Information and Software Technology 53 (2011) 774–788

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An automated framework for software test oracle

Seyed Reza Shahamiri *, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, Siti Zaiton Mohd Hashim

Department of Software Engineering, Faculty of Computer Science and Information Systems, Universiti Teknologi Malaysia, 81310 UTM Skudai, Johor, Malaysia

ARTICLE INFO

Article history:
Received 14 June 2010
Received in revised form 19 January 2011
Accepted 25 February 2011
Available online 4 March 2011

Keywords:
Software test oracle
I/O Relationship Analysis
Artificial neural networks
Mutation testing

ABSTRACT

Context: One of the important issues of software testing is to provide an automated test oracle. Test oracles are reliable sources of how the software under test must operate. In particular, they are used to evaluate the actual results that produced by the software. However, in order to generate an automated test oracle, oracle challenges need to be addressed. These challenges are output-domain generation, input domain to output domain mapping, and a comparator to decide on the accuracy of the actual outputs.
Objective: This paper proposes an automated test oracle framework to address all of these challenges.
Method: I/O Relationship Analysis is used to generate the output domain automatically and Multi-Networks Oracles based on artificial neural networks are introduced to handle the second challenge. The last challenge is addressed using an automated comparator that adjusts the oracle precision by defining the comparison tolerance. The proposed approach was evaluated using an industry strength case study, which was injected with some faults. The quality of the proposed oracle was measured by assessing its accuracy, precision, misclassification error and practicality. Mutation testing was considered to provide the evaluation framework by implementing two different versions of the case study: a Golden Version and a Mutated Version. Furthermore, a comparative study between the existing automated oracles and the proposed one is provided based on which challenges they can automate.
Results: Results indicate that the proposed approach automated the oracle generation process 97% in this experiment. Accuracy of the proposed oracle was up to 98.26%, and the oracle detected up to 97.7% of the injected faults.
Conclusion: Consequently, the results of the study highlight the practicality of the proposed oracle in addition to the automation it offers.

1. Introduction

Software testing is the process of evaluating the software behavior to check whether it operates as expected in order to improve its quality and reliability. Since the testing process is highly time and resource consuming, complete testing is almost impossible; thus, testers use automatic approaches to facilitate the process and decrease its costs [1].

Test oracle is a mechanism to determine whether an application is executed correctly. It is a reliable source of how the SUT[1] must operate [2]. It is also expected to provide correct results for any inputs that are specified by the software specifications, and a comparator to verify the actual behavior [3]. Automated test oracles are helpful in providing an adequate automated testing framework.

After test cases are executed and results of the testing are generated, it is necessary to decide whether the results are valid in order to determine the correctness of the software behavior. To verify the behavior of the SUT, correct results are compared with the results generated by the software. The results produced by the SUT that need to be verified are called *actual outputs*, and the correct results that are used to evaluate actual outputs are called *expected outputs* [2]. Test oracles are used as a complete and reliable source of expected outputs and a tool to verify the correctness of actual outputs. Usually, the verifier makes a comparison between actual and expected outputs. The process of finding correct and reliable expected outputs is called oracle problem [4].

Whittaker [2] explained software testing is divided into four phases: modeling the software's environment, selecting test scenario, running and evaluation test scenario, and measuring testing process. Test oracle is what testers usually use in the third phase when they want to evaluate the scenarios. In addition, according to [5], oracle information and oracle procedure are building blocks of each test oracle. The former is a source of expected results, and the last is the comparator. Thus, we concluded that the activities to provide an oracle and verify test cases could be as follows [6]:

* Corresponding author. Address: E1B-18-8, The Tamarind Condo, Jalan Sentul Indah, Sentul East, 51100 Kuala Lumpur, Malaysia. Tel.: +60 177136662 (H/P), +60 340411490 (home).

E-mail addresses: admin@rezanet.com (S.R. Shahamiri), wnasir@cs.utm.my (W.M.N. Wan-Kadir), suhaimiibrahim@utm.my (S. Ibrahim), sitizaiton@utm.my (S.Z. Mohd Hashim).

[1] Software Under Test.

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.02.006

1. Generate expected outputs.
2. Execute the test cases.
3. Map the input domain to the expected outputs and fetch the corresponding output for the executed test case.
4. Compare expected and actual outputs.
5. Decide whether there is a fault or not.

Among these activities, an oracle may deal with all of them instead of the test case execution activity. In particular, oracles are not responsible to execute the test cases, but they are employed to verify the execution results.

Automated oracle frameworks try to automate the related activities as much as possible. Nonetheless, we identified some challenges regarding the oracle related activities automation [6–8]. The first challenge deals with the first activity, which is how to provide the output domain automatically. It could be difficult and expensive to provide the expected outputs manually. In general, expected outputs are manually generated based on software specifications, domain specialist information and programmers knowledge of how the software should operate[2] [4]. An automated oracle needs automatic output-domain generation. The next challenge is to map the input domain to the output domain automatically, i.e. the third activity. A test oracle must provide expected results for any inputs mentioned by the software specifications. Moreover, it is impossible to provide an automated oracle without automated mapping. The final challenge is the automated comparator that deals with the two last activities. Sometimes it is not sufficient to perform a direct point-to-point comparison and the comparator needs to consider some tolerance when comparing the actual and the expected outputs. As an illustration, it is possible that the actual and expected outputs are different a little, but they can still be considered the same.

We applied *I/O Relationship Analysis* to address the first challenge, the output-domain generation. Then, a *Multi-Networks Oracle* based on several ANNs[3] was applied to handle the second challenge targeting complex software applications.[4] A set of thresholds to define the comparison tolerance and adjust the precision of the proposed oracle were defined in order to address the last challenge and make the automated comparator. Consequently, the proposed approach provides an automated oracle framework using I/O Relationship Analysis, Multiple-Networks Oracles based on ANNs and an automated comparator to address all the challenges mentioned before. We evaluated the proposed approach by *mutation testing* [9] a web-based car insurance application. First, a *Mutated Version* of the application was provided and injected with some faults. Then a fault-free version of the application was developed as a *Golden Version*[5] to evaluate the capability of the proposed oracle finding the injected faults.

The quality of the proposed approach was assessed by measuring the following parameters:

1. *Accuracy*: How accurate the oracle results are. It conveys what percent of expected outputs generated by the proposed oracle is accurate. It was measured by comparing the correct expected results, which were derived from the Golden Version, with the results produced by the proposed oracle.
2. *Precision*: How precise the oracle is. In particular, it means what the precision of the comparator is in order to compare the expected and actual results. It can be adjusted using the thresholds to adjust the comparison tolerance.
3. *Misclassification Error*: It is the amount of false reports produced by the comparator.
4. *Practicality*: It is the amount of the injected faults, which was identified by the proposed oracle. Note that it is different from accuracy because accuracy considers both successful and unsuccessful test cases but practicality considers only unsuccessful test cases (i.e. the test cases that reveal a fault).

The process of measuring above parameters is explained in Sections 5 and 6.

In addition, a comparative study between the existing automated oracles and the proposed one is provided based on which oracle automation challenges are handled, and what tool(s) they used to automate the oracle.

2. Related work and the comparative study

This section provides a survey on the prominent oracles. First, previous attempts to automate the output-domain generation are explained. Then, oracles that address the other challenges are reviewed. Finally, a comparative study on the popular oracles among software testers and the state-of-the-art automated oracles is proposed.

2.1. Automated output-domain generation

A few studies that could provide partly automated[6] expected-outputs-generation models were performed previously. *Combinatorial testing* is one of the test case generation methods. It verifies all possible input combinations; therefore, complete expected results sets are required. Since the number of these combinations may be very large in complex software, effective test data reduction is necessary; note that it is important to maintain the test quality. Although some combinatorial test reduction methods such as *Orthogonal Arrays* [10], *Experiment Design* [11] and *Random Sampling* [12] decreased the combination number, their impact on testing quality is unknown. Korel and Schroeder [13,14] proposed an approach to reduce the combinatorial test data by evaluating relationships between inputs and outputs. They claimed that the application of I/O analysis in test case reduction does not reduce the testing quality. Manual I/O analysis methods are program documentation analysis and interviewing with the stakeholders, while automated I/O analysis methods are *structural analysis* and *execution-oriented analysis*.

Structural analysis is either static or dynamic, and can be applied if testers have access to the source code. Static analysis examines the source code, but dynamic analysis examines the run-time information gathered from code execution. However, static analysis may overestimate program dependencies, and dynamic analysis is unable to guarantee full detection of I/O relationships [14], which it may result in imperfect test oracles.

Execution-oriented analysis is based on program execution. It finds I/O relationships by changing the input values, executing the program and observing the output changes. To put it differently, it finds the relationships between I/O by observing which outputs are being affected by the changed inputs. However, similar to dynamic analysis, it cannot guarantee to find all the I/O relationships. Note that dynamic analysis is a structural analysis method, i.e. the source code is required. On the other hand, execution-oriented analysis does not need the source code to identify I/O relationships.

[2] Direct Verification.
[3] Artificial Neural Networks.
[4] Complexity of the case study is discussed in Section 4.
[5] A Golden Version of a software application is its fault-free implementation that generates correct expected results. It was produced to evaluate the proposed oracle in this study.
[6] "Partly automated" means that only some of the required activities are automated.

Schroeder and Korel [15] used I/O Relationship Analysis to generate a reduced set of expected outputs. This study shows how a complete set of test cases can be generated based on some small sets of them. An I/O analysis was performed to discover associated inputs and outputs; then, a reduced set of test cases were created manually. Finally, the remaining test cases were created automatically by expanding the provided reduced sets of test cases.

2.2. Prominent oracles

Automated oracles have been considered lately finding an effective solution to address the oracle challenges. In the following, some state-of-the-art models to address the oracle automation challenges are explained after some popular test oracles are described.

