The Integration of Machine Learning into Automated Test Generation: A Systematic Literature Review

Afonso Fontes, Gregory Gay*

Department of Computer Science and Engineering, Chalmers | University of Gothenburg, Sweden

Abstract

Background: Machine learning (ML) may enable effective automated test generation.

Aims: We characterize emerging research, examining testing practices, researcher goals, ML techniques applied, evaluation, and challenges.

Method: We perform a systematic literature review on a sample of 97 publications.

Results: ML generates input for system, GUI, unit, performance, and combinatorial testing or improves the performance of existing generation methods. ML is also used to generate test verdicts, property-based, and expected output oracles. Supervised learning—often based on neural networks—and reinforcement learning—often based on Q-learning—are common, and some publications also employ unsupervised or semi-supervised learning. (Semi-/Un-)Supervised approaches are evaluated using both traditional testing metrics and ML-related metrics (e.g., accuracy), while reinforcement learning is often evaluated using testing metrics tied to the reward function.

Conclusions: Work-to-date shows great promise, but there are open challenges regarding training data, retraining, scalability, evaluation complexity, ML algorithms employed—and how they are applied—benchmarks, and replicability. Our findings can serve as a roadmap and inspiration for researchers in this field.

Keywords: Automated Test Generation, Test Case Generation, Test Input Generation, Test Oracle Generation, Machine Learning

*Corresponding author

Email addresses: afonso.fontes@chalmers.se (Afonso Fontes), greg@greggay.com (Gregory Gay)

1. Introduction

Software testing is invaluable in ensuring the reliability of the software that powers our society [1, 2]. It is also notoriously difficult and expensive [3], with severe consequences for productivity, the environment, and human life if not conducted properly [4]. New tools and methodologies are needed to control that cost without reducing the quality of the testing process.

Automation has a critical role in controlling costs and focusing developer attention [5, 6]. Consider test generation an effort-intensive task where sequences of program *input* and *oracles* that judge the correctness of the resulting execution are crafted for a system-under-test (SUT) [7]. Effective automated test generation could lead to immense effort and cost savings.

Automated test generation is a popular research topic [8], and outstanding achievements have been made in recent years [6]. Still, there are critical limitations to current approaches. Major among these is that generation frameworks are applied in a *general* manner—techniques target simple universal heuristics, and those heuristics are applied in a static manner to all systems equally. Parameters of test generation can be tuned by a developer, but this requires advanced knowledge and is still based on the same universal heuristics. Current generation frameworks are largely unable to adapt their approach to a particular SUT, even though such projects offer rich information content in their documentation, metadata, source code, or execution logs [9, 10, 11]. Such static application limits the potential effectiveness of automated test generation.

Advances in the field of machine learning (ML) have shown that automation can match or surpass human performance across many problem domains [12]. ML has advanced the state-of-the-art in virtually every field. Automated test generation is no exception [2]. Recently, researchers have begun to use ML either to *directly* generate input or oracles [13] or to *enhance* the effectiveness or efficiency of existing test generation frameworks [9]. ML offers the potential means to adapt test generation to a SUT, and to enable automation to optimize its approach without human intervention.

We are interested in understanding and characterizing emerging research around

the integration of ML into automated test generation[1]. Specifically, we are interested in which testing practices have been addressed by integrating ML into test generation, the goals of the researchers using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how such techniques are trained and validated, and how the whole test generation process is evaluated. We are also interested in identifying the emerging field's limitations and open research challenges.

To that end, we have performed a systematic literature review. Following a search of relevant databases and a rigorous filtering process, we have gathered 97 relevant studies. We have examined each study, gathering the data needed to answer our research questions. The findings of this study include:

- Machine Learning supports the generation of both input and oracles, with a greater focus on input generation (67% of publications). During input generation, ML either directly generates input or is used to improve the efficiency or effectiveness of existing test generation methods.
  - Input generation practices include system testing, specialized types of system testing (GUI, performance, Combinatorial Interaction Testing), and unit testing. Most of these approaches are Black Box approaches, with White Box approaches primarily restricted to unit testing.
  - There has been an increase in publications on system and GUI input generation since 2017, potentially related to the emergence of web and mobile applications and autonomous driving, as well as to the availability of open-source ML frameworks.
  - ML supports generation of test verdict, metamorphic (and other property-based), and expected output oracles.
- The most common types of ML are supervised (59%) and reinforcement learning

[1]We focus specifically on the use of ML to enhance test generation, as part of the broader field of AI-for-Software Engineering (AI4SE). There has also been research in automated test generation for ML-based systems (SE4AI) [14]. These studies are out of the scope of our review.

(36%). A small number of publications also employ unsupervised (4%) or semi-supervised (2%) learning.

  - Supervised learning is the most common ML for system testing, Combinatorial Interaction Testing, and all forms of oracle generation. Produced models perform regression (often for input generation) or classification (often for oracle generation). Neural networks, especially Backpropagation NNs, are the most common supervised techniques.
  - RL is the most common ML for GUI, unit, and performance testing, and is generally based on Q-Learning. It is effective for practices with scoring functions and when testing requires a sequence of input steps. It is also effective at tuning generation tools.
  - Unsupervised learning, e.g., clustering, is effective for filtering tasks such as discarding similar test cases or identifying weakly-tested parts of a SUT.

- ML-enhanced test generation is still evaluated by traditional metrics (e.g., coverage, fault detection). (Semi-/Un-)Supervised approaches are also evaluated using ML-related metrics (accuracy, training data and model size, labeling and training effort, # of clusters). RL approaches are evaluated using testing metrics (often tied to the reward function).

- We have also observed multiple limitations and challenges:

  - Supervised learning is limited by the required quantity, quality, and contents of training data—especially when human effort is required. Oracles particularly suffer from these issues. RL and adversarial learning are viable alternatives when data collection and labeling can be automated.
  - Models should be retrained over time. How often retraining occurs depends, partially, on the cost to gather and label additional data or on the amount of human feedback required.
  - Scalability of many ML techniques to real-world systems is not clear. When modeling complex functions, varying degrees of abstraction could be ex-

plored if techniques are unable to scale. In all evaluations, a range of systems should be considered, and a detailed analysis of scalability (e.g., of accuracy, training, learning rate) should be performed.

- Researchers rarely justify the choice of ML technique or compare alternatives. The use of open-source ML frameworks can ease comparison. Deep learning and ensemble techniques, as well as hyperparameter tuning, should also be explored.
- Research is limited by the overuse of simplistic examples, the lack of common benchmarks, and the unavailability of code and data. Researchers should be encouraged to use available benchmarks, and provide replication packages and open code. New benchmarks could be created for ML challenges (e.g., oracle generation).

Our study is the first to thoroughly summarize and characterize this emerging research field[2] We hope that our findings will serve as a roadmap for both researchers and practitioners interested in the use of ML as part of test generation and that it will inspire new advances in the field.

2. Background and Related Work

This section explains background concepts on software testing (Section 2.1) and ML (Section 2.2). We also explore related work (Section 2.3).

2.1. Software Testing

Before software can be trusted, it is essential to verify that it functions as intended. The verification process usually involves *testing*—the application of *input* to the system, and analysis of the resulting *output*, to identify visible failures or other unexpected behaviors in the system-under-test (SUT) [1].

[2]This study extends an initial SLR on test oracle generation [15]. Our comprehensive study also includes test input generation, updates the sample of publications to include those from 2021, and features an extended analysis and discussion of the publications.

```
@Test
public void testPrintMessage() {
    String str = "Test Message";
    TransformCase tCase = new TransformCase(str);
    String upperCaseStr = str.toUpperCase();
    assertEquals(upperCaseStr, tCase.getText());
}
```

Figure 1: Example of a unit test case written using the JUnit notation for Java.

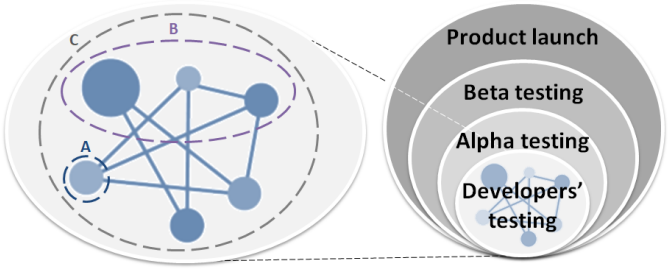


Figure 2: Example of granularity levels of the tests to the left and testing phases to the right. A: Unit Testing; B: Integration Testing; C: System Testing.

During testing, a *test suite* containing one or more *test cases* is applied to the SUT. A test case consists of a *test sequence (or procedure)*–a series of interactions with the SUT–with *test input* applied to some component of the SUT. Depending on the granularity of the testing effort [3], the input can range from method calls to API calls to actions within a graphical interface. Then, the test case will validate the output of the called components against a set of encoded expectations—the *test oracle*—to determine whether the test passes or fails [1]. An oracle can be a predefined specification (e.g., an assertion), output from a similar program or a past version of the SUT, a model, or even manual inspection by humans [7].

An example of a test case, written in the JUnit notation for Java, is shown in Figure 1. The test input consists of passing a string to the constructor of the `TransformCase` class, then calling its `getText()` method. This method should transform the string to upper-case. To ensure this is the case, an assertion checks whether the actual output matches the expected output—an upper-case version of the input.

Testing can be performed at different levels of granularity, using tests written in code or by humans through an external interface. This concept is illustrated in Figure 2. The lowest granularity level is unit testing (A), which focuses on testing of isolated code modules (generally classes). Module interactions are tested during integration testing (B). Then, during system testing (C), the SUT is tested through one of its defined interfaces—a programmable interface (e.g., a REST API), a command-line interface, a graphical user interface (GUI), or any other externally-accessible interface.

Our focus is on automated test generation, which is often focused on tests in an executable format. Human-driven testing activities, such as alpha/beta testing, are out of the scope of this study as such activities are often not amenable to automation.

2.2. Machine Learning

Machine Learning (ML) approaches construct models from observed data—and the structure of that data—to make predictions [16]. Instead of being explicitly programmed with a set of instructions like in traditional software, ML algorithms "learn" from observations using statistical analyses, facilitating the automation of decision-making processes. With the support and accessibility of powerful tools like Tensorflow [17], Keras [18], and OpenAI gym [19], ML has enabled and empowered many new applications in the past decade. As computational power and data availability increase, such approaches will increase in their capabilities and accuracy.

ML approaches largely fall into four categories—supervised, semi-supervised, unsupervised, and reinforcement learning [16]—as presented in Figure 3. In supervised learning, algorithms replicate patterns learned from previously labeled *training* data. That is, they infer a model from the training data that makes predictions about newly encountered data. Supervised learning algorithms are typically used for classification—prediction of which label from a finite set fits an input—or regression—predictions in an unrestricted format, e.g., a continuous value or production of text. If a sufficiently large training dataset with a low level of noise is used, an accurate model can often be trained quickly. However, a model is generally static once trained and cannot be improved without re-training.

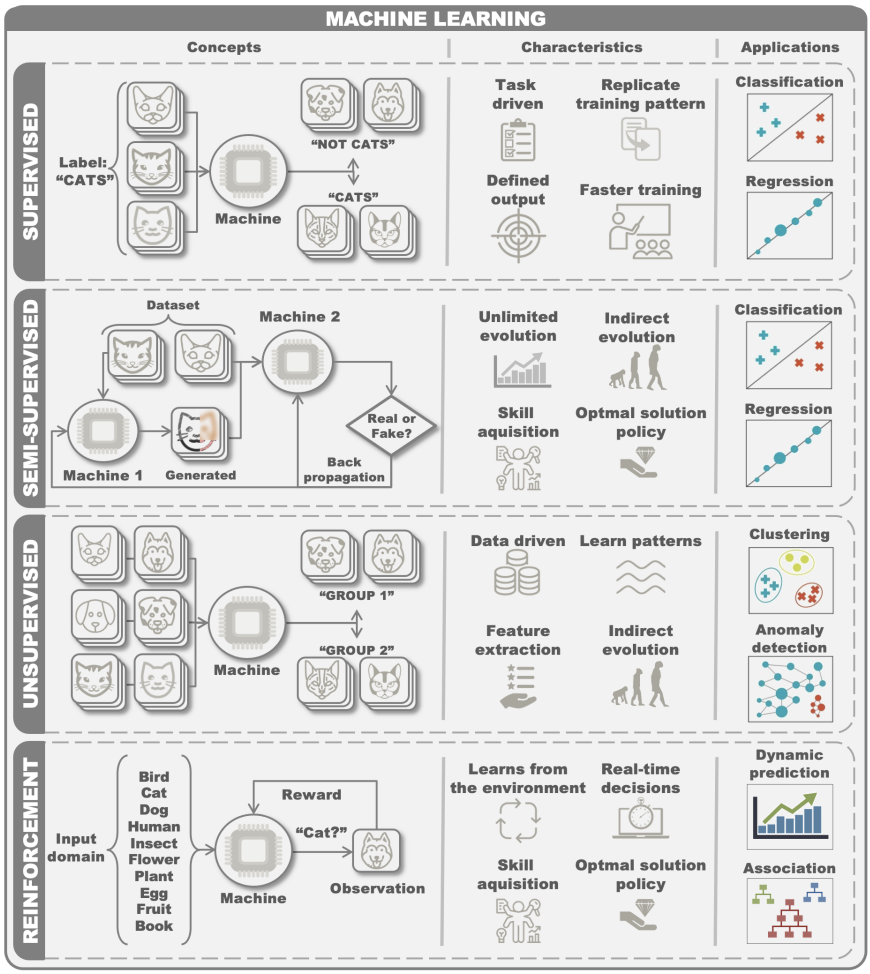


Figure 3: Types of Machine Learning and their concepts, characteristics, and applications.

