and King, I. "Mining test oracles of web search engines", In *26th IEEE/ACM Int. Conf. on Automated Software Engineering*, pp. 408–411 (2011).

39. Vineeta, Abhishek Singhal, Abhay Bansal "Generation of test oracles using neural network and decision tree model", In *5th Int. Conf.- Confluence The Next Generation Information Technology Summit*, pp. 313–318 (2014).

40. Wang, F., Yao, L.-W., and Wu, J.-H. "Intelligent test oracle construction for reactive systems without explicit specifications", In *9th IEEE Int. Conf. on Dependable, Autonomic and Secure Computing*, pp. 89–96 (2011).

41. Vanmali, M., Last, M., and Kandel, A. "Using a neural network in the software testing process", *International Journal of Intelligent Systems*, **17**(1), pp. 45–62 (2002).

42. Zhang, R., Wang, Y.-W., and Zhang, M.-Z. "Automatic test oracle based on probabilistic neural networks", In *Recent Developments in Intelligent Computing, Communication and Devices*, pp. 437–445 (2019).

43. Almaghairbe, R. and Roper, M. "Building test oracles by clustering failures", In *10th IEEE Int. Workshop on Automation of Software Test*, pp. 3–7 (2015).

44. Almaghairbe, R. and Roper, M. "Separating passing and failing test executions by clustering anomalies", *Software Quality Journal*, **25**(3), pp. 803–840 (2017).

45. Shahamiri, S.R., Kadir, W.M.N.W., and bin Ibrahim, S. "An automated oracle approach to test decision-making structures", In *3rd IEEE Int. Conf. on Computer Science and Information Technology*, pp. 30–34 (2010).

46. Shahamiri, S.R., Kadir, W.M.W., and Ibrahim, S. "A single-network annbased oracle to verify logical software modules", In *2nd IEEE Int. Conf. on Software Technology and Engineering*, pp. 272–276 (2010).

47. Gurney, K., *An Introduction to Neural Networks*, CRC press (2014).

48. Graupe, D., *Principles of Artificial Neural Networks*, World Scientific (2013).

49. "Mathworks", https://www.mathworks.com/ (2018).

50. Standard, D.E., *Federal Information Processing Standards Publication*, National Bureau of Standards, US Department of Commerce, **4** (1977).

51. "Itu-t technical paper telecommunication standardization sector of ITU", Technical report, Transmission Systems and Media, Digital Systems and Networks Digital Sections and Digital Line System (2010).

52. "Generic sound activity detector (GSAD)", https://www.itu.int/pub/T-REC-USB-2020.

53. Lee, E., *Embedded Software*, University of California at Berkeley, Berkeley (2001).

54. Livadas, L.J.C. and Lynch, N.A. "High-level modeling and analysis of tcas", In *20th IEEE Real-Time Systems Symp.*, pp. 115–125 (1999).

55. Shahamiri, S.R., Kadir, W.M.N.W., and Mohd-Hashim, S.Z. "A comparative study on automated software test oracle methods", In *4th Int. Conf. on Software Engineering Advances*, pp. 140–145 (2009).

Biographies

Meysam Valueian is a PhD student at the Department of Computer Engineering in Sharif University of Technology, Tehran, Iran. He received his BS and MS degrees from Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. Software testing and repair are among his research interests.

Niousha Attar received her BS and MS degrees from Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. She is currently a 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, Tehran, 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 an 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 and MS and PhD degrees in Software Engineering from Iran University of Science and Technology. His research area includes software testing and debugging and human computer interaction.