Human oracles are the most popular type of un-automated oracles. They are people that understand the software domain such as programmers, customers, and domain experts. Human oracles address none of the automation challenges, but they can handle them manually. Nevertheless, they cannot be completely reliable when we have thousands of test cases. Furthermore, human oracles could be extremely expensive.

Random testing is a black-box test-data-generation technique that randomly selects test cases to match an operational profile [16]. It uses statistical models in order to predict the SUT reliability. It can scarcely be considered as a test oracle because it cannot provide expected outputs for the entire I/O domain that violates the oracle definition. Similarly, it automates the test case selection but the expected outputs must still be provided by any other approaches. Moreover, although the cost of random testing is low, its effectiveness is not adequate [17].

Cause-effect graphs and *decision tables* [18] are two popular test oracles that can be applied to address the mapping challenge. Although they are some tools to create the required structures using formal software specifications and written programs in order to create the oracles automatically, they still need some human observations and improvements to achieve the best oracle. They are usually considered as a black-box testing method but they may be seen as gray-box in case the improvements are necessary to examine the source code. In addition, they are limited to verify the logical relationships between causes and their effects, which are being implemented by the SUT. Similarly, they can grow very quickly and lose readability in complex applications that have large conditions and related actions [16]. Therefore, these oracles may not be effective in case the SUT applies heavy calculations to create the results.

Formal oracles were considered before. They can be generated from formal models such as formal specification [19,20] and relational program documentation [21,22]. They may address all the oracle automation challenges and provide a reliable oracle in case an accurate and complete formal model of the SUT, which is used to generate the oracle, is existed. However, especially for large-scale software applications, formal models can be extremely expensive and difficult being provided. Moreover, it seems to be compelling evidence of their effectiveness for large-scale software applications is not existed [23].

N-Version diverse system is a testing method based on various implementations of the SUT. It conveys a different implementation of *Redundant computation* as the Gold Version. To put it differently, it uses various versions of the SUT that they all implement the same functionalities, but they are implemented independently using several development team and methods. The independent versions may be seen as a test oracle. Nonetheless, the idea could result in an extremely expensive process because different versions of the SUT must be implemented by different programming teams. In addition, it is unable to guarantee the efficiency of the testing process since each of the implementations can be faulty itself. Manolache and Kourie [24] suggested an approach based on a similar approach to decrease the cost of N-Version Diverse Systems. In particular, the authors explained another solution based on N-Version Testing called *M-Model Programs testing* (M-mp). The new approach considers reducing the cost of the former approach and increasing the reliability of the testing process by providing more precise oracle. M-mp testing implements only different versions of the functions to be tested but N-Version Diverse implements several versions of the whole software. Although M-mp testing reduces the cost of the previous method, it can still be very expensive and unreliable.

There have been several attempts to apply *Artificial Intelligence* (AI) methods in order to make test oracles automatically. These methods are varied according to the applied AI method. As an illustration, Last and his colleges [25,26] introduced a fully automated black-box regression tester using *Info Fuzzy Network* (IFN). IFN is an approach developed for knowledge discovery and data mining. The interactions between the input and the target attributes (discrete and/or continuous) are represented by an information theoretic connectionist network. An IFN represents the functional requirement by an *oblivious* tree-like structure, where each input attribute is associated with a single layer, and the leaf nodes corresponds to the combinations of input values [25]. The authors developed an automated oracle that could generate test cases, execute, and evaluate them automatically based on previous versions of the SUT to *regression test* [27] the upcoming versions. This oracle may be applied to verify the old functionalities that remained unchanged in the new version of the SUT. Particularly, it is inapplicable of a fresh testing and for verifying the newly inserted functionalities.

Genetic algorithms (GA) were employed to provide oracles for *condition-coverage testing*, which concerns conditional statements [28]. Manual searching to find test cases that increase condition coverage is difficult when the SUT has many nested decision-making structures. Therefore, using an automated approach to generate effective test cases that cover more conditional statements is helpful. Michael et al. [29,30] introduced an automated *Dynamic Test Generator* using GA in order to identify effective test cases. Dynamic test generators are one of the white-box techniques that examine the source code of the SUT in order to collect information about it; then, this information can be used to optimize the testing process. The authors applied GA to perform condition coverage for C/C++ programs identifying test cases that may increase the source code coverage-criteria. The drawback of this approach is it may not be reliable for testing complex programs consisted of many nested conditional statements that each of them has several logical operations in their statements. Furthermore, this model is not platform-independent.

Taking advantages of the above approach, Sofokleous and Andreou [31] employed GA with a *Program Analyzer* to propose an automated framework for generating optimized test cases, targeting condition-coverage testing. The program analyzer examines the source code and provides its associated control flow graph, which the graph can be used to measure the coverage achieved by the test cases. Then, GA applies the graph to create optimized test cases that may reach higher condition-coverage criteria. The authors claimed that this framework provides better coverage than the previous one in order to testing complex decision-making structures. However, it is still platform dependent. Such GA based test frameworks may be considered as an automated test oracle that can address the mapping challenge only for condition-coverage testing, although they do not guarantee that full coverage can be achieved. Therefore, they may provide an imperfect oracle. Other methods such as *Equivalence Partitioning* and *boundary value analysis* [4,16] are more test case generation approaches than test oracles, hence they are not discussed here as test oracles.

Memon et al. [32–34] applied *AI planning* as an automated *GUI*[7] *test oracle*. Representing the GUI elements and actions, the internal behavior of the GUI was modeled in order to extract the GUIs expected state automatically during execution of each test case. A formal model of the GUI, which it was composed of its objects and their specifications, was designed based on GUI attributes and used as an oracle. GUI actions were defined by their preconditions and effects, and expected states were automatically generated using both the model and the actions from test cases. Similarly, the actual states were described by a set of objects and their properties that obtained by the oracle from an execution monitor. In addition, the oracle applied a verifier to compare the two states automatically and find faults in the GUI. The problem is a formal model of the GUI is required; moreover, it can only be used to verify the GUI states.

We conducted a study representing prominent methods that may be considered in order to automate software testing activities and showed how ANNs can be used to automate the software testing activities mentioned before [6,35]. As an illustration, Su and Huang proposed an approach to estimate and model software reliability using ANNs [36]. An effective test case selection approach using ANNs is introduced in [37], which it studied the applications of I/O analysis to identify which input attributes have more influence over outputs. It concluded that I/O analysis could significantly reduce the number of test cases. The ANN was applied to automate I/O analysis to determine important I/O attributes and their ranks. Khoshgoftar et al. [38,39] proposed a method to employ ANNs predicting the number of faults in the SUT based on software metrics. In addition, testability of program modules was studied by the same authors in [40].

Vanmali and his colleges proposed a Single-Network Oracle based on ANNs [41]. A Single-Network Oracle uses only one neural network to map the input domain to the output domain. The authors modeled an ANN to simulate the software behavior using the previous version of the SUT, and applied this model to regression test unchanged software functionalities. Using the previous version of the SUT, expected outputs were generated and the training pairs were employed to train the ANN. Aggarwal et al. applied the same approach to solve the *triangle classification problem* [42]. Particularly, an application implemented the triangle classification was tested using a trained ANN. Their work was followed by [43].

We applied Single-Network Oracles to test decision-making structures [7] and verify complex logical modules [8]. Particularly, a *Multilayered Perceptron ANN* that modeled a university subject-registration policy was trained using domain experts' knowledge of the system. Then, the trained ANN was employed to test an application that performed the subject registration and verified whether students followed the policies or not.

All the above ANN-based oracles were applied to test discrete functions. Mao et al. formulated ANNs as test oracles to test continuous functions [44]. Consider the continuous function $y = F(x)$ where x is the software input vector, y is the corresponding output vector, and F is the software behavior. The function F was modeled and expected outputs were generated using a trained ANN.

Despite Multilayered Perceptron neural networks that applied in all the above researches, Lu and Mao [45] used a different type of ANNs to provide automated oracles and test a small mathematic continues function. The Perceptron neural networks are explained later.

Although all of these studies on ANN-based oracles are shown the significance of Single-Network Oracles, none of them provides a complete automated framework because they did not consider the first challenge, the output-domain generation. They all assumed the expected results were pre-existed and did not discuss how they can be produced automatically in order to make the oracle. Hence, they only tried to address the second oracle challenge, i.e. the mapping. Moreover, Single-Network Oracles may not be reliable when the complexity of the SUT is increased because they require larger training samples that may make the ANN learning process complicated. A tiny ANN error could increase the oracle misclassification error significantly in large software applications. Previous ANN-based oracle studies were evaluated by small applications having small I/O domains. Therefore, one neural network was enough to perform the mapping in above studies. Finally, ANN-based oracles may not provide the very same output vector as the expected results. In particular, it is possible that a minuscule difference between the expected output generated by the ANN-based oracle and the correct one is existed. Any direct comparison may classify these results as faulty where they can be perceived as correct because the SUT may not require ultimate precision.

We used I/O Relationship Analysis to provide the expected results and the training data automatically in order to satisfy the first challenge that previous ANN-based oracles did not provide. Furthermore, we introduced Multi-Networks Oracles to address the mapping challenge in order to test more complicated software applications when Single-Network Oracles may fail to deliver a high quality oracle [46]. They are explained in detail in the next section. In addition, the proposed framework considers a comparator with some defined thresholds in order to adjust the oracle precision, and to prevent correct results being classified as incorrect where the distance between oracle results and the expected results are different a little. Consequently, the proposed framework may address the entire challenges automatically and provides an all-in-one black-box oracle.

Table 1 summarizes and compares the proposed approach with the above oracle models based on how they manage to automate the oracle challenges. It focuses on how different oracles handle the identified challenges automatically, and what their advantages and limitations are. For example, human oracles are completely un-automated; thus, none of the challenges can be automated if we consider humans as oracles. Although human oracles provide the expected outputs, map input domain to the output domain, and compare the actual and expected results manually, they are not able to propose an automated solution to address the challenges. On the other hand, as an illustration, IFN Regression Tester automates the entire oracle activities; therefore, all the challenges are marked as "handled automatically".