In contrast to supervised methods, unsupervised algorithms do not use previously-labeled data. Instead, approaches identify patterns in unlabeled data based on the similarities and differences between data items. They model the data indirectly, with little-to-no human input. Rather than making predictions, as Figure 3 shows, unsupervised techniques aid in understanding data by, e.g., clustering data (group related items), extracting interesting features, or detecting anomalies.

Semi-supervised algorithms start with training data, then employ feedback mechanisms that evolve the model through automated retraining. For example, adversarial

networks, illustrated under the semi-supervised tab in Figure 3 refine accuracy by augmenting the training data with new input. Its strategy consists of putting two supervised learning algorithms in competition. One of the algorithms attempts to create new inputs that mimic the training data, while the second predicts whether these inputs are part of the training data or are impostors. The first refines its ability to create convincing fakes, while the second tries to separate the fakes from the originals. Semi-supervised approaches require a longer training time than traditional supervised approaches, but can achieve more optimal models, often with a smaller initial training set.

Reinforcement learning (RL) algorithms select actions based on an estimation of their effectiveness towards achieving a measurable goal [20]. RL often does not require training data, instead learning through sequences of interactions with its environment. Based on these interactions, the agent learns an optimal decision policy. RL "agents" use feedback on the effect of the actions taken to improve their estimation of the actions most likely to maximize achievement of this goal. Feedback is provided by a "reward function"—a scoring function that indicates the progress made when a specific action is taken. The agent can also adapt to a changing environment, as estimations are refined each time an action is taken. Such algorithms are often the basis of automated processes, such as AI chatbots or autonomous driving, and are very effective in situations where sequences of predictions are required in real-time.

Recent research often focuses on "deep learning" (DL). Figure 4 presents the conceptual difference between DL and traditional ML. DL approaches are able to make complex and highly accurate inferences from massive datasets. Many DL approaches are based on complex many-layered neural networks—networks that attempts to mimic how the human brain works [21]. Such neural networks employ a cascade of nonlinear processing layers where one layer's output serves as the successive layer's input. DL can abstract complex data and extract relevant features to learn not only from observation, but also from training data (labeled or not) [21]. Deep learning requires a computationally intense training process and larger quantities of data than traditional ML approaches, but can learn highly accurate models, can extract features and relationships from data automatically, and can potentially apply learned models across applications. "Deep" approaches exist for all four of the ML types discussed above.

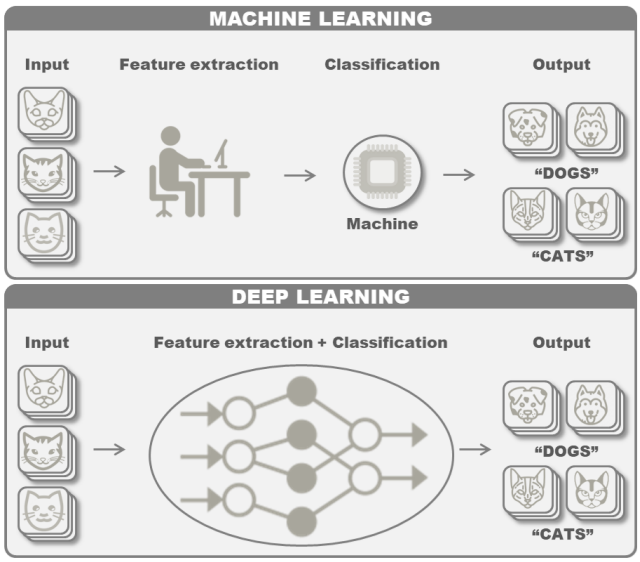


Figure 4: Illustrated example of Machine Learning and Deep Learning approaches.

Machine learning can benefit many aspects of test generation. For example, given a set of monitored test executions, a model can be inferred that associates test input with the likelihood of producing a crash, with a prediction of the amount of the code-base executed by the input, or with a particular class of program output. Such models can be used to identify new input that is likely to achieve the particular goal of the testing process. ML models or RL agents can be used to tune the parameters of existing test generation approaches, learning the parameters that lead to the best outcomes for new systems-under-test. Unsupervised approaches could intelligently group existing test cases into clusters, identifying functionality that has been not been sufficiently tested.

2.3. Related Work

Other secondary studies, including SLRs, traditional surveys, and mapping studies, overlap with ours in scope. We briefly mention some of these publications below. Our SLR is the first to date focused specifically on the application of ML to automated test generation, including both input and oracle generation, and no related study overlaps in full with our research questions. We have also examined a larger and more recent sample of publications than related secondary studies.

Durelli et al. performed a systematic mapping study on the application of ML to software testing [22]. Their scope is broad, examining how ML has been applied to any aspect of the testing process. They mapped 48 publications to categories of testing activities, study types, and ML algorithms employed. They observe that ML has been used to generate both input and test oracles. They note that supervised algorithms are used more often than other categories of ML algorithms, and that Artificial Neural Networks (ANN) are the most used algorithm. Jha and Popli also conducted a short review of literature applying ML to testing activities [23], and note that ML has been used for both input and oracle generation.

Ioannides and Eder conducted a survey on the use of AI techniques to generate test cases targeting code coverage—known as "white box" test generation approaches [24]. Much of their survey focuses on optimization techniques, such as genetic algorithms, but they do note that ML has been used to generate coverage-maximizing test input.

Barr et al. performed a detailed survey of research on test oracles up to 2014 [7]. They divide test oracles into four broad types, including those specified by human testers, those derived automatically from development artifacts, those that reflect implicit properties of all programs, and those that rely on a human-in-the-loop to judge test results. Approaches based on ML will primarily fall into the "derived" category, as they learn automatically from project artifacts in order to replace or augment human-written oracles. They discuss early approaches to using ML to derive oracles.

Balera et al. conducted a systematic mapping study on hyper-heuristics used in search-based test generation [25]. Search-based test generation applies optimization algorithms to generate test input. A hyper-heuristic is a secondary optimization performed to tune the search strategy employed by the primary optimization. In this case, a hyper-heuristic would adjust the parameters used to generate test cases to better adapt test generation to the current system-under-test. A hyper-heuristic can be performed using ML, especially reinforcement learning, but can also be guided by many other algorithms. In our study, we also observe the use of ML-based hyper-heuristic.

ID	**Research Question**	**Objective**
RQ1	Which testing practices have been supported by integrating ML into the generation process?	Highlights testing scenarios and systems types targeted for ML-enhanced test generation.
RQ2	What is the goal of using machine learning as part of automated test generation?	To understand the reasons for applying ML techniques to enable or enhance test generation.
RQ3	How was machine learning integrated into the process of automated test generation?	Identifies the category of ML applied, how it was integrated, and how it was trained and validated.
RQ4	Which machine learning techniques were used to perform or enhance automated test generation?	Identify specific ML techniques used in the process, including type, learning method, and selection mechanisms.
RQ5	How is the test generation process evaluated?	Describe the evaluation of the ML-enhanced test generation process, highlighting common metrics and artifacts (programs or datasets) used.
RQ6	What are the limitations and open challenges in integrating ML into test generation?	Highlights the limitations of enhancing test generation with ML and future research directions.

Table 1: List of research questions, along with motivation for answering the question.

3. Methodology

Our concern in this work is to understand how researchers have used ML to enhance or directly perform automated test generation, including both input and oracles. We have investigated contributions to the literature related to this topic and seek to understand their methodology, results, and insights. According to Petersen et al. [26], in order to achieve such goals, it is necessary to carry out a secondary study. To that end, we have performed a Systematic Literature Review (SLR) [27].

We are interested in assessing the *effect* of integrating ML into the test generation process, understanding the *adoption* of these techniques—how and why they are being integrated, and which specific techniques are being applied, and identifying the potential *impact* and *risks* of this integration. Table 1 lists the research questions we are interested in answering and clarifies the purpose of asking such questions.

The first three questions are high-level questions that clarify how ML has enabled or enhanced test generation, why ML was applied, and which specific testing scenarios were targeted by the enhanced generation techniques. **RQ1** enables us to categorize publications in terms of specific testing practices. **RQ2** is motivational, covering the authors' primary objectives. We are interesting in how the authors intended to use

ML—e.g., to directly generate input, to enhance an existing test generation technique, or to identify weaknesses in a testing strategy.

In contrast, **RQ3-5** are technical questions. **RQ3** examines the category of ML technique (i.e., supervised, unspervised, semi-supervised, or reinforcement learning), as well as its training and validation processes. **RQ4** examined which specific ML techniques were used to perform the generation task. **RQ5** focuses on how the test generation approach is evaluated, including metrics and types of systems tested. Finally, the last research question covers the limitations of the proposed approaches and open research challenges (**RQ6**).

To answer these questions, we have performed the following tasks:

1. Formed a list of publications by querying publication databases (Section 3.1).
2. Filtered this list for relevance (Section 3.2).
3. Extracted data from each study, guided by properties of interest (Section 3.3).
4. Identified trends in the extracted data in order to answer each research question (described along with results in Section 4).

3.1. Initial Study Selection

In order to locate publications for consideration in this review, a search for relevant literature was conducted using four databases: IEEE Xplore, ACM Digital Library, Science Direct, and Scopus. To narrow the database results, we created a search string by combining terms of interest regarding software test generation and machine learning.The search string used was:

(“test case generation” OR “test generation” OR “test oracle” OR “test input”) AND (“machine learning” OR “reinforcement learning” OR “deep learning” OR “neural network”)

These keywords are not guaranteed to capture all existing articles on the use of machine learning in test generation. However, they are intended to attain a relevant sample of the research literature. Specifically, we combine terms related to test

case generation—including specific test components—and terms related to machine learning—including common technologies.

Our focus is specifically on the use of ML in test generation, not on any particular form of test generation. To obtain a representative sample, we have selected ML terms that we expect will capture a wide range of publications. These terms may omit some techniques that could be in-scope, but allow us to obtain a representative sample while controlling the amount of manual inspection.

We applied an initial filter to the results using the advanced search option in each database. We limited our search to publications in conferences and journals (excluding pre-prints, technical reports, abstracts, editorials, and book chapters) in the English language. Our set of articles was gathered in December 2021, containing an initial total of 2,805 articles. This is shown as the first step in Figure 5.

To evaluate the search string's effectiveness, we conducted a three-step verification process. First, we randomly sampled ten entries from the final publications that remained following manual filtering. Then we looked in each article for ten citations that were in scope, resulting in a list of 100 citations. We checked whether the search string also retrieved the citations in the list, and all 100 were retrieved by the string. Although this is a small sample, it indicates the robustness of the search string.

3.2. Selection Filtering

It is unlikely that all 2,805 publications would be relevant. Therefore, we next applied a series of filtering steps to obtain a focused sample. Figure 5 presents the filtering process and the number of resulting entries after applying each filter. The number in box 1 represents the initial number of articles. The numbers in the the other boxes represent the number of entries removed in that particular step. The numbers between the steps show the total number of articles after applying the previous step.

To ensure that publications are relevant, we used keywords to filter the list. We first searched the title and abstract of each study for the keyword "test" (including root usage, e.g., "testing"). We then searched the remaining publications for either "learning" or "neural"—representing the application of ML. We merged the filtered lists for both keywords, and removed all duplicate entries. We then removed all secondary studies

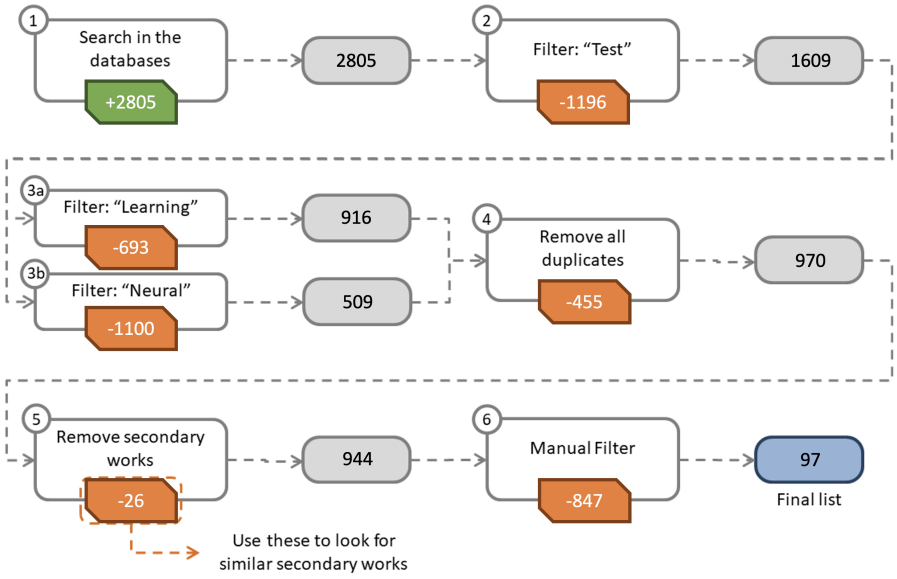


Figure 5: Steps taken to determine the final list of publications to analyze.

from the article list. This removed an additional 26 articles, leaving 944 publications.

We examined the remaining publications manually, removing all publications not in scope following an inspection of the title and abstract. We removed any publications not related to software test generation or that do not apply ML during the test generation process (e.g., ML is used in a separate activity such as test suite reduction, a non-ML approach is used such as metaheuristic search, or an ML algorithm is being tested). This determination was made by first reading the abstract and introduction. Then, if the publication seemed in scope, we proceeded to read the entire study. In a small number of cases, publications were deemed out-of-scope only after close inspection. Both authors independently inspected publications during this step to prevent the accidental removal of relevant publications. In cases of disagreement, the authors discussed each study and came to a conclusion.

This process resulted in a final sample of 97 articles. Due to the size and scope of the sample, we have not performed snowballing as part of this study. We believe that this sample is sufficiently broad to characterize research in this field.

The publications are listed in Section 4.2, associated with the specific testing prac-

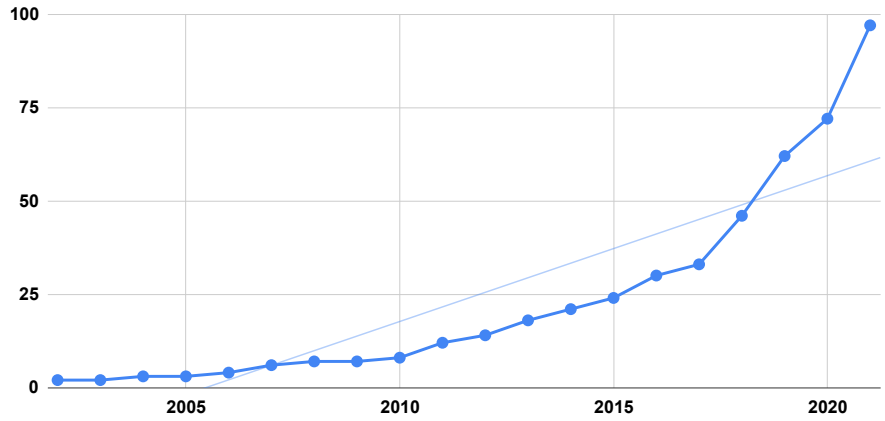


Figure 6: Growth of the use of ML in test generation from 1993-2021.

tice addressed. Figure 6 shows the growth of interest in this topic from 1993—when the oldest study was published—to the present day. We can see modest, but growing, interest from 1993 to 2010. The advancements in ML in the past decade have resulted in significantly more interest in the use of ML in test generation, especially starting in 2018. Over two-thirds of the publications in our sample were published in the past five years alone. This is an area of growing interest and maturity, and we expect the number of publications to increase significantly in the next few years.

3.3. Data Extraction

To answer the questions in Table 1, we have extracted a set of key properties from each study, identified in Table 2. Each property listed in the table is briefly defined and is associated with the research questions. Several properties may be collectively used to answer a RQ. For example, RQ2—covering the goals of using ML—can be answered using property P2 in many cases. However, P1 provides context to the research and the particular testing practice addressed may dictate how ML is applied.

In reported experiments, the proposed approach either exceeded or failed to meet the initial hypotheses. This is covered by the third property, P3, which could lead to or be part of the answer for RQ1 and RQ3. The fourth property targets RQ3 and notes how the adopted ML technique is integrated into the testing process. To understand how ML

ID	**Property Name**	**RQ**	**Description**
P1	Testing Practices Addressed	RQ1, RQ2	The specific type of testing scenarios or application domain focused on by the approach. It helps to categorize the publications, enabling comparison between contributions.
P2	Proposed Research	RQ2	A short description of the approach proposed or research performed.
P3	Hypotheses and Results	RQ1, RQ3	Highlights the differences between expectations and conclusions of the proposed approach.
P4	ML Integration	RQ3	Covers how ML techniques have been integrated into the test generation process. It is essential to understand what aspects of generation are handled or supported by ML.
P5	ML Technique Applied	RQ4	Name, type, and description of the ML technique used in the study.
P6	Reasons for Using the Specific ML Technique	RQ4	The reasons stated by the authors for choosing this particular ML technique.
P7	ML Training Process	RQ4	How the approach was trained, including the specific data sets or artifacts used to perform this training. This property helps us understand how each contribution could be replicated or extended.
P8	External Tools or Libraries Used	RQ4	External tools or libraries used to implement the ML technique.
P9	ML Objective and Validation Process	RQ4, RQ5	This attribute covers the objective of the ML technique (e.g., reward function or validation metric), and how it is validated, including data, artifacts, and metrics used (if any).
P10	Test Generation Evaluation Process	RQ5	Covers how the ML-enhanced oracle generation process, as a whole, is evaluated (i.e., how successful are the generated input at triggering faults or meeting some other testing goal?). Allows understanding of the effects of ML on improving the testing process.
P11	Potential Research Threats	RQ6	Notes on the threats to validity that could impact each study.
P12	Strengths and Limitations	RQ6	This property is used to understand the general strengths and limitations of enhancing a generation process with ML by collecting and synthesizing these aspects for both the ML techniques and entire test generation approaches.
P13	Future Work	RQ6	Any future extensions proposed by the authors, with a particular focus on those that could overcome the identified limitations.

Table 2: List of properties used to answer the research questions. For each property, we include a name, the research questions the property is associated with, and a short description.

techniques can enhance automated test generation, it is important to understand which techniques are applied as well as the motivation behind adopting a specific technique. These aspects are covered by P5 and P6, which are used to answer RQ4. We also note whether the research is new or a continuation of prior research.

The next three properties focus on understanding the application of ML in the study, including a partial assessment of the potential to replicate the research, by covering

core characteristics of the ML technique—the training process (P7), external tools used to implement the technique (P8), and the validation process (P9). P7 focuses on the datasets or other information sources used to train the learning technique. Our primary focus with P8 is to cover how external tools, environments, or ML libraries—e.g., PyTorch—are used to train, build, or execute the ML technique. The combination of properties P7, P8, and P9 will answer RQ4, which examines how the ML technique is trained, validated, and assessed as part of its integration. RQ5 examines how the full test generation process is evaluated, and is answered using P9 and P10.

Research question RQ6 covers the limitations of ML-enhanced test generation techniques and open challenges. Properties P11 and P12 contribute to answering this question, including both the limitations identified by the authors and threats to validity—either disclosed by the authors or inferred from our analysis. Finally, P13 centers around major challenges to overcome in ML-enhanced test generation.

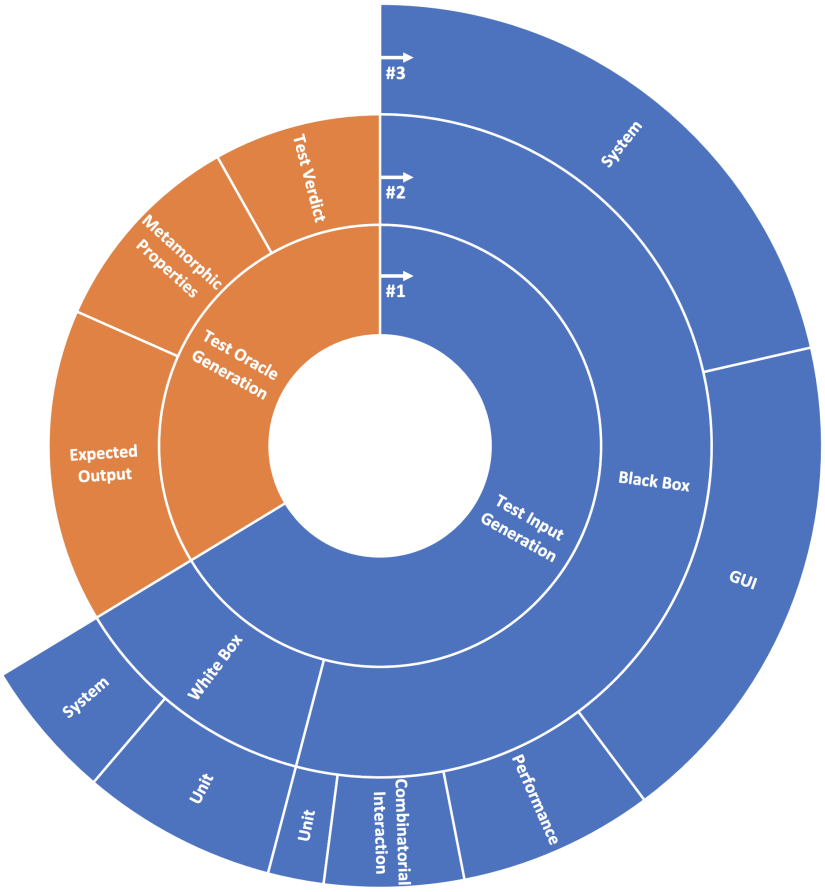
Data extraction was performed primarily by the first author of this study. However, to ensure the accuracy of the extraction process, the second author performed an independent extraction for a sample of ten randomly-chosen publications. We compared our findings, and found that we had near-total agreement on all properties. The second author then performed an independent verification of the findings of the first author for the remaining publications. A small number of corrections were discussed between the authors, but the data extraction was generally found to be accurate.

4. Results and Discussion

In this section, first, we identify the testing practices addressed by ML-enhanced test generation (RQ1, Section 4.1). We then note observations for individual testing practices (Section 4.2). Finally, we present answers to RQ2-6 (Section 4.3).

4.1. RQ1: Testing Practices Addressed

There are many different types of tests. Therefore, the purpose of our first research question is to give an overview of which testing practices have been targeted by the sampled publications. Figure 7 provides an overview. In this sun chart, we divide



Layer #1	
Test Input Generation	65
Test Oracle Generation	33

Layer #2	
Black Box	53
White Box	12
Expected Output	15
Metamorphic Properties	10
Test Verdict	8

Layer #3	
System Test Generation (Black Box)	21
GUI Test Generation	18
Performance Test Generation	7
Combinatorial Interaction Testing	5
Unit Test Generation (Black Box)	2
Unit Test Generation (White Box)	7
System Test Generation (White Box)	5

Figure 7: Testing practices addressed by test generation approaches incorporating ML.

articles into successive layers, with each layer representing finer levels of granularity. The total number of publications in each category is reported below the figure.

The specific formulation of a test case depends on the product domain and technologies utilized by the SUT [3]. However, broadly, a test case is defined by a set of

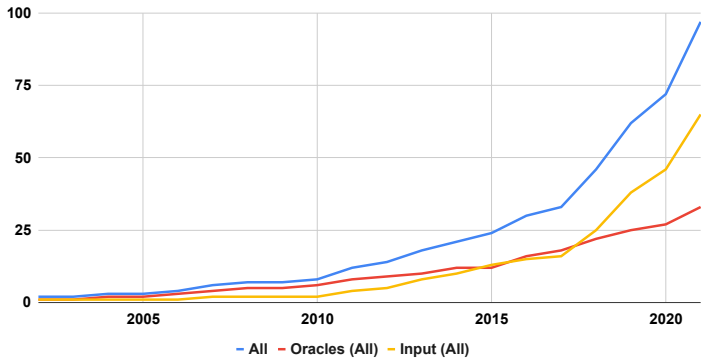


Figure 8: Growth of the use of ML for input and oracle generation since 2002.

input steps and test oracles [7], both of which can be the target of automated generation. Therefore, *input* and *oracles* constitute our first division of the publications.