3. The proposed approach

The proposed approach focuses on the following three steps to provide the automated oracle framework, which is shown in Fig. 1.

3.1. Step 1: Applying I/O relationship analysis to provide the training data

The first step is to provide the expected result vectors and prepare the training samples. Large software applications may have thousands of I/O combinations; therefore, it could be very difficult and/or expensive to provide them manually. I/O Relationship Analysis may be applied to automate the output-domain generation [15].

Considering I/O relationships within an application, we only need to generate the expected outputs for the inputs that are related to them, which they are called *Reduced Sets*. Next, the remained outputs can be generated using the provided reduced

[7] Graphical User Interface.

Table 1
Automated test oracle models comparative study.

Oracle model	Automation tool	Test approach	Addressed challenges automatically[a]	Comments
Human oracles	None	None	× Output domain generation	Un-automated
			×Mapping	Very expensive
			×Comparison	A human oracle must completely understand the software domain
Random testing	None	Black-box	×Output domain generation	Un-automated
				Unreliable
			×Mapping	Low cost
			×Comparison	Imperfect oracle
Decision tables	Table	Gray-box	×Output domain generation	Automation requires specification or code analysis tool to fetch the logical relationships
			√Mapping	Needs human improvements
			×Comparison	High complexity when the number of conditions are increased
Cause-effect graphs	Graph	Black-box	×Output domain generation	Automation requires specification or code analysis tool to fetch the logical rules
			√Mapping	Expensive in case humans are considered to provide the logical relationships and required graphs
			×Comparison	Can lose readability in complex applications
Formal oracles	Formal model	Gray-box	√Output domain generation	Can be reliable in case an accurate and comprehensive formal model of the SUT is existed
			√Mapping	Can be extremely expensive for large-scale applications
			√Comparison	Its effectiveness for large-scale applications cannot be guaranteed
Genetic algorithm based test framework	Genetic algorithm	White-box	×Output domain generation	Only can be used for condition-coverage testing
			√Mapping	Cannot guarantee the oracle is perfect because they may not rich full coverage
			×Comparison	Platform dependent
AI planner test oracle	AI planning	Gray-box	√Output Domain	For GUI test only Based on GUI state comparison
			Generation	Requires formal model of the GUI
			√Mapping	Requires reliable documentations to provide a comprehensive formal model
			√Comparison	Expensive (needs formal model)
N-version diverse systems and m-model program testing	Redundant Computation	White-Box	√Output domain generation	Requires several implementations of system functionalities
			√Mapping	Extremely expensive
			√Comparison	Not reliable
IFN regression tester	Info Fuzzy Networks	Black-Box	√Output domain generation	Cannot be used for a fresh test
				Only applicable in regression testing
				Cannot test new functionalities
			√Mapping	Requires a reliable legacy system
			√Comparison	Requires additional knowledge for IFN modeling
Single-network ANN oracle	Single-network ANN	Black-box	×Output domain generation	Requires manual expected output generation
				It could not be reliable to test complex software and precise calculations
			√Mapping	Requires additional knowledge for ANN modeling
			√Comparison	Less expensive in case the output domain has already been provided
The proposed approach	I/O Relationship analysis and multi-networks Oracle	Black-box	√Output domain generation	Fully automated
				Can be used for a fresh testing
				Can evaluate both new and old functionalities
			√Mapping	More reliable than traditional Single-Network Oracles
			√Comparison	Less expensive

[a] The " × " sing means the oracle model does not provide an automated solution to the corresponding challenge, i.e. it must be addressed manually. On the other hand, the "√" means the model can handle the challenge automatically, considering the comments mentioned in the front.

sets. In the following, the process of providing the output domain is explained.

The *input vector* is defined as:

$$X = \{x_1, x_2, \ldots, x_n\}$$

Similarly, the *output vector* is:

$$Y = \{y_1, y_2, \ldots, y_n\}$$

Any possible x_n values are $D(x_n)$ and any possible y_m values are $D(y_n)$. Eq. (1) is a complete dataset consisted of all possible combinatorial input values.

$$T = D(x_1) \times D(x_2) \times \ldots \times D(x_n) \quad (1)$$

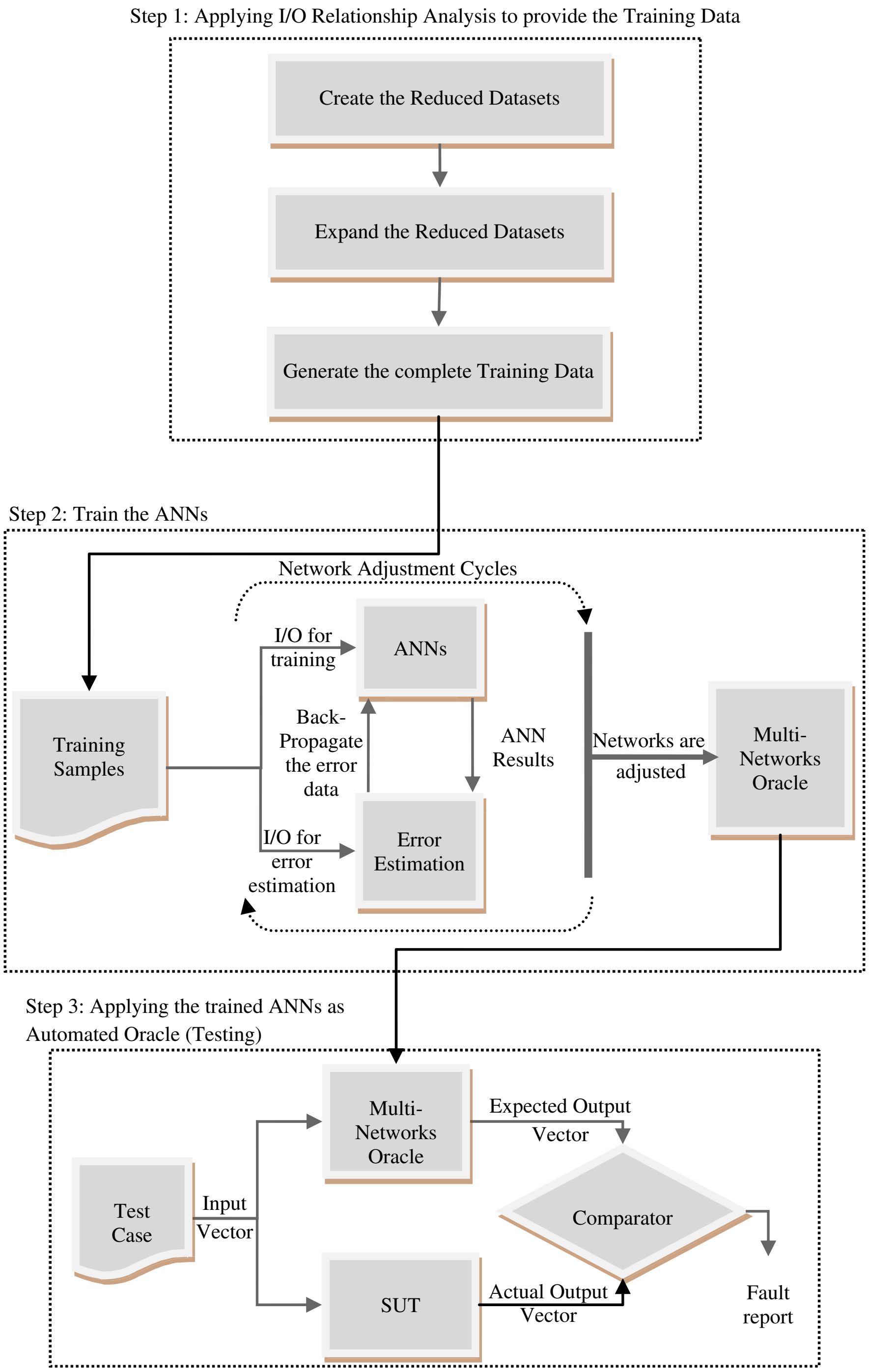

Step 1: Applying I/O Relationship Analysis to provide the Training Data
Create the Reduced Datasets
Expand the Reduced Datasets
Generate the complete Training Data
Step 2: Train the ANNs
Network Adjustment Cycles
I/O for training
ANNs
Training Samples
Back-Propagate the error data
ANN Results
Networks are adjusted
Multi-Networks Oracle
I/O for error estimation
Error Estimation
Step 3: Applying the trained ANNs as Automated Oracle (Testing)
Multi-Networks Oracle
Expected Output Vector
Test Case
Input Vector
Comparator
SUT
Actual Output Vector
Fault report

Fig. 1. The proposed approach framework.