A majority of articles focus on input generation (67% of publications). Automated input generation has become a major research topic in software testing over the past 20 years [6], and many different forms of automated generation have been proposed [8], using approaches ranging from symbolic execution [28] to optimization [29].

Oracle generation has long been seen as a major challenge for test automation research [7, 6], even earning the name of the "test oracle problem". Oracle generation is a much more difficult challenge to solve, and comparatively fewer approaches have been proposed [30]. However, ML is a realistic route to achieve automated oracle generation [7], and publications have started to appear (33% of our sample).

RQ1 (Testing Practices): ML supports generation of both test input and oracles, with a greater focus on input generation (67% of the sample).

Figure 8 shows the growth in both topics since 2002 (only one article was published prior to 2002). Both show a similar trajectory until 2017, with a sharp increase in input generation after. New ML technologies, such as deep learning, and the growing maturity of open source learning frameworks, such as OpenAI Gym, have potentially contributed to this increase.

4.1.1. Test Input Generation

In the second layer of Figure 7, we divide test input generation by the source of information used to create test input:

- **Black Box Testing:** Also known as **functional testing** [3], approaches in this category use information about the program gleaned from documentation, requirements, user manuals, and other project artifacts to create test inputs.
- **White Box Testing:** Also known as **structural testing** [3], approaches in this category use the structure of the source code to select test inputs (e.g., generating input that achieves a particular outcome for an `if`-statement). Approaches in this category require access to source code, but do not require domain knowledge.

Of the 65 publications addressing test input generation, 53 propose Black Box approaches and 12 propose White Box approaches. White Box approaches are very common in test generation, as the "coverage criteria"—checklists of testing goals [4]—that are the focus of White Box testing offer measurable optimization targets for automated generation [31]. Such approaches can benefit from the inclusion of ML [32]. However, ML may have greater potential to enhance Black Box testing. Such approaches are based on domain and project data about how the system should behave. ML opens new opportunities to extract or exploit that data as part of input generation, as shown by 82% of input generation publications offering Black Box approaches.

The third layer of Figure 7 further subdivides approaches based on testing practice:

- **System Test Generation (26 publications):** Tests target a system interface (e.g., CLI or API) and verify high-level functionality. Tests generated at this level target either the system as a whole, or subsystems with an external interface.
- **GUI Test Generation (18 publications):** A form of system testing where tests target a graphical user interface to identify both incorrect functionality and responsiveness, graphical issues, or accessibility issues (e.g., broken links) [33].
- **Unit Test Generation (9 publications):** Tests target a single class and exercise its functionality in isolation from the rest of the system [34].

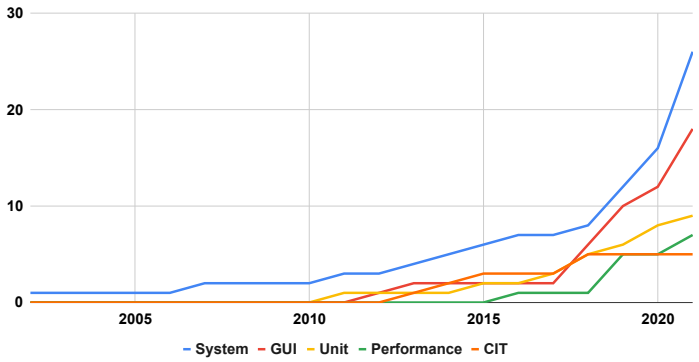


Figure 9: Growth of the use of ML for different forms of input generation since 2002.

- **Performance Test Generation (7 publications):** Tests focus on the speed, throughput, and responsiveness of the SUT, often examining system behavior when computational resources like CPU, memory, or disk capacity are varied [35].
- **Combinatorial Interaction Testing (5 publications):** A system-level test generation technique that produces a minimal set of test cases that cover important interactions between input variables [36].

System-level testing is the most common category (40% of input generation publications). This is followed by GUI (28%), then unit testing (14%). The remaining categories—performance (11%) and combinatorial interaction testing (CIT) (8%)—represent specialized forms of system testing with their own distinct challenges.

RQ1 (Testing Practices): Input generation practices include system testing, specialized types of system testing (GUI, performance, CIT), and unit testing. The majority of these are Black Box approaches, with White Box approaches primarily restricted to unit testing.

Figure 9 shows the growth in the number of publications in each sub-area of input generation since 2002. We see a particularly strong growth in both system and GUI testing since 2017. In addition to the emergence of open-source ML frameworks, we also hypothesize that this growth is partially driven by the emergence of new types

of systems, including mobile and web applications and autonomous vehicles. Mobile applications are tested primarily through a GUI, as are many web applications—leading to increased interest in GUI testing. Other web applications are tested through a REST API, and are included in the system testing category. Autonomous vehicles also require new generation approaches, as they are tested in complex simulators [37].

RQ1 (Testing Practices): There has been an increase in publications on system and GUI input generation since 2017, potentially related to the emergence of web and mobile applications and autonomous driving, as well as to the availability of robust, open-source ML and deep learning frameworks.

4.1.2. Test Oracle Generation

The second layer under test oracle generation in Figure 7 divides approaches based on the *type* of test oracle produced:

- **Expected Output (15 publications):** The oracle predicts concrete output behavior that should result for a given input. Often, this will be an abstracted output (i.e., rather than exact output, a *type* of output).
- **Metamorphic Relations and Other Properties (10 publications):** A metamorphic relation is a property of a function, relating input to output produced [38]. For example, a relation for sin is $sin(x) = sin(\pi - x)$. Such properties, as well as other properties of program behavior, can be applied to many inputs. Violations of properties identify potential faults.
- **Test Verdicts (8 publications):** The oracle predicts the final test verdict for a given input (i.e., a "pass" or "fail" verdict for the test case).

RQ1 (Testing Practices): ML supports generation of test verdict, metamorphic (and other property-based), and expected output oracles.

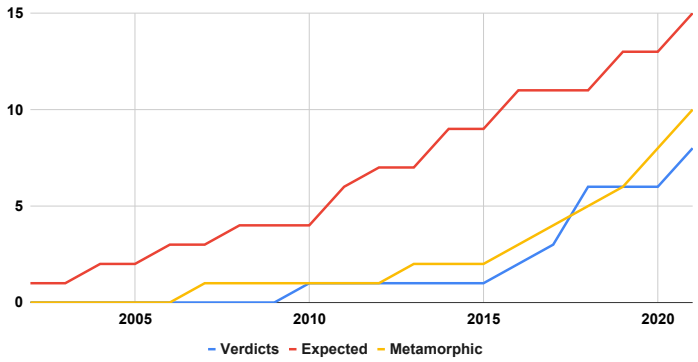


Figure 10: Growth of the use of ML for different forms of oracle generation since 2002.

Figure 10 shows the growth in the number of publications in each type of oracle generation since 2002. We see a steady growth over time in research on expected value oracles, with later emergence of research on test verdicts and metamorphic properties.

4.2. Examining Specific Practices

Before we answer the remaining research questions, we will examine how ML has supported input or oracle generation for each of the individual areas identified in RQ1.

4.2.1. System Test Generation

A total of 26 publications propose a system testing approach. Tables 3-4 outline the 21 Black Box approaches, while Table 5 outlines the 5 White Box approaches. Each table lists the year, type of ML (RL, supervised, semi-supervised, or unsupervised), specific ML technique, training data (if applicable), objective of using ML, evaluation metrics used, and the types of systems used in the evaluation. Each table is sorted by the ML type, then by the first author's last name. When discussing the objective, we indicate both the type of prediction (classification, regression, clustering, or reward function) and a summary of the purpose of the prediction. We will first discuss approaches that *generate test input directly*, then cases where ML is used to *enhance existing test generation methods*.

Input Generation (Supervised, Semi-Supervised): Supervised approaches are generally used to train models that associate input with particular qualities of interest.

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[39]	2016	RL	Q-Learning	N/A	Reward (Plan Coverage)	Code Coverage, Assertion Coverage	Robotic Systems
[40]	2021	RL	Q-Learning	N/A	Reward (Criticality)	Faults Detected	Autonomous Vehicles
[41]	2013	RL	Delayed Q-Learning	N/A	Reward (Test Improvement)	% of Runs Where Requirements Met	Ship Logistics
[42]	2021	RL	Q-Learning	N/A	Reward (Code Coverage)	Code Coverage	Triangle Classification, Nesting Structure, Complex Conditions
[43]	2020	RL	Deep RL	N/A	Reward (Transition Coverage)	Not Evaluated	OpenAPI APIs
[44]	2020	RL	Monte Carlo Control	N/A	Reward (Input Diversity)	Input Diversity, Code Coverage	XML, JavaScript Parsing
[45]	2021	Semi-supervised	GAN, CNN	Image Input	Regression (Speed)	Faults Detected	Autonomous Vehicles
[46]	1993	Supervised	Not Specified	System Executions	Regression (Output)	Not Evaluated	N/A
[47]	2018	Supervised	Backpropagation NN	System Executions	Regression (Output)	Output Coverage	Train Controller
[48]	2021	Supervised	Gaussian Process, Decision Trees, AdaBoostedTree, Random Forest, SVM, ANN	System Executions	Regression (Output)	Accuracy	Power Grid Control
[49]	2019	Supervised	LSTM NN	Existing Inputs	Regression (Valid Input)	Accuracy, Code Coverage	FTP Programs

Table 3: Publications 1-11 under **System Test Generation (Black Box)** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation.

Papadopoulos et al. [54], Budnik et al. [47], and Bergadano et al. [46] infer a model from system execution logs containing inputs and resulting output. The model is then used to predict input leading to an output of interest. For example, Budnik et al. identify small changes in input that lead to large differences in output, indicating boundary areas where faults are likely to emerge. Budnik and Bergadano suggest comparing predictions with real output, and using misclassifications to indicate the need to re-train.

Another concern is achieving code coverage. Majma et al. [63] use supervised NNs for input and oracle generation. A Backpropagation NN is trained to associate inputs with paths covered through the source code, then generates inputs that execute uncovered paths. Utting et al. [57] cluster log files with information on system execution—

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[50]	2019	Supervised	CRF	Test Descriptions	Regression (Requirement Associations)	Accuracy	Telecom Systems
[51]	2019	Supervised	LSTM NN	Existing Inputs	Regression (Failing Input)	Faults Detected, Efficiency	Smart TV
[52]	2021	Supervised	Parallel Distributed Processing	System Executions	Regression (Output)	Efficiency, Faults Detected, Model Size	Autonomous Vehicles
[53]	2021	Supervised	MLP	System Executions	Classification (Input Validity)	Accuracy	REST APIs (GitHub, LanguageTool, Stripe, Yelp, YouTube)
[54]	2015	Supervised	C4.5	Existing Inputs	Regression (Output)	Mutation Score	Triangle, BMI, Air Traffic Control
[55]	2020	Supervised	LSTM NN	Simulink models	Regression (Validity Rules)	Input Validity, Faults Detected	Simulink tools
[56]	2021	Supervised	CRF	Specifications	Regression (Requirement Associations)	Accuracy	Unspecified
[57]	2020	Supervised, Unsupervised	Decision Trees, Gradient Boosting, K-Nearest Neighbor, MeanShift	System Executions	Regression (Validity Rules), Clustering (Covered Input)	Num. Clusters, Accuracy, Event Coverage	Bus System, Supply Chain
[58]	2007	Supervised	Backpropagation NN	System Executions	Regression (Output)	Accuracy, Efficiency	Fault Tolerant System, Arc Length
[59]	2019	Supervised	SVM	Existing Inputs	Regression (Validity Rules)	Tests Generated, Tests Executed, Test Size, Faults Detected	Domain-Specific Compiler

Table 4: Publications 12-21 under **System Test Generation (Black Box)** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation.

gathered from customer reports—using the MeanShift algorithm, then compare these clusters to logs from executing existing test cases to identify weakly-tested areas of the SUT. Supervised learning is then used to fill in these gaps. The traces are formatted as vectors of actions using a bag-of-words algorithm, and a learned model is used to predict the next input action in the sequence.

Others predict input that will fail. Kirac et al. [51] train a LSTM NN to identify usage behaviors likely to lead to failures using existing test cases. Eidenbenz et al. [48] create a randomly-generated set of inputs and label them based on whether they failed, then cluster the failing instances to enhance accuracy. They train a model using several

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[60]	2021	RL	ReLU Q-Learning	Constraints	Reward (Solving Cost)	Code Coverage, Queries Solved	GNU coreutils
[61]	2021	RL	Deep Q-Network	N/A	Reward (Code Coverage, Path Length)	Code Coverage	Sorting
[62]	2021	Supervised	LSTM NN, Tree-LSTM, K-Nearest Neighbour	Constraints	Regression (Solving Time)	Accuracy, Constraint Solving Time	GNU coreutils, Busybox utils, SMT-COMP
[63]	2014	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage	Binary Search, Sorting, Median, GCD, Triangle Class.
[64]	2011	Supervised	Backpropagation NN	Existing Inputs, Code Coverage	Regression (Code Coverage)	Code Coverage	Triangle Classification

Table 5: Publications under **System Test Generation (White Box)** category and their year of publication, ML approach, ML technique, data used to train (if applicable), target goal of the ML approach, evaluation metrics, and the applications used in the evaluation.

algorithms, then compare their ability to generate failing input. They propose an iterative process where more training data is added over time, and predictions are verified by developers. They find that Gaussian Process regression was the most accurate.

Several authors generate input based on models inferred from requirements or other specifications of program behavior. These tests can show that specifications are met. Kiruma et al. [50] use Conditional Random Fields (CRF) to associate test cases and requirements, creating a training set where requirements are tagged with contextual information indicating output that will follow if the conditions are met. CRF identifies associations between actions, conditions, and outputs in the tagged specifications, then generates new test cases with inputs, conditions, and expected output. Ueda et al. [56] also use CRF to classify elements of a natural language specification. Their method takes new specifications in natural language and transforms them into a structured abstract test recipe that can be concretely instantiated with different input.

Meinke et al. [52] learn a model of system execution. Use cases are modeled in a constraint-based language and used to generate input from the model based on the constraints. Deng et al. [45] use behavioral properties written by human testers to generate new input intended to violate those properties. They present an adversarial scenario

where NNs work in opposition to each other—the output of one network is applied to a second, which offers feedback that enhances the performance of the first network. This allows the use of a small initial training set. They use three Generative Adversarial Networks (GANs) and a Convolutional Neural Network (CNN) to manipulate image data used as input to an autonomous driving system. Collectively, these models predict which input will violate behavioral properties through different translations (e.g., changing day to night, adding rain).

Finally, multiple authors use supervised approaches to generate complex input for particular system types. For example, Shrestha [55] train a Long Short-Term Memory (LSTM) NN to generate valid Simulink models for testing tool-chains based on the modelling language. Simulink is a visual language for modelling and simulation.

In compiler testing, an input is a full program, resulting in a large space of potential inputs. Zhu et al. [59] use ML to restrict the range of inputs to avoid wasted effort. They focus on specialized compilers for restricted domains, and try to generate input appropriate for that domain. They extract features from the source code, such as the number of loops or matrix operations, then train a Support Vector Machine (SVM) to predict whether a new test case belongs to the domain of interest. Test cases not belonging to that domain are discarded.

Protocol test generation requires text-based input that conforms to a specified format. Often, determining conformance requires manual construction of a grammar. Gao et al. [49] use of ML to generate protocol test input without a pre-defined grammar. They use seq2seq-attention—an encoder-decoder model that uses a LSTM NN to transform an input sequence of indeterminate length into a uniform semantic feature and decode it. This model learns the probability distributions of every character of a message, enabling generation of new valid text sequences.

Input Generation (Reinforcement Learning): Both Araiza-Illan et al. [39] and Huurman et al. [43] use RL to generate input to cover states of a model. Araiza-Illan generate input for robots. The agent operates on beliefs (facts about the environment), desires (goals), and intentions (interaction plans). Q-Learning is used to explore the robot's environment, using coverage of plan models as the reward function. Huurman

et al. model APIs as stateful systems—where requests trigger transitions—and use ML to generate API calls intended to cover all states. They adopted Deep RL, and target APIs specified in the OpenAPI format, using transition coverage as the reward.

Baumann et al. [40] use RL to select input for autonomous driving systems intended to violate critical requirements. The reward function encapsulates headway time, time-to-collision, and required longitudinal acceleration to avoid a collision.

Reddy et al. [44] use RL to generate valid complex input (e.g., structured documents). They use a tabular, on-policy RL technique—Monte Carlo Control—where the reward function favors unique and valid input. As uniqueness depends on previously-generated input, this is not a problem that can easily be solved with supervised learning.

Enhancing Test Generation: ML can be used to improve efficiency or effectiveness of test generation methods. A common target for improvement are Genetic Algorithms (GAs). A GA generates test cases intended to maximize or minimize a fitness function—a domain-specific scoring function, much like the reward function in RL.

Buzdalov et al. [41] use RL to modify the fitness function, adding and tuning sub-objectives that will assist in optimizing the core objective of the search. Zhao et al. [58] replace the fitness function with a Backpropagation NN that predicts which input will cover unseen output behaviors. Mishra et al. [64] also replace the fitness function, training a Backpropagation NN to predict the statements that will be covered by a test input. This model is intended to be used when there is no tool support to measure code coverage, or in cases where measuring coverage would be expensive.

Esnaashari et al. [42] use RL to manipulate test cases within the GA by modifying input. Paduraru et al. [61] similarly use RL to improve the effectiveness of a fuzzing tool. The RL agent modifies input selected by the core fuzzing algorithm to improve either code coverage or longest execution path length.

Mirabella et al. [53] use a Multilayer Perceptron (MLP)—a basic NN, often with a single hidden layer—to predict input validity for REST APIs. This approach would allow a generation framework to filter invalid input before applying it.

Luo et al. and Chen et al. both enhance symbolic execution. Luo et al. [62] focus on improving the efficiency of constraint solving. Normally, a fixed timeout is used.

They used both offline learning, based on LSTM and Tree-LSTM, as well as an online approach, based on K-Nearest Neighbours, to predict time needed to solve a constraint. Chen et al. [60] also examine constraint solving, using ReLU Q-Learning to identify the optimal solving strategy for a constraint. Constraint solving is modeled as a Markov Decision Process and the RL agent is first trained offline, then is applied online.

4.2.2. GUI Test Generation

Table 6 details the 18 GUI testing publications. Of these, twelve focus on mobile, four on web, and two on desktop applications. GUI test generation often focused on a state machine model of the interface that formulates changes to the display as transitions taken following input actions. Fifteen publications used ML to generate input to cover the states of this model.

Almost all of these publications adopted RL, as it can learn from feedback after applying an action to the GUI, and many GUIs require a sequence of actions to access certain visual elements. The most commonly used RL technique is Q-Learning [76, 74, 71, 65, 77, 67, 77, 78], which associates the value of an action with particular states, but can handle stochastic transitions. Q-Learning uses the same reward function to evaluate actual and projected actions. This can exaggerate estimations of future actions. Double Q-Learning, as used by Koroglu and Sen [72, 73], attempts to correct this issue by using a different function to estimate future rewards. Collins et al. [69] adopted a deep convolutional neural network to guide RL—a Deep Q-Network.

The main difference between these publications lies in the reward function. Many base the reward function in part on state coverage (e.g., [77]), while also incorporating additional information to bias selection of states. These additional factors include the magnitude of the change in state [69], usage specifications and expectations [71, 72], unique code functions called [74], a curiosity factor—favoring exploration of new elements [75, 78]—coverage of different means of interaction (e.g. click, double click, drag, hold) [70], and avoidance of loops in navigation (e.g., where “home” or “back” buttons are pressed and previous actions are re-executed) [68].

Rather than the state coverage, Koroglu and Sen [73] use the reward function to find violations of specifications written in a formal logic. Ariyurek et al. [66] also apply

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[65]	2018	RL	Q-Learning	N/A	Reward (State Cov.)	State Coverage	Android Apps
[66]	2021	RL	Monte Carlo Tree Search, Sarsa	N/A	Reward (Test Goal Coverage)	Faults Detected	2D Games
[67]	2021	RL	Q-Learning	N/A	Reward (State Cov.)	Qualitative	Resource Planning
[68]	2013	RL	Not Specified	N/A	Reward (State Coverage, Loop Interactions)	State Coverage	Android Apps
[69]	2021	RL	Deep Q-Network	N/A	Reward (State Change Magnitude)	Code Coverage, Faults Detected	F-Droid
[70]	2019	RL	Not Specified	N/A	Reward (State Cov., Element Interaction)	State Coverage	F-Droid
[71]	2018	RL	Q-Learning	N/A	Reward (State Cov., Specifications)	State Coverage	F-Droid
[72]	2020	RL	Double Q-Learning	N/A	Reward (State Cov., Specifications)	State Coverage	F-Droid
[73]	2021	RL	Double Q-Learning	N/A	Reward (Specifications)	Faults Detected	F-Droid
[74]	2012	RL	Q-Learning	N/A	Reward (State Cov., Calls)	State Coverage	Password Manager, PDF Reader, Task List, Budgeting
[75]	2020	RL	Q-Learning + LSTM	N/A	Reward (State Cov., Curiosity)	State Coverage	Android Apps
[76]	2018	RL	Q-Learning	N/A	Reward (State Cov.)	State Coverage	Android Apps
[77]	2021	RL	Q-Learning	N/A	Reward (State Cov.)	Code Coverage, Faults Detected	Android Apps
[78]	2021	RL	Q-Learning	N/A	Reward (State Cov., Curiosity)	Code Coverage, Faults Detected, Scalability	Web Apps (Research, Real-World, Industrial)
[79]	2019	Supervised	Deep NN	System Executions	Regression (Action Probability)	State Coverage	Android Apps
[80]	2018	Supervised	Recurrent NN	Existing Inputs	Regression (Test Flows)	State Coverage	Unspecified Web App
[81]	2019	Supervised	Random Forest	Web Pages	Classification (Page Elements)	Mutation Score	Task List, Job Recruiting Web Apps
[82]	2019	Supervised	Feedforward ANN	Generated Inputs	Regression (Output)	State Coverage	Login Web App

Table 6: Publications under the **GUI Test Generation** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation. All approaches in this category are Black Box.

RL—based on Monte Carlo Tree Search and Sarsa—to select input for grid-based 2D games. The game state is represented as a graph, and "test goals" are synthesized from the graph. The reward function emphasizes coverage of test goals.

Li et al. targeted mobile applications using supervised learning [79], training a model to mimic human interaction based on recorded user interaction logs. The authors used a Deep Neural Network (DNN), a complex NN with several hidden layers. The DNN associates GUI elements with a probability of usage by a human, using those probabilities to bias the selection of actions.

Santiago et al. [80, 81] use supervised ML to generate "test flows"—sequences of interactions—for web applications. Their approach is trained using human-written test flows spanning several webpages. They use a Recurrent Neural Network. Their second study extends the approach to interact with forms by using Natural Language Processing (NLP) to extract feedback messages when interacting with forms [81]. The framework learns constraints for form input, and a constraint solver creates input that meets those constraints. A Random Forest is used to classify page components before applying NLP. This helps control how NLP is applied to different component types. Their approach could, potentially, be faster and more accurate than RL. However, it requires a complex training phase and a large set of human-created input. Models can potentially be used for multiple websites, decreasing the training burden.

Kamal et al. use ML to filter redundant test cases as part of enhancing a search-based test generation framework for web applications [82]. They use a Feedforward ANN—a simplistic NN, with no cycles or loops in neuron connections—to associate input and output, then uses the predicted output to decide if tests are redundant.

4.2.3. Unit Test Generation

Unit test generation focuses on individual classes, rather than interfaces. This means that the majority of publications listed in Table 7 are "White Box" approaches, and that they are not tied to particular system types (e.g., web applications) as—at the unit level—domain concerns are less applicable.

Groce [86] and Kim et al. [13] use RL to generate input with the goal of attaining code coverage. Both employ code coverage as the reward function. In the former case, the RL agent generates input for the function directly. In the latter case, a Double Deep Q-Network is used to generate optimization algorithms. The optimization algorithms generate test input, which is manipulated over several rounds of evolution.