Eq. (2) defines the size of the complete dataset T.

$$|T| = |D(x_1)| \times |D(x_2)| \times \ldots \times |D(x_n)| \tag{2}$$

Let $X(y_m)$ be all the input values of the vector X that are related to the output y_m. In particular, the value of y_m is changed if the value of any elements in $X(y_m)$ is changed; therefore, output y_m is related to the inputs in $X(y_m)$. Eq. (3) is the set of inputs that are associated with output y_m as *Reduced Set* (y_m) noted as $T_{red}\,(y_m)$.

$$T_{red}(y_m) = D'(x_1) \times D'(x_2) \times \ldots \times D'(x_n) \tag{3}$$

where

$$D'(x_i) = D(x_i), \textit{if } x_i \epsilon X(y_m)$$

$$D'(x_i) = \{\}, \textit{if } x_i \notin X(y_m)$$

To put it differently, $T_{red}\,(y_m)$ comprised of any input values that are related to the output y_m, and the rest input values are eliminated. $T_{red}\,(y_m)$ must be provided for each output element in the output vector Y. Finally, the complete output domain can be generated by merging the generated reduced sets automatically, i.e. by providing the following set:

$$D(Y) = \{T_{red}(y_1) \times T_{red}(y_2) \times \ldots \times T_{red}(y_m)\}$$

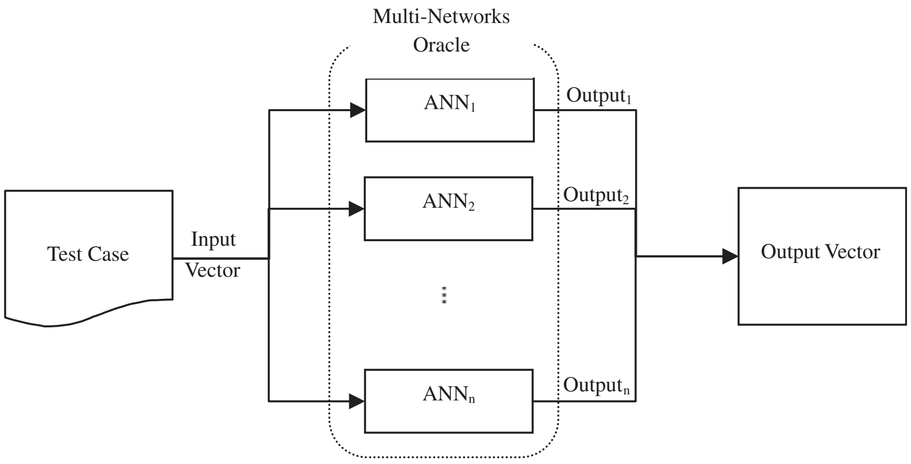


Fig. 2. Multi-networks Oracle.

3.2. Step 2: Train the ANNs

In recent years, we have seen a convincing move by the research community from theoretical research into practical one in order to find solutions for hardly solvable problems. Similarly, researchers are interested in model-free intelligent dynamic systems based on experimental data. ANNs are one of these systems to discover the hidden knowledge of experimental data and learn it while processing them. They are called intelligence systems because they can learn from calculations on numerical data and examples.

ANNs are mathematical models of human-neural-system that are restrictively able to simulate the information processing capability of the natural–neural-system [47]. They are network structures comprised of some correlated elements called *neurons*, each one has input(s) and output(s), and they perform a simple local *add* operation. Each neuron input has its corresponding *weight*, which it acts as the neuron memory. Furthermore, neurons have their *Bias Weights* and *Activation Functions* to squash the add operation results into specific values.

Adjusting these weights causes the network to learn the hidden knowledge being modeled through a process called the *Training Process*. In particular, *Training Samples*, which they are consisted of inputs and corresponding outputs, are given to the network by the process while the ANN tries to discover the relationships between inputs–outputs and learn them. After training, whenever an input combination is provided to the trained network, it should be able to generate its corresponding result(s). In other words, the ANN can recognize how to response to each input pattern. Note that the accuracy of the ANN depends on how well the network structure is defined and the training process is done.

Learning Rate is one of the training parameters that show how fast the ANN learns and how effective the training is. It can be in range (0...1); nevertheless, choosing a value very close to zero requires a large number of training cycles and makes the training process extremely slow. On the contrary, large values may diverge the weights and make the objective error function fluctuate heavily; thus, the resulted network reaches a state where the training procedure cannot be effective any more.

Networks with only one neuron have limitations due to inabilities to implement non-linear relationships; hence, Multilayer Perceptron networks are one of the most popular types of ANNs to solve the non-linearity problems [48]. They are multi-layered networks with no limitation to choose the number of neurons, and they must have an *input layer*, one or more *hidden layers* (or middle layers) and one *output layer*. The general model of Perceptron networks is *Feed-Forward* with *Back-Propagation* training procedure, which feed-forward are networks with inputs of the first layer connected and propagated to middle layers, the middle layers to the final layer, and the final layer to the output layer. In back-propagation procedure, after results of the network are generated, the parameters of the last layer to the first layer will be corrected in order to decrease the network misclassification error.

The misclassification error can be presented by *MSE*,[8] which it is the squared difference between the training sample outputs and the results generated by the network. According to [49], MSE can be considered to show how well the ANN outputs fit the expected outputs. Lower MSE represents better training quality and more accuracy to produce correct results.

The proposed approach uses the data generated in the previous step (step 1) as training samples to train the ANNs. Actual data is shown in section 5. Once the training is done, the trained networks can be employed as an automated oracle.

As mentioned in the previous sections, Single-Network Oracles that comprised of only one ANN may not be able to model the SUT if the software application is too complicated and generates several results. The main drawback of Single-Network Oracles is they learn the entire functionalities using only one ANN. Thus, if the number of the functionalities or the complexity of them is increased, the single ANN may fail to learn them with enough accuracy. This is why we introduced Multi-Networks Oracle, which uses several standalone ANNs in parallel instead of one, in order to distribute the complexity of the SUT among several ANNs. The advantages of Multi-Networks Oracles over Single-Network oracles are discussed in [46].

Suppose the output domain of the SUT consisted of outputs O_1 *to* O_n:Output Domain = $\{O_1, O_2, \ldots, O_n\}$.

In order to test the software using Single-Network oracles, the entire I/O domain is modeled using only one ANN. In particular, the ANN must learn the required functionalities to generate all outputs. On the other hand, using Multi-Networks Oracle, the functionalities associated to each output are modeled by a standalone ANN; thus, it is easier for the ANNs to converge because there is less to learn. Fig. 2 shows the structure of the Multi-Networks Oracle.

3.3. Step 3: Applying the ANNs as automated oracle (testing)

Once the networks are trained, they can be applied to the testing process. The test cases are executed on the trained networks (Multi-Networks Oracle) and the SUT simultaneously. An

[8] Mean Squared Error.

automated comparator is defined to decide whether the actual outputs produced by the SUT are faulty or not. The algorithm of the automated comparator is provided by section 6.

4. The case study

For the evaluation purpose, we applied the proposed approach to an industry-size case study. The case study is a web-based car insurance application to manage and maintain insurance records, determine the payment amount claimed by the customers, handle their requests for renewal, and other related insurance operations. Moreover, the application is responsible to apply the insurance policies on the customers' data and produce four outputs that are used throughout the application. The customers' data are fed to the application by eight inputs; thus, the input vector has eight inputs and the output vector has four outputs. Half of the outputs are continuous and the other half is binary.

Equivalence Partitioning was considered to provide the input domain (i.e. the inputs of the training samples), which it is a test case reduction technique that classifies a group of data values into an equivalence class where the SUT processes them the same way. The test of one representative of the equivalence class is sufficient because for any other input value of the same class the SUT will not behave differently [16,18,50–52]. Table 2 shows the input vector and its equivalence partitions, and Table 3 depicts the output vector.

The case study has an I/O domain comprised of 13824 equivalence partitions as explained is section 5. The business layer of the application was implemented by 1384 lines, which user-interface layer implementation is not included. *Cyclomatic Complexity* implies the structural complexity of program code. Particularly, it measures how many different paths must be traversed in order to fully coverage each line of the code. Bigger numbers mean more testing is required and the SUT is more complicated. The Cyclomatic Complexity of the insurance policies implemented by the case study was 38, which was measured using *Software Measurements* package included in *Microsoft Visual Studio.Net 2010*. The case study was developed using *C#* and *ASP.Net*.

Table 2
the input vector X and $D\ (x_n)$.

	Inputs	Equivalence partitions ($D\ (x_n)$)	$\|D\|$
1	The driver's experience	Less than 5 years Between 5 and 10 years More than 10 years	3
2	Type of the driver license	A–D	4
3	Type of the car	Sedan SUV MPV Hatchback Sport Pickup	6
4	The credit remains in the insurance account	Less than 100$ More than 100$	2
5	Cost of this accident	Less that 25% of the initial credit Between 25% and 50% of the initial credit More than 50% of the initial credit	3
6	Type of the insurance	Full Third party	2
7	The car age	Less than 10 years Between 10 to 15 years Between 15 to 20 years More than 20 years	4
8	Number of registered accidents since last year	0 Between 0 and 5 Between 5 and 8 more than 8	4

Table 3
The output vector Y.

	Outputs	Description
1	Insurance extension allowance	A Boolean data item that is true if the insurance is allowed for extension and false otherwise
2	Insurance elimination	A Boolean data that is true if the insurance account is terminated and false otherwise
3	The payment amount	The amount to be paid for accidents according to the claimed amount and the insurance rules. This output is continuous
4	Credit	This amount is assigned to each insurance account as its available credit. This output is continuous

We applied the proposed approach to verify how well the insurance policies were implemented by the application. In order to provide a complete test oracle, we must provide at least one training sample for each of the insurance policies and train the ANNs completely. Each insurance policy represented by one of the identified equivalence partitions. Therefore, as mentioned in the next section, we need 13824 equivalence partitions as the training samples to cover all the insurance policies implemented by the case study. To put it differently, the insurance policies were reflected by 13 824 different training (I/O) pairs, which they were chosen by Equivalence Partitioning. I/O Relationship Analysis was employed to generate the SUT responses to the identified partitions (i.e. the output domain) and provide the training samples.

5. The Experiment

As mentioned before, the training samples must be provided before the training begins. We employed I/O Relationship Analysis to provide the training sample outputs automatically.