Ref	Year	Test Gen. Approach	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[83]	2017	Black	Supervised	Query Strategy Framework	System Executions	Regression (Output)	Mutation Score	Math Library, Time Library
[84]	2018	Black	Unsupervised	Backpropagation NN	Existing Inputs	Clustering (Input Similarity)	Not Evaluated	N/A
[32]	2020	White	RL	UCB, DSG-Sarsa	N/A	Reward (Num. Exceptions)	Num. Exceptions, Faults Detected	Compiler, Math Library, String Library, Time Library, Spreadsheet, Mocking Library
[85]	2020	White	RL	UCB, DSG-Sarsa	N/A	Reward (Input Diversity)	Input Diversity, Faults Detected	JSON Parser
[86]	2011	White	RL	Not Specified	N/A	Reward (Code Coverage)	Code Coverage	Data Structures
[87]	2015	White	RL	Q-Learning	N/A	Reward (Code Coverage)	Code Coverage	Data Structures, Collection Library, Primitives Library, Java/XML Parsers
[13]	2018	White	RL	Double Deep Q-Network	N/A	Reward (Code Coverage)	Code Coverage, Efficiency	GCD, EXP, Remainder
[88]	2021	White	Supervised	Gradient Boosting	Code Metrics	Classification (Fault Prediction)	Accuracy, Faults Detected	Compression, Imaging Library, Math Library, NLP, String Library
[89]	2019	White	Supervised	Backpropagation NN	Existing Inputs	Regression (Code Coverage)	Not Evaluated	N/A

Table 7: Data on the sampled publications in the **Unit Test Generation** category and their year of publication, test generation approach, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation.

The RL agent manipulates the heuristics used by the optimization algorithm. Their test execution platform is compatible with the OpenAI Gym and keras-rl RL libraries.

Walkinshaw et al. [83] use a supervised approach to generate input, with the goal of testing parts of the system that have only been weakly tested. A model is trained to predict the output of generated input. Intuitively, the model will have more confidence in the prediction accuracy for input similar to that in the training data. Therefore, input with low certainty is retained, as they are likely to test parts of the system that were ignored in the training data. Even if these inputs do not reveal faults, they can be fed back into the training data to re-train the model, shifting the focus to other parts of the system. Their implementation uses genetic programming rather than a ML algorithm.

Many of the authors propose the use of ML to enhance existing unit test genera-

tion approaches—often based on genetic algorithms (GA)—rather than using ML to directly generate input. Almulla and Gay [32, 85] use RL to adapt the test generation strategy of a GA. The RL agent selects the fitness functions optimized by the GA, with the goal of identifying fitness functions that increase the number of exceptions thrown [32] and the test suite diversity [85]. They use the Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa) algorithms.

He et al. [87] present a Q-Learning technique to improve the coverage of private and inherited methods by augmenting externally generated tests. The RL agent can make two types of changes to test cases. It can replace a method in a call sequence with another method whose return type is a subclass of the original method's. It can also replace a call to a public method with a call to another method that can call a private method. The reward function is focused on coverage of hard-to-cover methods.

Hershkovich et al. [88] use classification algorithm to predict whether a class is likely to contain a fault. This can improve the efficiency of test generation by determining which classes to target. They learn using source code metrics, labelled based on whether a class had past faults. They use Gradient Boosting—a method that integrates multiple learners to minimize training errors.

Ji et al. [89] use supervised learning to replace a costly fitness evaluation in a GA. They focus on code coverage based on data-flow. Calculating data flow coverage is very expensive. Therefore, a model can be trained to predict coverage for new input generated by the GA, replacing the need to actually measure data flow. They use a Backpropagation NN (BPNN) to train this model. Hooda et al. [84] also propose the use of ML to improve efficiency of a GA. They also use a BPNN to cluster test cases. When the algorithm generates new test cases, those that fall too close to the centroid of a cluster are rejected to reduce redundancy.

4.2.4. Performance Test Generation

Performance testing focuses on exposing quality issues, such as slow operations. Table 8 details the 7 publications in this category. Performance can generally be measured, which offers immediate feedback for use in subsequent rounds of test generation. As a result, the majority of approaches in this category are based on iterative processes,

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[90]	2019	RL	Dueling Deep Q-Network	N/A	Reward (Execution Time)	Identified Bottlenecks	Auction Website
[91]	2019	RL	Q-Learning	N/A	Reward (Path Length, Feasibility)	Paths Explored, Efficiency	Biological Computation, Parser, Sorting, Data Structures
[35]	2019	RL	Q-Learning	N/A	Reward (Response Time Deviation)	Not Evaluated	N/A
[92]	2019	RL	Q-Learning	N/A	Reward (Response Time Deviation)	Not Evaluated	N/A
[93]	2021	Semi-Supervised	Conditional GAN	System Executions	Regression (Perf. Requirements), Classification (Test Realism)	Identified Bottlenecks, Accuracy, Labelling and Training Effort	Auction Website
[94]	2016	Supervised	RIPPER	System Executions	Regression (Rule Learning)	Identified Bottlenecks	Insurance, Online Stores, Project Management
[95]	2021	Supervised	Multivariate Time Series	Session Logs	Regression (Load)	Accuracy	Student Information

Table 8: Data on the sampled publications in the **Performance Test Generation** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation. All approaches are Black Box.

including RL [90, 91, 35, 92], rule learning [94], and adversarial learning [93].

Ahmad et al. [90] use a form of Q-Learning called Dueling Deep Q Networks to generate input that exposes performance bottlenecks. Traditional Q-Learning has trouble adapting to situations where the number of combinations of state and action are large. Dueling DQN uses a neural network to generalize states. The reward function is based on maximization of execution time. The authors note room for improvement by integrating other performance indicators into the reward.

Rather than generating input, Moghadam et al. [35, 92] apply Q-Learning to control the execution environment (an implicit form of input). They identify resource configurations (CPU, memory, disk) where timing requirements are violated, with reward based on response time deviation.

Sedaghatbaf et al. [93] generate input using a Conditional GAN. This technique dynamically learns to generate input meeting conditions—in this case, violating performance requirements—using two competing neural networks. The generator produces

input, and the discriminator attempts to classify whether the input violates requirements. This feedback improves the generator.

Luo et al. [94] use the RIPPER rule learning algorithm to identify classes of input that lead to intensive computations. The algorithm identifies and refines rules in an iterative process. When tests are executed, data is collected on method invocations and execution time, then executions are clustered into "good" and "bad" cases, where "good" cases consume the most time. RIPPER learns and refine rules differentiating the clusters, which are then used to generate new input.

Schulz et al. [95] generate workloads (implicit input) for load testing. The learned model generates realistic load levels on a system at various times and in different scenarios. Past session logs are clustered, and a multivariate time series is applied to predict the system load over the performance of a scenario.

Finally, Koo et al. [91] use RL to improve the effectiveness of symbolic execution during stress testing. They seek to identify input that triggers worst-case execution time, defined as inputs that triggers a long execution path. The authors use Q-Learning to control the exploration policy used by the symbolic execution, to identify a policy that favors long paths. The reward function is based on the path length and feasibility of generating input for that path—whether the path constraint can be solved.

4.2.5. Combinatorial Interaction Testing

Table 9 shows the 5 publications that use ML as part of CIT. Three publications use supervised learning (Artificial Neural Networks, specifically) to generate covering arrays (sets of input that cover all pairwise interactions between value classes of input variables) [97, 98, 99]. For example, Patil et al. [99] use ANN to predict coverage of interactions by a test input. They use this ANN to identify a small covering array.

In the approach by Mudarakola et al. [97], each hidden layer of the ANN is mapped to an input variable, and each node of that layer represents a variable value class. The values are connected by their connection to another variable's value. They do not use the network for prediction, but as a structuring mechanism to generate a covering array. Code coverage is used to prune redundant test cases, resulting in a minimal array.

In a follow-up study [98], they manually construct an ANN using SUT require-

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metrics	Evaluated On
[96]	2015	RL	SOFTMAX	N/A	Reward (Input Combinations)	Covering Array Size, Efficiency	Misc. Synthetic, Real Systems
[97]	2014	Supervised	ANN	Pairwise Input Combinations	Other (Structure Input Space)	Covering Array Size	Web Apps
[98]	2018	Supervised	ANN	Specifications	Other (Structure Input Space), Regression (Output)	Covering Array Size	Temperature Monitoring
[99]	2018	Supervised	ANN	Pairwise Input Combinations	Regression (Input Coverage)	Covering Array Size, Efficiency	Unspecified
[100]	2013	Unsupervised	Expectation-Maximization	System Executions	Clustering (Code Coverage)	Qualitative Analysis	Bubble Sort, Math Functions, HTTP Processing, Banking

Table 9: Data on publications in the **Combinatorial Interaction Testing** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and applications used in the evaluation. All approaches are Black Box.

ments, linking outputs to combinations of input values, with each input node mapping to an input variable, hidden layers linked to conditions extracted from the requirements on input, and output nodes linked to predicted SUT output. The neural network again provides structure—a minimal covering array is generated based on paths through the network. The predicted output from the model is compared to actual output, and differences are used to identify faults.

Jia et al. [96] use RL to tune the generation strategy of a Simulated Annealing-based test generation framework. The RL agent—based on the SOFTMAX selection rule—selects how the Simulated Annealing algorithm mutates a covering array. The reward function is based on the change in the coverage of pairwise combinations after imposing a selected mutation. Their framework adaptively recognizes and exploits policies that improve coverage of pairwise combinations.

CIT assumes that values of input variables are divided into representative value classes. This division is generally done manually, but identifying divisions is non-trivial. Nguyen et al. [100] propose the use of clustering to identify value classes, based on executed code lines (and how many times each line was executed).

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metric	Evaluated On
[101]	2018	Supervised	Adaptive Boosting	System Executions	Classification (Verdict)	Mutation Score	Shopping Cart
[102]	2021	Supervised	CNN	Screenshots	Classification (Verdict)	Accuracy, Faults Detected	Games (Android, iOS)
[103]	2018	Supervised	Backpropagation NN	System Executions	Classification (Verdict)	Mutation Score	Embedded Software
[104]	2017	Supervised	Not Specified	System Executions	Classification (Verdict)	Faults Detected	Automotive Applications
[105]	2018	Supervised	L*	System Executions	Classification (Verdict)	Faults Detected, Efficiency	Platoon Simulator
[106]	2016	Supervised	MLP	System Executions	Classification (Verdict)	Accuracy	User Creation
[107]	2010	Supervised	Backpropagation NN	System Executions	Classification (Verdict)	Mutation Score	Student Registration
[108]	2021	Supervised	MLP + LSTM	System Executions	Classification (Verdict)	Accuracy, Training Data Size	Blockchain Module, Deep Learning Module, Encryption Library, Stream Editor

Table 10: Data for RQ2-5 for articles in the **Test Verdicts Test Oracle Generation** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation.

4.2.6. Test Oracle Generation

We have observed three types of oracles in the sample: oracles that (1) directly issue *verdicts* on a test case, (2) state an *expected output*, either concrete or abstract, or (3), specify *metamorphic relations or other properties* on output.

ML supports decision processes. Given an observation, a ML technique makes a prediction. That prediction can be a decision, or it can offer information needed to make a decision. Test oracles follow a similar model, consisting of the oracle *information*—a set of facts used to issue a verdict—and the oracle *procedure*—the actions taken to arrive at a verdict [109]. ML offers a natural means to replace either of these components. Learned test verdict oracles replace the entire procedure, while expected output and property-based oracles offer information needed to arrive at a verdict.

Tables 10-12 summarize the 33 publications where ML generates test oracles. Immediately, we can see that almost all approaches adopted supervised learning. These publications train oracles using previously-captured and labeled system executions, screenshots, or metadata about source code features. The model then predicts the cor-

rectness of output or specific output behavior that should result from applying an input. Three approaches also apply RL. We will discuss each oracle type in turn.

Test Verdicts: 8 publications apply supervised learning to associate patterns in the training data with a resulting test verdict. Most approaches learn from past system execution. Makondo et al. [106] utilize a Multilayer Perception (MLP). Shahamiri et al. [107] and Gholami et al. [103] utilized Feed-forward Backpropagation NNs. Braga et al. [101] use a classifier based on adaptive boosting. Tsimpourlas et al. use a NN architecture with MLP and LSTM [108].

Chen et al. [102] train a model to identify rendering errors in video games using screenshots of previous faults. This model can predict the presence of graphical glitches. They adopted a supervised Convolutional Neural Network for this task.

Khosrowjerdi et al. [104] combine ML and model checking. A learning algorithm (L*) infers a model from system executions that predicts the output of the SUT. Then, given the model and specifications, a model checker can assess whether each specification is met by model, yielding the final verdict. If a specification is violated, a test case is generated that can be executed on the real system to confirm whether there is a fault. If there is not a fault in the actual system, the test case and its outcome can be used as part of retraining the model. In a follow-up study [105], the authors demonstrate their technique on systems-of-systems—assessing a N-vehicle platoon simulation.

Expected Output: The approaches train on system executions, and then predict the output given a new input. The output is often abstracted to a small set of representative values or limited to functions with an enumerated set of values (i.e., classification), rather than a specific output (i.e., regression). A common application is the "triangle classification problem"—a function that classifies a triangle as scalene, isosceles, equilateral, or not a triangle based on the length of its sides. This is a challenging function, given its branching behavior. However, it has a limited set of outputs, making it a reasonable starting point for oracle generation.

Zhang et al. [123] model a function that judges whether an integer is prime—a binary classification problem. Shahamiri et al. [117, 118] generate oracles for a car insurance application, while Singhal et al. [119] and Vanmali et al. [120] generate

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metric	Evaluated On
[110]	2004	Supervised	Backpropagation NN	System Executions	Classification (Output)	Correct Classifications	Triangle Classification
[111]	2021	Supervised	Regression Tree, SVM, Ensemble, RGP, Stepwise Regression	System Executions	Regression (Time)	Accuracy	Elevator
[112]	2016	Supervised	SVM	System Executions	Classification (Output)	Mutation Score	Image Processing
[113]	2021	Supervised	Regression Tree, SVM, Ensemble, TRGP, Stepwise Regression	System Executions	Regression (Time)	Accuracy	Elevator
[114]	2008	Supervised	Backpropagation NN	System Executions	Classification (Output)	Correct Classifications	Triangle Classification
[63]	2014	Supervised	Backpropagation NN	System Executions	Classification (Output)	Faults Detected	Static Analysis
[115]	2019	Supervised	Deep NN	System Executions	Regression (Output)	Mutation Score	Mathematical Functions
[116]	2011	Supervised	RBF NN	System Executions	Regression (Output)	Correct Classifications	Triangle Classification
[117]	2011	Supervised	MLP	System Executions	Classification (Output)	Mutation Score	Insurance Application
[118]	2012	Supervised	MLP	System Executions	Classification (Output)	Mutation Score	Insurance Application
[119]	2016	Supervised	Backpropagation NN + Cascade	System Executions	Classification (Output)	Accuracy	Credit Analysis
[120]	2002	Supervised	Not Specified	System Executions	Classification (Output)	Mutation Score	Credit Analysis
[121]	2014	Supervised	Backpropagation NN, Decision Tree	System Executions	Classification (Output)	Mutation Score	Triangle Classification
[122]	2006	Supervised	MLP	System Executions	Regression (Output)	Mutation Score	Mathematical Functions
[123]	2019	Supervised	Probabilistic NN	System Executions	Classification (Output)	Correct Classifications	Prime, Triangle Class

Table 11: Data for RQ2-5 for articles in the **Expected Output Test Oracle Generation** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation.

oracles for a credit analysis at a bank. Ding et al. [112] generate oracles for an image processing function that classifies a type of cell from image sections. All of these applications produce output from an enumerated set.

However, other authors [122, 115, 111, 113] generate oracles for functions with integer output. Some have a limited set of outputs (e.g., route length prediction). Others offer an indication that ML can model complex functions with unconstrained output (e.g., average waiting time for an elevator).

Ref	Year	ML Approach	Technique	Training Data	ML Objective	Evaluation Metric	Evaluated On
[124]	2020	RL	Not Specified	N/A	Reward (Relations)	Not Evaluated	Ocean Modeling
[125]	2021	RL	Not Specified	N/A	Reward (Relations)	Not Evaluated	Ocean Modeling
[126]	2020	RL	Contextual Bandit	N/A	Reward (Faults Detected)	Faults Detected	Object Detection
[38]	2018	Supervised	SVM	Code Features	Classification (Property)	Accuracy	Misc. Functions
[127]	2013	Supervised	SVM, Decision Trees	Code Features	Classification (Property)	Mutation Score	Misc. Functions
[128]	2016	Supervised	SVM	Code Features	Classification (Property)	Mutation Score	Misc. Functions
[129]	2021	Supervised	Decision Trees	System Executions	Regression (Conditions)	Accuracy	Android Apps
[130]	2019	Supervised	SVM	Code Features	Classification (Property)	ROC	Matrix Calculation
[131]	2007	Supervised	L*	System Executions	Classification (Violation)	Training Data Size	Handshake Protocols
[132]	2017	Supervised	RBF NN	Code Features	Classification (Property)	Accuracy	Misc. Functions

Table 12: Data for RQ2-5 for articles in the **Metamorphic Properties Test Oracle Generation** category and their year of publication, ML approach, ML technique, data used to train (if applicable), objective of the ML approach, evaluation metrics, and the applications used in the evaluation.

The approaches generally make use of some form of NN. Five use a Backpropagation NN [110, 114, 119, 121, 63]. Three employ MLP [117, 118, 122]. Sangwan et al. use a Radial Basis Function (RBF) NN [116]. Monsefi et. al [115] adopt a Deep NN, which has more input and output layers than a regular NN, with a fuzzy encoder and decoder. Zhang et al. use a probabilistic NN [123]. Two adopted Support Vector Machines (SVM) [110, 111]. Arrieta et al. and Gartziandia et al. compared regression trees, SVM, an ensemble model, a Regression Gaussian Process (RGP), and a stepwise regression [111, 113]. Arrieta et al. found that the regression tree yielded the best performance, while Gartziandia et al. found that regression tree, ensemble, and RGP were all valid. Finally, Ding et al. [112] used SVM with label propagation—a technique where both labeled and unlabeled training data are used, and the algorithm propagates labels to similar, unlabeled data. This can reduce the quantity of training data needed.

Metamorphic Relations: Ten approaches generate properties that describe output be-

havior [38]. Violations of such properties identify potential faults. Several publications build on the initial ideas of Kanewala et al. [127], whose approach (a) converts source code of functions into control-flow graphs, (b) selects code elements as features for a data set, and (c), trains a model that predicts whether a feature exhibits a particular metamorphic relation from a pre-compiled list. This requires training data where features are labeled with a binary classification based on whether or not they exhibit a particular relation. SVM and Decision Trees are used to train the model.

Kanewala et al. extended this work by adding a graph kernel [128]. Hardin et al. adapted this approach for label propagation [38]. Finally, Zhang et al. [132] experimented with the use of a Radial-Basis Function (RBF) NN. They extended the approach to a multi-label classification that can handle multiple metamorphic relations at once. Nair et al. [130] extended this work by demonstrating how data augmentation can enlarge the training dataset using mutants as the source of additional training data.

Korkmaz et al. [129] use a decision tree to predict the conditions on screen transitions in a GUI. Their model is trained using past system execution and potential guard conditions. Shu et al. [131] also propose the use of ML to assess security properties of protocol implementations. A protocol is specified using a state machine, and message confidentiality is assessed based on the reachability of a message. The L* algorithm is used to infer the model. A model is inferred, and then assessed for security violations. If a violation is found, input is produced that is then checked against the real implementation. If the violation is a false positive, it is used to refine to model.

Hiremath et al. [124, 125] present early work using ML to predict metamorphic relations for an ocean modeling application. The approach would pose a set of relations, evaluate whether they hold, and attempt to minimize a cost function based on the validity of the set of proposed relations. They do not specify a specific ML technique, but RL would be appropriate. Speiker and Gotlieb [126] also use RL to *select* metamorphic relations from a superset of potentially-applicable relations. Their approach, based on the contextual bandit algorithm evaluates whether a fault was discovered in an image classification algorithm after selecting relations. Ultimately, the agent learns which relations are applicable.

Type of Goal	Goal	# of Publications
Generate Input	Maximize Coverage	25
	Expose Performance Bottlenecks	6
	Show Conformance to (or Violation of) Specifications	5
	Generate Complex Input	5
	Improve Input or Output Diversity	4
	Predict Failing Input	2
Generate Oracle	Predict Output	15
	Predict Properties of Output	9
	Predict Test Verdict	8
Enhance Existing Method	Improve Effectiveness	11
	Improve Efficiency	7

Table 13: ML goals and the number of publications pursuing each goal.

4.3. Answering the Research Questions

We previously examined each broad testing practice where ML has been used to enhance test generation. In this section, we derive overall answers to our questions.

4.3.1. RQ2: Goals of Applying ML

Table 13 lists the identified goals of the authors in adopting ML in test generation, sorted by higher-level "category" and the number of times the objective is stated. We see three broad categories of goals. In the first two, a ML algorithm is used directly as part of test generation—to generate input or an oracle. As previously discussed, oracle generation results in three types of oracle. ML is used to directly predict output, to predict properties of output, or to predict a test verdict.

Regarding input generation, the most common goal is to use ML to increase coverage of some criterion associated with effective testing (53% of publications in this category). This measurement includes various forms of code coverage, states or transitions of a model, or pairwise input interactions. Other uses of ML include generating input that exposes performance bottlenecks, that demonstrates either conformance to—or violation of—behavioral specifications, or that increases input or output diversity. Others generate valid input for a complex data type or input predicted to fail.

In the final category, ML is used to tune the performance or effectiveness of an

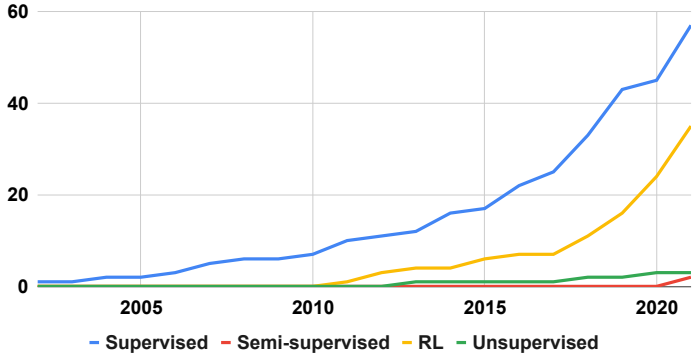


Figure 11: Growth in studies over time applying each type of ML approach.

existing test generation framework, often a GA. To improve efficiency, ML is used to cluster redundant tests, to replace an expensive fitness calculation with a prediction, to choose classes to target for generation, or to check input validity to avoid wasted time on invalid input. To improve effectiveness, ML is used to manipulate test cases (e.g., to replace method calls with those that will attain higher coverage of private code) or to tune the generation strategy (e.g., selecting fitness functions and mutation heuristics or the timeout for constraint solving).

RQ2 (Goal of ML): ML is integrated into the generation process to directly generate input (49%)—particularly to maximize some form of coverage—or oracles (33%)—particularly to predict the expected output. It is also used to improve the efficiency or effectiveness of existing test generation methods (19%).

4.3.2. RQ3: Integration of ML into Test Generation

The primary objective of RQ3 is to highlight where and how ML techniques have been integrated into the testing process. This includes the broad types of ML applied (i.e., supervised, unsupervised, semi-supervised, and reinforcement learning), the training data used by supervised techniques, and how ML was integrated (e.g., regression, classification, reward functions in RL). Figure 11 illustrates the growth in the number of publications applying each type of ML approach.

RQ3 (Integration of ML): The most common types of ML are supervised (59%) and reinforcement learning (36%). A small number of publications also employ unsupervised (4%) or semi-supervised (2%) learning.

The majority of the publications surveyed adopted a supervised approach, where a model is trained and used to make predictions. Supervised techniques were the first applied to either input or oracle generation, and remain the most common. Supervised techniques are—by far—the most common for oracle generation. They are also the most common for system and combinatorial interaction testing.

The predictions made by trained models are either from a pre-determined set of options (classification) or open (regression). Classification is often used in oracle generation, e.g., to produce a verdict (pass/fail) or to predict output for programs with a limited range of output options. Regression is more common in input generation, where a wide range of predictions can be made.

Both training time and quantity of training data need to be accounted for when considering a supervised technique. After being trained, a model is static and will not learn from new interactions, unlike when RL is used. A model must be retrained with a new set of training data if the user wishes to improve its accuracy. Therefore, it is important that supervised methods be supplied with sufficient quantity and quality of labeled training data. The incorporated supervised techniques generally learn from past system executions, labeled with a measurement of interest. If the label can be automatically recorded, then gathering sufficient data is often not a major concern. However, if SUT is computationally inefficient or information is not easily collectible (e.g., if a human must label data), it can be difficult to incorporate a supervised technique.