Fig. 3 shows the I/O relationships for the case study. Since we developed the case study in-house, the relationships between inputs and outputs were known to us. However, the testers can use the techniques mentioned on Section 2.1 in case they do not have the I/O relationships. According to Section 3.1, I/O Relationship Analysis parameters can be defined as follows:

$$X = \{inp_1, inp_2, \ldots, inp_8\} \text{ (as defined in Table 2)}$$
$$Y = \{out_1, out_2, out_3, out_4\} \text{ (as defined in Table 3)}$$
$$D(x_n) : \text{as defined in Table 2}$$
$$|T| = |D(inp_1)| \times |D(inp_2)| \times \cdots \times |D(inp_8)|$$
$$= 3 * 4 * 6 * 2 * 3 * 2 * 5 * 4 = 13824$$

According to the relationships in Fig. 3, the sized of the reduced datasets can be calculated as:

$$|T_{red}(out_1)| = |D(inp_3)| \times |D(inp_7)| \times |D(inp_8)| = 6 \times 4 \times 4 = 96$$

$$|T_{red}(out_2)| = |D(inp_4)| \times |D(inp_8)| = 2 \times 4 = 8$$

$$|T_{red}(out_3)| = |D(inp_1)| \times |D(inp_2)| \times |D(inp_5)| \times |D(inp_6)|$$
$$= 3 \times 4 \times 3 \times 2 = 72$$

$$|T_{red}(out_4)| = |D(inp_1)| \times |D(inp_2)| \times |D(inp_7)| \times |D(inp_0)|$$
$$= 3 \times 4 \times 4 \times 4 = 192$$

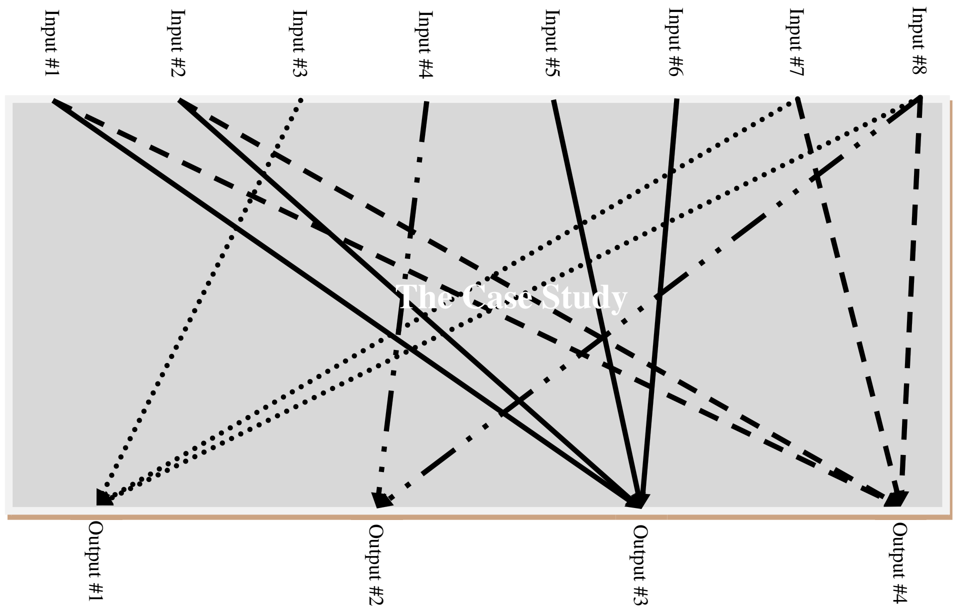


Fig. 3. I/O relationships for the case study (X (y_m)).

Table 4
The Multi-Networks Oracle Training parameters and MSEs.

	Corresponding output	Type of the output	Input neuron #	Hidden neuron #	Output neuron #	Learning rate	Training cycles	*MSE*
Network 1	Output #1 (insurance extension allowance)	Binary	8	13	1	0.01	1400	0.00025
Network 2	Output #2 (insurance elimination)	Binary	8	3	1	0.1	150	0.00002
Network 3	Output #3 (the payment amount)	Continuous	8	30	1	0.25	2000	0.0023
Network 4	Output #4 (credit)	Continuous	8	30	1	0.25	2000	0.00048
						Total MSE		0.00076

Therefore, the size of the final reduced set is:

$$\begin{aligned}|T_{red}(Y)| &= |T_{red}(out_1)| + |T_{red}(out_2)| + |T_{red}(out_3)| + |T_{red}(out_4)| \\ &= 96 + 8 + 72 + 192 = 368\end{aligned}$$

Finally, the complete output domain was created by making a union of the generated reduced sets:

$$D(Y) = \bigcup_{i=1}^{4} T_{red}(out_i)$$

Manual generation must provide all the samples manually. Nonetheless, considering I/O Relationship Analysis, we only produced 368 samples manually by asking the customer to provide the expected results after we created the reduced sets. The other 13 456[9] samples were generated automatically using I/O Relationship Analysis explained above. Thus, 97% of the required training samples were provided automatically.

After the training samples were ready, we proceeded with the training process. A small tool was created to train the required ANNs using *NeuronDotNet*[10] package, which it provides the required libraries and tools to create and train ANNs using *Microsoft Visual Studio.Net* easily.

Since the insurance policies were complicated, a Single-Network Oracle seemed to fail learning them with enough accuracy; the minimum MSE we saw through several experiments was as big as 0.0046 over 10 000 training cycles on a $8 \times 30 \times 4$ Multilayer Perceptron network with learning rate 0.01 (eight input, 30 hidden and four output neurons). On the other hand, the proposed Multi-Networks Oracle could reach a lower MSE 0.00 076 as mentioned in Table 4. Note that all of the networks were Multilayer Perceptron networks with *Sigmoid* activation function and one hidden layer. They were trained using back-propagation algorithm with the same training samples. Table 4 highlights the Multi-Networks Oracle structure and the MSEs, and Fig. 4 depicts the corresponding training error graphs. Each of the networks in Fig. 4 was associated to its output. For example, Network 1 is the first ANN that generates out_1; Network 2 is the second ANN that generates out_2 and so on.

Since the output vector has four outputs, we needed four ANNs to make the Multi-Networks Oracle. Note that each of the ANNs in the Multi-Networks Oracle only was given by the corresponding output; for example, the training samples for the second ANN that modeled the second output contained all of the inputs but only out_2 since the ANN must generate the second output only. Thus, out_1, out_3 and out_4 are not required while training the second ANN.

All of the inputs fed and outputs generated by the ANNs are normalized. In particular, out_3 and out_4 are continuous data items that were scaled to range [0.1]. For example, The Payment Amount (i.e.

[9] 13 824-358 = 13 456.
[10] http://neurondotnet.freehostia.com/.

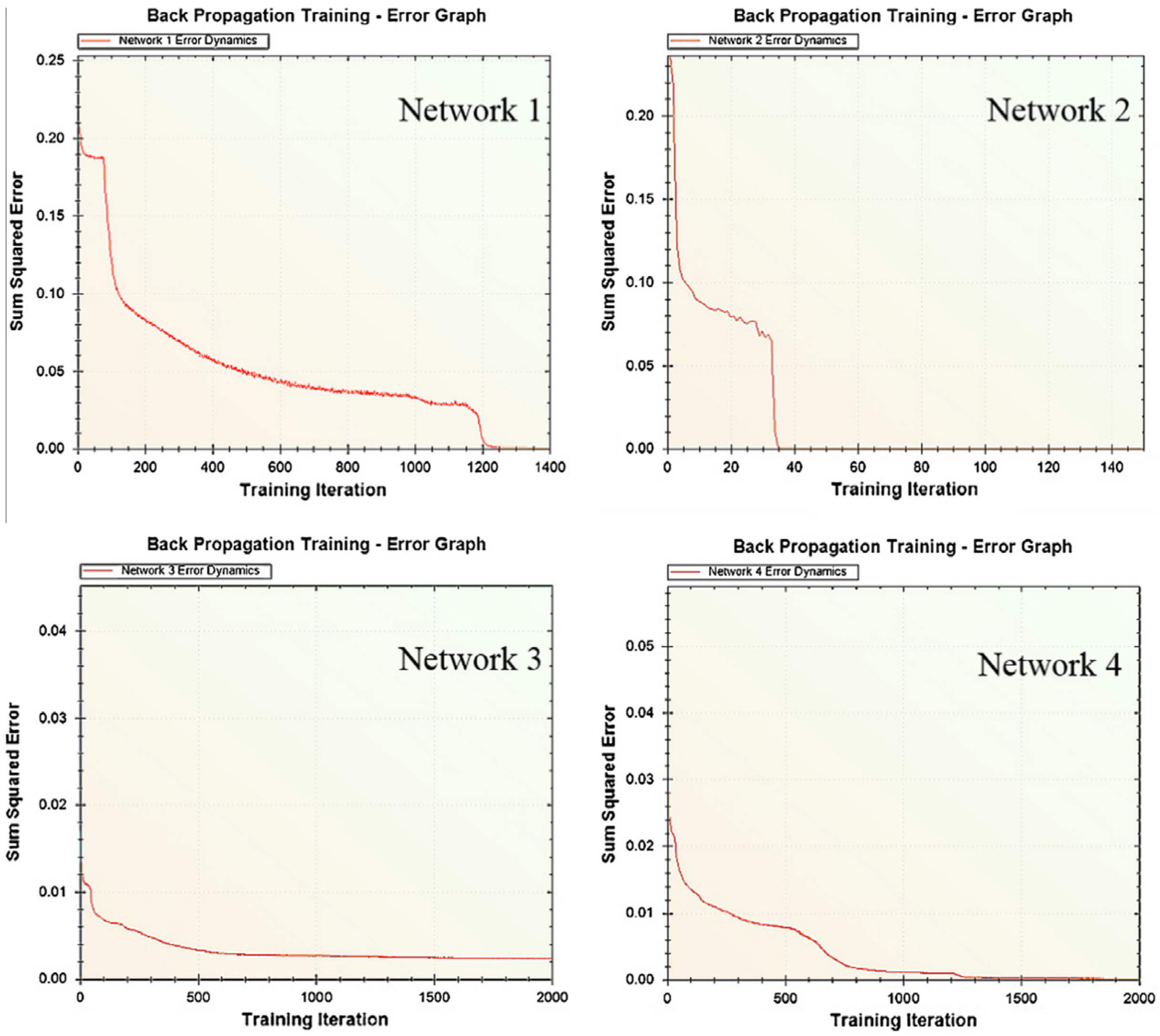


Fig. 4. The training error graphs.

out_3) is the percentage of the claimed amount according to the insurance rules. The percentage was divided to 100 and scaled to the range. The out_4 is the same. Binary inputs were scaled to zero percent (i.e. "False") and one hundred percent (i.e. "True"). The out_1 and out_2 were binary so they were treated the same. Regarding the discrete inputs, they were mapped to some integer values. As an illustration, the "A" value of inp_2 was regarded as one, "B" as two and so on.

Thus, using a scale dependent parameter dose not raise any issue because all of the outputs were scaled to the same unit. Scale independent metrics are used when comparisons across series of different units are required. MSE is neither a random variable nor it depends on any random quantities. Among scalar estimators, MSE is the most commonly used cost that has been used for back-propagation algorithm. Similarly, most of the previous studies on ANN-based oracles have cited MSE as the performance function [7,8,42–44].

Providing better confidentiality of the networks, we also measured the actual *Absolute Error Rate* of the trained networks after applying the test cases. The absolute error rate shows squared difference between the proposed oracle results and the correct expected results at the testing time.

Once the neural networks were trained, it was possible to use them as an automated oracle. Nevertheless, the quality of the resulted oracle needed to be verified against accuracy, precision, misclassification error and practicality to evaluate the proposed approach as mentioned in the first section.

6. Evaluation

We have evaluated the proposed approach using mutation testing by developing two versions of the case study. The first version was a Golden Version, which it was a complete fault-free implementation of the case study to generate correct expected results. The other version was a Mutated Version that was injected with common programming mistakes.

In order to verify the oracles, the ability of the networks to find the injected faults was evaluated using the following steps:

1. The test cases were executed on both the Golden Version and the Mutated Version.
2. Similarly, the test cases were given to the proposed oracle.
3. The Golden Version results were completely fault free and considered as expected results.

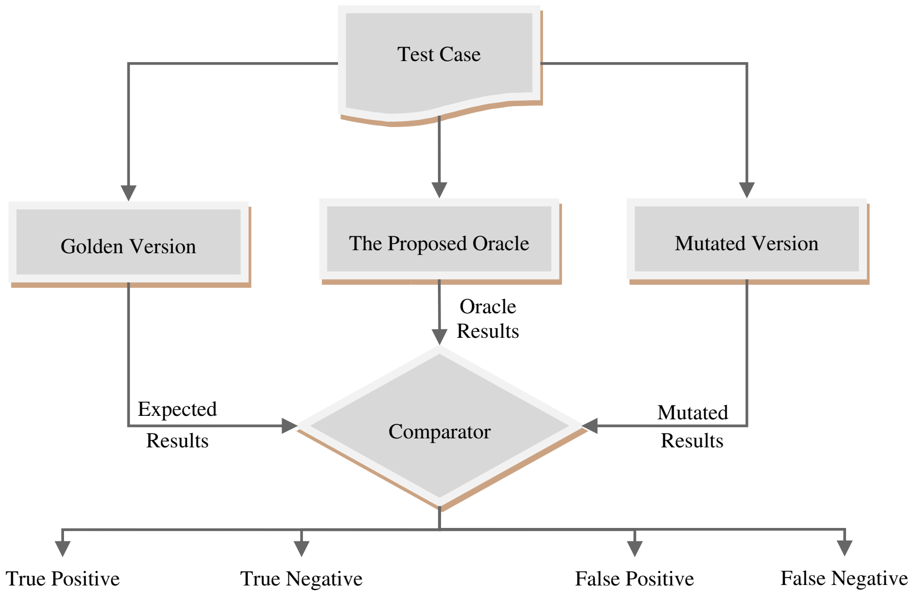


Fig. 5. The evaluation process.

4. All the results (the results of the oracles, the mutated results and expected results) were compared with each other and any distance between them more than a defined tolerance was reported as a possible fault as explained later.
5. To make sure the oracle results were correct, the outputs of the oracles were compared with the expected results (the Golden Version results) and their distances were considered as oracle absolute error.

Fig. 5 shows the verification process. The results of the comparator can be divided into four categories:

1. True Positive: All the results are the same. Therefore, the comparator reports "No fault". This means there is actually neither any fault in the mutated version nor the oracle. True positive represents the successful test cases.
2. True Negative: Although the expected and the oracle results are the same, they are different from the mutated results. In this case, the oracle results are correct. Therefore, the oracle correctly finds a fault in the mutated version.
3. False Positive: Both the oracle and mutated version produced the same incorrect results. Therefore, they are different from the expected results and faults in the oracle and the mutated version are reported. To put it differently, the oracle missed a fault.
4. False Negative: The mutated and the expected results are the same, but they are different from the oracle results. Thus, the comparator reports a faulty oracle.

Since the third and fourth categories represent the oracle misclassification error, we mainly considered these types of faults to measure the accuracy of the proposed oracle.

Moreover, as illustrated in Table 4, the MSEs were not exactly zero. Hence, we defined four thresholds presenting how much distance between the expected and the oracle results could be ignored[11]. If the distance was less than the thresholds, both results were considered the same. Otherwise, the results of the oracle were not accurate and false positive or false negative faults were reported. The thresholds can be used to define the precision of the proposed oracle, and it can be increased or decreased as necessary, as discussed later. Fig. 6 shows the comparison algorithm.

7. Results

Using the proposed approach, we could provide a framework to automatically execute 3000 test cases and verify them. An automated test driver was developed to create the test cases, execute them on the case study, apply Equivalence Partitioning and ask the proposed oracle to generate the expected outputs for the identified partition, verify the mutated and oracle results against the expected results (Fig. 5), and create a comprehensive report, all automatically. The tester only needs to set the comparator precision (i.e. the thresholds), choose how many test cases must be created, and leave the tool to do the rest automatically. In addition, it is possible to create a test case manually using any test case generation method, such as Boundary Value Analysis, and run it automatically. Fig. 7 portrays the test driver user-interface.

More than 9000 faults were injected to the mutated class, which the proposed approach was used to find them. Table 5 shows some of the mutants that explain the type of the faults injected to the Mutated Version. As can be seen, the types of the faults are ordinary programming mistakes such as wrong usage of logical operators, incorrect arguments, wrong assignments and typecasting.

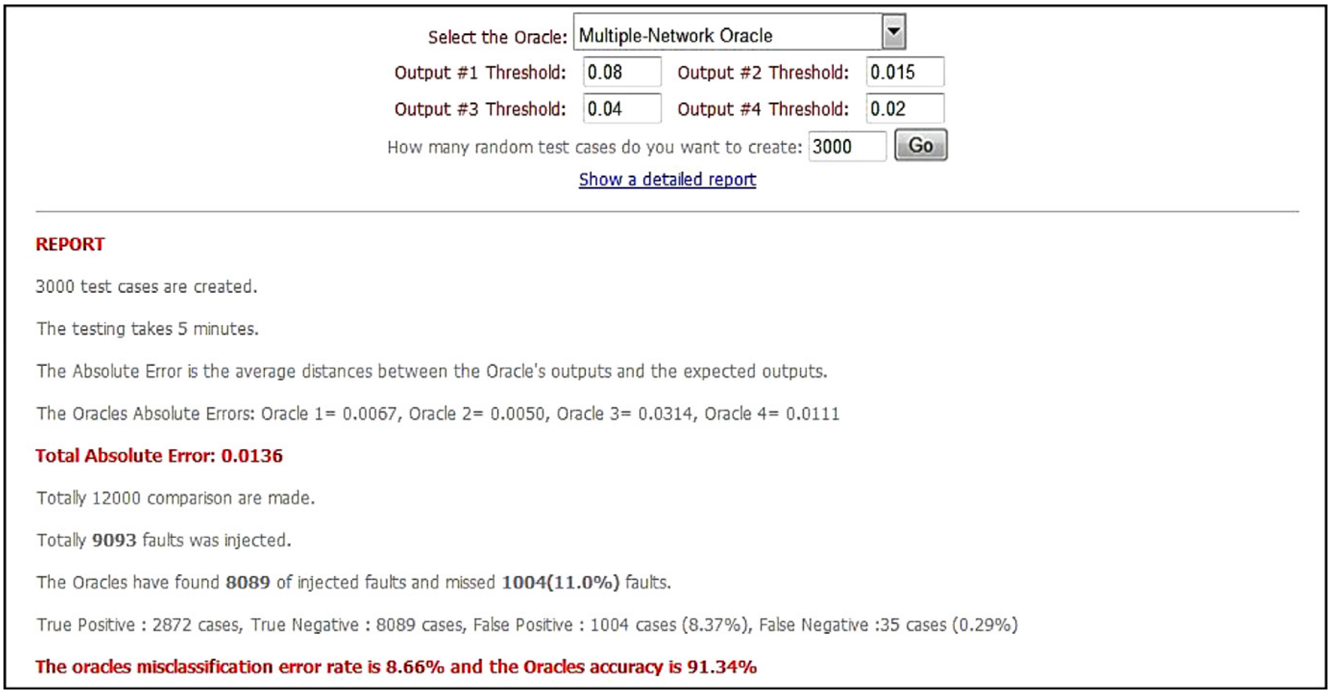
Furthermore, we selected three different thresholds combinations to measure the quality of the proposed approach with highest, mid and lower precision, and repeated the experiment in order to show the behavior of the proposed oracle under different precision.

Table 6 highlights the results. The thresholds represent the comparison precision. Lower thresholds make the proposed oracle more precise. True negative (the injected faults detected by the proposed approach) and false positive (faults that were missed and remained in the application) are illustrated as well. The misclassification error rate is the sum of false negative (the percentage

[11] The comparison tolerance.

```
1. Start
2. Calculate the distance between the expected result and the oracle result as absolouteDistance
3. Calculate the distance between the mutated result and the oracle result as mutatedDistacne
4. If the absoluteDistance ≤ threshold then
        a.  The oracle is correct
        b. If the mutatedDistance ≤ thresdhold then True Positive is reported (no fault)
        c. Else True Negative is reported (a fault is found)
5. Else
        a. The oracle is faulty
        b. If the mutated result and the expected result are not  the same
            i. False Positive is reported (missed fault)
        c. Else False Negative is reported (the oracle failed to verify the mutated result)
6. End
```

Fig. 6. The comparison algorithm.


Select the Oracle: Multiple-Network Oracle

Output #1 Threshold: 0.08 Output #2 Threshold: 0.015

Output #3 Threshold: 0.04 Output #4 Threshold: 0.02

How many random test cases do you want to create: 3000 Go

Show a detailed report

REPORT

3000 test cases are created.

The testing takes 5 minutes.

The Absolute Error is the average distances between the Oracle's outputs and the expected outputs.

The Oracles Absolute Errors: Oracle 1= 0.0067, Oracle 2= 0.0050, Oracle 3= 0.0314, Oracle 4= 0.0111

Total Absolute Error: 0.0136

Totally 12000 comparison are made.

Totally **9093** faults was injected.

The Oracles have found **8089** of injected faults and missed **1004(11.0%)** faults.

True Positive : 2872 cases, True Negative : 8089 cases, False Positive : 1004 cases (8.37%), False Negative :35 cases (0.29%)

The oracles misclassification error rate is 8.66% and the Oracles accuracy is 91.34%

Fig. 7. The automated test driver.

Table 5
Samples of the mutants.

Original code	Mutated code	Error type	Affected output
If (Credit ⩽ 100)	If (Credit < 100)	Operator change	3
If ((DriverLicence = = "C") || (DriverLicence = = "D"))	If ((DriverLicence = = "A") && (DriverLicence = = "D"))	Argument change + operator change	3
Initialcredit = 0.6 * RequestedCredit	Initialcredit = 0.3 * RequestedCredit	Value change	4
Double inp5 = (double) ClaimedAmount/Credit	Double inp5 = (float) ClaimedAmount/Credit	Typecasting change	3
Return false	Return true	Value change	2
If (((CarAge > 15) && (CarAge ⩽ 20)) && (CarType = = "Sport"))	If (((CarAge > 10) && (CarAge ⩽ 20))||(CarType == "Sport"))	Argument change + operator change	1

of correct mutated outputs classified as being incorrect) and false positive (the percentage of incorrect mutated outputs classified as being correct) that was measured based on the total comparisons were made. In particular, misclassification error rate is the inability of the oracle to produce correct results. On the other hand, the accuracy is the percentage of the oracle correct generated results. It represents how many percent of the oracles results are equal to the expected results and classified as true positive or true negative during the comparison, considering the thresholds.

The lowest thresholds combination provides the highest precision (the precision and accuracy are discussed on the next section in detail). It is equal to the minimum distance between

Table 6
The proposed approach evaluation results.

	Lower threshold (highest precision)	Mid threshold	Higher threshold
Thresholds	Output 1: 0.08 Output 2: 0.015 Output 3: 0.04 Output 4: 0.02	Output 1: 0.08 Output 2: 0.015 Output 3: 0.08 Output 4: 0.04	Output 1: 0.08 Output 2: 0.015 Output 3: 0.08 Output 4: 0.08
Total comparisons	12000		
Average threshold (Precision)	0.155	0.215	0.255
ANN 1 absolute error		0.006	
ANN 2 absolute error		0.005	
ANN 3 absolute error		0.03	
ANN 4 absolute error		0.01	
Total absolute error		0.014	
Number of injected faults	9123	9139	9046
True positive	2841	2851	2950
True negative (detected faults) (practicality)	8094 (88.7% of the injected faults)	8874 (97.1% of the injected faults)	8841 (97.7% of the injected faults)
False positive (missed faults)	1029 (11.3% of the injected faults)	265 (2.9% of the injected faults)	205 (2.3% of the injected faults)
False negative	36 (0.3% of the total comparisons)	10 (0.08% of the total comparisons)	4 (0.03% of the total comparisons)
Misclassification error rate	8.88%	2.29%	1.74%
Accuracy	91.13%	97.71%	98.26%

each possible expected result values. We did not need to adjust the thresholds for the two binary outputs very carefully because their MSEs and absolute errors are tiny (almost zero). In addition, the distance between the binary output values (true or one, and false or zero) is large enough. Nonetheless, we selected thresholds 0.08 for the first output from ANN_1, and 0.015 for the second output from ANN_2 to increase the precision of the oracle. For example, any ANN_1 results between 0.92 and 1.00 considered as true (or 1), and any output in the range [0,0.08] considered as false (or zero). Any other ANN_1 output values are considered as the oracle misclassification error.

Since the minimum distances between the third and fourth expected results values are tiny (0.04 for the third output and 0.02 for the fourth output, which they were measured by observing their values), the thresholds for the continuous outputs must be selected carefully. In case of the highest precision, those minimum distances need to be selected as the corresponding thresholds.

8. Discussion

As mentioned before, Equivalence Partitioning was applied to select the training samples. It ensured that each possible input–output pattern was considered. Since all of the 13824 equivalence partitions were applied as the training samples, each possible test case fell into one of the partitions that were already given to the ANNs. To put it differently, the training samples covered the entire I/O domain so all the possible test cases were given to the ANNs as the training samples. Then, at the testing time, the test driver applied the same Equivalence Partitioning and asked the ANNs to provide outputs for the identified partition. Consequently, network generalizability validation is not necessary where there is nothing to be generalized. In particular, there are no new or un-foreseen test cases that Equivalence Partitioning has already not provided as the training samples.

However, if the SUT or its I/O domain is changed so a new equivalence partition is appeared, some new test cases should be inserted into the training samples in order to verify the changes. In this situation, the ANNs should be re-trained with the new training samples that reflect the changes.

The proposed approach applies ANNs in order to map the input domain to the output domain. In our experiment, no network prediction was required because of Equivalence Partitioning. Nonetheless, if the testers are unable to cover the entire I/O domain by providing adequate training samples, it is necessary to evaluate the generalizability because ANN prediction is required to perform the mapping.

Thresholds define the precision and directly influence the accuracy. Higher thresholds may consider oracles faulty outputs as correct and lower thresholds may make the oracles correct results faulty. Moreover, lower thresholds make the oracle more precise because fewer distances may be ignored. On the other hand, higher thresholds provide higher accuracy owing to more distance is being ignored, and more oracle outputs are being considered as correct. However, it increases the chance that faulty oracle outputs are classified as expected outputs. Therefore, the precision and accuracy have an inverse relationship: more precision and less accuracy, and vice versa.

Selecting the threshold depends on the required SUT reliability. Since critical software applications and high-risk software modules require extremely high reliability, we suggest using lower thresholds to increase their precision because they must be very reliable so any missed faults may cause them to fail. For non-critical software applications that lower reliability is accepted, higher thresholds may provide enough accuracy and automation.

The aim of the present paper is to provide an automated oracle. In order to automate an oracle, we must address the challenges mentioned in section 1 and automate their related oracle activities. The proposed framework addresses the issues as follows: the first challenge, i.e. output-domain generation, is handled using I/O Relationship Analysis. According to section 5, 97% of this challenge was automated in our experiment. The second challenge is the mapping challenge, which is addressed using Multi-Networks Oracles based on ANNs. The only manual activity is to define and train the ANN that is a simple task considering the existing related tools. Note that training samples are already provided automatically using I/O Relationship Analysis. The last challenge is the comparator. It is completely automated using the algorithm provided by Fig. 6. Thus, we can say that the entire oracle activities are automated with a little human effort to prepare the required data and the environment. Consequently, the proposed model may handle all the oracle challenges to provide an automated oracle.

The results show automations offered by the proposed approach. It was possible to automate the testing process and test the case study in a few minutes applying the proposed approach; particularly, it took about five minutes to execute and verify 3000 test cases on our lab PC.

The accuracy, practicality and misclassification error of the proposed approach were measured as shown in Table 6. The precision of the proposed oracle can be set by adjusting the thresholds as necessary. Although increasing the precision causes the accuracy to decrease, the minimum accuracy 91.13%, which was achieved by adjusting the precision being highest, seems reasonable considering the automation offered by the proposed approach. Furthermore, if the application to be tested does not require extremely high reliability, lower precision can be considered as well. Our experiment shows the proposed oracle was accurate 97.71%, selecting a mid-thresholds set.

9. Conclusion and future work

This research offers an automated framework to produce test oracles. The challenges to provide an automated oracle are explained in detail. Then, I/O Relationship Analysis, Multi-Networks Oracle based on ANNs, and an automated comparator were applied to address the challenges and test the case study. A comparative study between the prominent oracles and the proposed one was also provided.

I/O Relationship Analysis was used to provide the output domain and generate the ANNs training samples automatically. Then, the generated samples were applied to train a Multi-Networks Oracle, which it considered several ANNs to learn the SUT instead of one, in order to address the mapping challenge. Finally, an automated comparator that applied some defined thresholds was employed to complete the framework. The thresholds determine how much distance between the expected results and the oracle results may be ignored.

The proposed approach was evaluated using mutation testing. The case study was mutated and some faults were injected to it; then, the proposed oracle was asked to find the injected faults. A fault-free version of the case study was developed as a Golden Version to evaluate the oracle results and generate the expected outputs. All of the testing activities were performed automatically.

Three experiments were conducted to measure the quality of the proposed oracle with different precision levels. All the experiments were able to find the injected faults with minimum accuracy 91.17% to maximum accuracy 98.26%.

Almost all of the previous ANN-based oracle studies considered supervised learning paradigm to model the software application as test oracles. It can be helpful if unsupervised learning and reinforced learning paradigms will be investigated as well.

This paper uses the output domain as the criteria to separate the ANNs and distribute the complexity in order to make the Multi-Networks Oracle. We suggest that other distribution criteria, such as software modules, being considered to study whether the Multi-Networks Oracle produces better results or not.

Moreover, among the prominent automated oracles mentioned in Table 1, IFN-based oracles seem to be adequate, despite the fact that they can only be applied for regression testing. However, it is suggested that other studies being conducted in order to use our proposed model but replace the ANNs with IFNs and see whether the resulted model can be used for a fresh testing or not. IFNs may replace ANNs to address the mapping challenge. A comparison between our model and the resulted model that demonstrates how both models handle the challenges in terms of quality will be beneficial.

References

[1] A.M.J. Hass, Guide to Advanced Software Testing, Artech House, Boston, 2008.
[2] J.A. Whittaker, What is software testing? And why is it so hard?, IEEE Software 17 (2000) 70–79
[3] L. Ran, C. Dyreson, A. Andrews, R. Bryce, C. Mallery, Building test cases and oracles to automate the testing of web database applications, Information and Software Technology 51 (2009) 460–477.
[4] P. Ammann, J. Offutt, Introduction to Software Testing, first ed., Cambridge University Press, New York, 2008.
[5] Q. Xie, A.M. Memon, Designing and comparing automated test oracles for GUI-based software applications, ACM Transactions on Software Engineering and Methodology 16 (2007) 4.
[6] S.R. Shahamiri, W.M.N.W. Kadir, S.Z. Mohd-Hashim, A comparative study on automated software test oracle methods, in: Proceedings of the 2009 Fourth International Conference on Software Engineering Advances, IEEE Computer Society, Porto, Portugal, 2009, pp. 140–145.
[7] S.R. Shahamiri, W.M.N.W. Kadir, S. Ibrahim, An automated oracle approach to test decision-making structures, in: Proceedings of the 2010 3rd IEEE International Conference on Computer Science and Information Technology, IEEE Publication, Chengdu, China, 2010, pp. 30–34.
[8] S.R. Shahamiri, W.M.N.W. Kadir, S. Ibrahim, A single-network ANN-based oracle to verify logical software modules, in: Proceedings of the 2010 2nd International Conference on Software Technology and Engineering, IEEE Publication, Puerto Rico, USA, 2010, pp. 272–276.
[9] M.R. Woodward, Mutation testing: its origin and evolution, Information and Software Technology 35 (1993) 163–169.
[10] M.S. Phadke, Planning efficient software tests, CrossTalk 10 (10) (1997) 11–15.
[11] S.R.J. Dalal, A. Karunanithi, N. Leaton, J.M. Lott, C.M., Model-based testing of a highly programmable system, in: The Ninth International Symposium on Software Reliability Engineering, IEEE, Paderborn, 1998, pp. 174–178.
[12] T.Y. Chen, On the expected number of failures detected by subdomain testing and random testing, IEEE Transactions on Software Engineering 22 (2) (1996) 109–119.
[13] B. Korel, P.J. Schroeder, Maintaining the quality of black-box testing, The Journal of Defense Software Engineering 14 (5) (2001) 24–28.
[14] P.J. Schroeder, B. Korel, Black-box test reduction using input–output analysis, SIGSOFT Software Engineering Notes 25 (2000) 173–177.
[15] P.J. Schroeder, P. Faherty, B. Korel, Generating expected results for automated black-box testing, in: Proceedings 17th IEEE International Conference on Automated Software Engineering ASE, Edinburgh, UK, 2002, pp. 139–148.
[16] A. Spillner, T. Linz, H. Schaefer, Software Testing Foundations, 2nd ed., Rocky Nook Inc., Santa Barbara, 2007.
[17] S.C. Ntafos, On comparisons of random, partition, and proportional partition testing, IEEE Transactions on Software Engineering 27 (2001) 949–960.
[18] P.C. Jorgensen, Software Testing: a Craftsman's Approach, second ed., CRC Press, LLC, 2002.
[19] P. Stocks, D. Carrington, A framework for specification-based testing, IEEE Transactions on Software Engineering 22 (1996) 777–793.
[20] D.J. Richardson, S.L. Aha, T.O. O'Malley, Specification-based test oracles for reactive systems, in: Proceedings of the 14th International Conference on Software Engineering, ACM, Melbourne, Australia, 1992, pp. 105–118.
[21] D. Peters, D.L. Parnas, Generating a test oracle from program documentation, in: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, Seattle, WA, USA, 1994, pp. 58.
[22] D.K. Peters, D.L. Parnas, Using test oracles generated from program documentation, IEEE Transactions on Software Engineering 24 (1998) 161–173.
[23] S.L. Pfleeger, L. Hatton, Investigating the influence of formal methods, Computer 30 (1997) 33–43.
[24] L.I. Manolache, D.G. Kourie, Software testing using model programs, Software, Practice and Experience 31 (2001) 1211–1236.
[25] M. Last, M. Freidman, Black-box testing with info-fuzzy networks, in: M. Last, A. Kandel, H. Bunke (Eds.), Artificial Intelligence Methods in Software Testing, World Scientific, 2004, pp. 21–50.
[26] M. Last, M. Friendman, A. Kandel, Using data mining for automated software testing, International Journal of Software Engineering and Knowledge Engineering 14 (2004) 369–393.
[27] L.C. Briand, Y. Labiche, S. He, Automating regression test selection based on UML designs, Information and Software Technology 51 (2009) 16–30.
[28] M.R. Woodward, M.A. Hennell, On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC, Information and Software Technology 48 (2006) 433–440.
[29] C. Michael, G. McGraw, Automated software test data generation for complex programs, in: Proceedings of the 13th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, Honolulu, Hawaii, 1998, pp. 136–146.
[30] C.C. Michael, G. McGraw, M.A. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering 27 (2001) 1085–1110.
[31] A.A. Sofokleous, A.S. Andreou, Automatic, evolutionary test data generation for dynamic software testing, Journal of Systems and Software 81 (2008) 1883–1898.
[32] A. Memon, A. Nagarajan, Q. Xie, Automating regression testing for evolving GUI software, Journal of Software Maintenance and Evolution 17 (2005) 27–64.
[33] A.M. Memon, Automated GUI regression testing using AI planning, in: M. Last, A. Kandel, H. Bunke (Eds.), Artificial Intelligence Methods in Software Testing, World Scientific, 2004, pp. 51–99.
[34] A.M. Memon, M.E. Pollack, M.L. Soffa, Automated test oracles for GUIs, SIGSOFT Software Engineering Notes 25 (2000) 30–39.

[35] S.R. Shahamiri, M.N.W.K. Wan, Intelligent and automated software testing methods classification, in: 4th Postgraduate Annual Research Seminar (PARS'08), UTM Skudai, Malaysia, 2008.

[36] Y.-S. Su, C.-Y. Huang, Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models, Journal of Systems and Software 80 (2007) 606–615.

[37] P. Saraph, M. Last, A. Kandell, Test case generation and reduction by automated input–output analysis, in: IEEE International Conference on Systems, Man and Cybernetics, IEEE, Washington DC, United States, 2003, pp. 768–773.

[38] T.M. Khoshgoftaar, A.S. Pandya, H.B. More, A neural network approach for predicting software development faults, in: A.S. Pandya (Ed.), Third International Symposium on Software Reliability Engineering, Research Triangle Park, NC 1992, pp. 83–89.

[39] T.M. Khoshgoftaar, R.M. Szabo, P.J. Guasti, Exploring the behavior of neural network software quality models, Software Engineering Journal 10 (1995) 89–96.

[40] T.M. Khoshgoftaar, E.B. Allen, Z. Xu, Predicting testability of program modules using a neural network, in: E.B. Allen (Ed.) 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology, Richardson, TX, USA, 2000, pp. 57–62.

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

[42] K.K. Aggarwal, Y. Singh, A. Kaur, O.P. Sangwan, A neural net based approach to test oracle, ACM SIGSOFT Software Engineering Notes 29 (2004) 1–6.

[43] J. Hu, W. Yi, C. Nian-Wei, G. Zhi-Jian, W. Shuo, Artificial Neural Network for Automatic Test Oracles Generation, in: Proceedings of the 2008 International Conference on Computer Science and Software Engineering, IEEE Computer Society, Wuhan, Hubei, 2008.

[44] Y. Mao, F. Boqin, Z. Li, L. Yao, Neural networks based automated test oracle for software testing, in: Proceedings of the 13th International Conference Neural Information Processing, Springer Verlag, Heidelberg, D-69121, Germany, Hong Kong, China, 2006, pp. 498–507.

[45] Y. Lu, M. Ye, Oracle model based on RBF neural networks for automated software testing, Information Technology Journal 6 (2007) 469–474.

[46] S.R. Shahamiri, W.M.N.W. Kadir, S. Ibrahim, S.Z. Mohd-Hashim, Artificial neural networks as multi-networks automated test oracle, Automated Software Engineering, in preparation.

[47] R.J. Schalkoff, Artificial Neural Networks, McGraw-Hill, 1997.

[48] M.B. Menhaj, Basics of Neural Networks, first ed., Amirkabir Technology University, Tehran, 2001.

[49] A. Heiat, Comparison of artificial neural network and regression models for estimating software development effort, Information and Software Technology 44 (2002) 911–922.

[50] J.D. McCaffrey, Software Testing: Fundamental Principles and Essential Knowledge, BookSurge Publishing, 2009.

[51] G.J. Myers, The Art of Software Testing, second ed., Wiley, 2004.

[52] R. Patton, Software Testing, second ed., Sams Publishing, 2005.