Adversarial learning may help to overcome a subset of training data challenges. This strategy forces models to compete, creating a feedback loop where performance is automatically improved without the need for additional human input. Two recent publications adopted adversarial networks, both in cases where input was associated with a numeric quality (performance of the SUT and speed of a vehicle). Neither case requires human labeling, so the models can be trained and automatically improved.

RQ3 (Integration of ML): Supervised learning is the most common ML for system testing, CIT, and all forms of oracle generation. Produced models perform regression (often for input generation) or classification (often for oracle generation). Quantity and quality of training data are concerns, especially when human input is required. Adversarial approaches can improve model accuracy.

Reinforcement learning is the second most common type of ML. Both supervised learning and RL have seen a sharp increase in use after 2017. RL was used more often than supervised learning in 2020 (eight publications versus two), and almost as often in 2021 (11 versus 12). RL has been used in all input generation problems, and it is the most common technique for GUI, unit, and performance input generation.

RL is often an appealing strategy because it does not require pre-training and can automatically improve its accuracy through interactions—as guided by feedback from a domain-specific reward function. RL is most applicable in situations where testing effectiveness can be judged using a numeric score, i.e., where a measurable assessment function already exists. This includes performance measurements—e.g., speed, memory or disk usage—or code coverage—as used in unit testing.

RL is effective when the SUT has branching or stateful qualities. This is the case in GUI testing, where a *sequence* of input steps are required to explore the interface. The selection of input depends on input already selected, and some GUI elements require several steps to access. Similarly, performance bottlenecks often emerge as the consequence of a sequence of actions, and coverage of code elements may require multiple input steps to put units in the correct state for input to interact with those elements.

Outside of individual test cases, RL is also highly effective stateful *test generation processes*, where test cases are manipulated after creation. Search-based test generation, often performed using genetic algorithms, is one such process—where test suites are evolved over a series of subsequent generations. RL can interact with this process, tuning aspects of the generation algorithm. Feedback from the reward function can be used to shape a new test suite or improve an existing one (e.g., by adjusting how the GA mutates test cases). Testing practices with numeric scoring functions, like those

mentioned above, often benefit from such a process. If a test suite attains high coverage or demonstrates performance bottlenecks, those qualities can often be improved further through slight manipulation of the test cases. RL can perform this generation task directly in some cases. However, in other cases, it can also be integrated into an existing test generation framework to manipulate how that framework functions. In the publications studied, RL is often used to manipulate the generation strategy to improve efficiency of effectiveness of test generation.

RQ3 (Integration of ML): RL is the most common ML for GUI, unit, and performance testing. It is effective for practices with numeric scoring functions and situations where testing requires a sequence of input steps. It is also effective at tuning existing generation tools.

Three publications applied unsupervised learning to cluster similar test cases to improve generation efficiency or to identify weakly tested areas of the SUT. Clustering has not been used often as part of generation, but is common in other areas of testing (e.g., to identify a subset of tests to execute [133]). It has potential for greater use in various filtering tasks during generation, especially with regard to improving efficiency.

RQ3 (Integration of ML): Unsupervised learning, e.g., clustering, is effective for filtering tasks such as discarding similar test cases or identifying weakly-tested parts of a SUT.

4.3.3. RQ4: ML Techniques Applied

RQ4 examines the specific ML techniques used by the authors. By answering this question, we expect to have a clearer view of how ML can support test generation. Table 14 lists all techniques employed, divided by ML type and sorted by the number of publications where each technique is employed.

Neural networks are the most common techniques used as part of supervised learning. In particular, Backpropagation NNs are used most (12% of publications). Support vector machines are also employed often, as are various forms of decision trees.

Type	Family	Technique	Publications
Supervised	**Neural Networks**	Backpropagation NN	12
		Multi-Layer Perceptron	5
		Artificial NN, Long Short-Term Memory (LSTM) NN	4
		Deep NN, Radial-Basis Function NN	2
		Backpropagation NN + Cascade, Convolutional NN, Feedforward ANN, MLP + LSTM , Probabilistic NN, Recurrent NN	1
	Trees	Decision Tree	5
		Gradient Boosting, Random Forest	2
		Ada-Boosted Tree, C4.5, Regression Tree, Tree-LSTM	1
	Others	Support Vector Machine	9
		L*, Conditional Random Fields, K-Nearest Neighbors	2
		Adaptive Boosting, Ensemble, Gaussian Process, Multivariate Time Series, Parallel Distributed Processing, Query Strategy Framework, Regression Gaussian Process, RIPPER, Stepwise Regression	1
RL	**Q-Learning**	Q-Learning	14
		Deep Q-Network, Double Q-Learning	2
		Delayed Q-Learning, Dueling Deep Q-Network, Double Deep Q-Network, Q-Learning + LSTM, ReLU Q-Learning	1
	Others	Differential Semi-Gradient Sarsa (DSG-Sarsa), Upper Confidence Bound (UCB)	2
		Contextual Bandit, Deep RL, Monte Carlo Control, Monte Carlo Tree Search, Sarsa, SOFTMAX	1
Semi-supervised		CNN, Generative Adversarial Network (GAN), Conditional GAN	1
Unsupervised		Backpropagation NN, Expectation-Maximization, MeanShift	1

Table 14: ML techniques adopted—divided by ML type and family of ML techniques—ordered by number of publications where the technique is adopted.

Backpropagation NNs are a classic technique where a network is composed of multiple layers. In each layer, a weight value for each node is calculated based on available information. In such networks, information is feed *forward*—there are no cyclic connections back to earlier layers. However, the backpropagation feature propagates error backwards, allowing earlier nodes to adjust weights if necessary. This leads to less complexity and faster learning rates. In recent years, more complex neural networks have continued to implement backpropagation as one (of many) features.

Recently, NNs utilizing Long Short-Term Memory (LSTM) have also become quite common. Unlike in traditional feedforward NNs, LSTM has feedback connections. This creates loops in the network, allowing information to persist. This adaptation al-

lows such networks to process not just single data points, but sequences of information where one data point depends on earlier data points. LSTM networks and deep neural networks are likely to become more common in the near future.

Reinforcement learning techniques are dominated by forms of Q-Learning—variants are used in 22% of all sampled publications. Q-Learning is a prototypical form of off-policy reinforcement learning, meaning that it can choose either to take an action guided by the current "best" policy—maximizing the expected reward—or it can choose to take a random action in the hopes of learning a better policy or refining the expectations of the current policy. Many of the other RL techniques are also off-policy techniques, and follow a similar process to Q-Learning, with various differences (e.g., calculating reward or action decisions in a different manner).

RQ4 (ML Techniques): Neural networks, especially Backpropagation NNs, are the most common supervised learning techniques. Reinforcement learning techniques are generally based on Q-Learning.

Some authors have chosen algorithms because they worked well in previous work (e.g., [72, 128]). Others saw these algorithms work well on similar problems outside of test generation (e.g., [96]), or chose algorithms thought to represent the state-of-the-art at that time for a class of problem (e.g., [51]). However, most authors do not justify their choice of algorithm, nor do they often compare alternative choices.

In older publications, the majority of authors either implemented their own ML algorithms or adapted unspecified implementations. However, in recent years, mature open-source ML frameworks have emerged. These frameworks can accelerate the pace and effectiveness of research by making robust ML algorithms available. Some of the ML frameworks used in the sampled publications include keras-rl [13], OpenAI Gym [43, 13, 126], PyTorch [126], scikit-learn [38], Theano [91], and WEKA [94, 121]. The use of a framework constrains the choice of algorithm. However, all of these frameworks offer a variety of options, and may even allow researchers to compare results across multiple algorithms.

Type	Metric	Publications
Supervised	Faults Detected (Inc. Mutants, Performance Issues)	25
	Accuracy (Inc. Correct Classifications, ROC)	24
	Coverage (Code, State, etc.)	7
	Efficiency (Inc. Scalability, # Tests Gen or Executed, Time)	7
	Test Size (Case, Suite, Covering Array)	4
	Training Data Size	2
	Input/Output Diversity, Input Validity, **Model Size**	1
RL	Coverage (Code, State, etc.)	20
	Faults Detected (Inc. Mutants, Performance Issues)	10
	Efficiency (Inc. Scalability, # Tests Gen or Executed, Time)	4
	Input/Output Diversity	2
	# Exceptions, Qualitative Analysis, Queries Solved, Requirements Met, Test Case Size	1
Semi-supervised	Faults Detected (Inc. Mutants, Performance Issues)	2
	Accuracy, **Labeling and Training Effort**	1
Unsupervised	**# Clusters**, Qualitative Analysis	1

Table 15: Evaluation metrics adopted (similar metrics are grouped), divided by ML approach and ordered by number of publications using each metric. Metrics in bold are related to ML.

RQ4 (ML Techniques): The choice of algorithm is often not explained by authors, but may be inspired by insights from previous or related work, an algorithm having performed well on a similar problem, or algorithms available in open-source frameworks (e.g., OpenAI Gym, PyTorch, or WEKA).

4.3.4. RQ5: Evaluation of the Test Generation Framework

The goal of RQ5 is to understand how authors have evaluated their work—in particular, how ML affects the evaluation. The metrics adopted by the authors are listed in Table 15, divided by ML approach and ranked by the number of publications. We group similar metrics (e.g., coverage metrics, notions of fault detection, etc.).

Almost all of these are standard evaluation metrics for test generation. Some are

specific to a testing practice (e.g., covering array size) or aspect of test generation (e.g., number of queries solved), while others can be applied across testing practices (e.g., fault detection). Naturally—whether ML is incorporated or not—a generation framework must be evaluated on its effectiveness at generating tests.

However, many authors separately evaluate the impact of ML. Supervised learning approaches were often evaluated using some notion of accuracy of the trained model—using various accuracy measurements, correct classification rate, and ROC. Supervised approaches have also been evaluated on training data and model size. Semi-supervised approaches were also evaluated using accuracy and labeling and training effort. One unsupervised approach was evaluated on number of clusters produced.

Reinforcement learning approaches were not evaluated using ML-specific metrics. However, this is reasonable, as RL is generally used to learn the policy that maximizes a numeric reward function. The reward is based on the goals of the overall test generation framework. Rather than evaluating using an absolute notion of accuracy, the success of the ML component can be seen in improved reward measurements, attainment of a checklist of goals, or metrics such as fault detection.

RQ5 (Evaluation): ML-enhanced generation is still evaluated by traditional metrics (e.g., fault detection). (Semi-/Un-)Supervised approaches are also evaluated using ML metrics (accuracy, training data/model size, labeling/training effort, # of clusters). RL is evaluated using testing metrics tied to the reward.

4.3.5. RQ6: Limitations and Open Challenges

The sampled publications show great potential in improving automated test generation. However, we have observed multiple challenges that must be overcome to transition research into use in real-world software development.

Volume, Contents, and Collection of Training Data: (Semi-)Supervised ML approaches require training data to create a predictive model. There are multiple challenges related to the *required volume* of training data, the *required contents* of the training data, and *human effort* required to produce that training data.

Regardless of the specific testing practice addressed, the volume of training data that is needed can be vast. This data is generally attained from labeled system execution logs, which means that the SUT needs to be executed *many* times to gather the information needed to train the model. Approaches based on deep learning could produce highly accurate models, but may require thousands of executions to gather the required training data. Some approaches also must preprocess the collected data before training. While it may be possible to automatically gather training data, the time required to produce the training data can still be high and must be considered.

This is particularly true for cases where a regression is performed rather than a classification—e.g., an expected value oracle or complex test input. Producing a complex continuous value is a more difficult task than a simple classification, and requires significant training data—with a range of outcomes—to make accurate classifications.

In addition, the contents of the training data must be considered. If generating test input, the training data must consist of a wide range of input scenarios with diverse outcomes that reflect the specific problem of interest and its different branching possibilities. Consider code coverage. If one wishes to predict the input that will cover a particular element, then the training data must contain sufficient information content to describe how to cover that element. That requires a diverse training set.

Models based on output behavior—e.g., expected value oracles or models that predict input based on particular output values—suffer from a related issue. The training data for expected value oracles must either come from passing test cases—i.e., the output must be correct—or labels must be applied by humans. A small number of cases accidentally based on failing output may be acceptable if the algorithm is resilient to noise in the training data, but training on faulty code can result in an inaccurate model. This introduces a significant barrier to automating training by, e.g., generating test input and simply recording the output that results.

Models that make predictions based on failures—e.g., test verdict oracles or models that produce input predicted to trigger a failure or performance issue—require training data that contains a large number of *failing test cases*. This implies that faults have already been discovered in the system and, presumably, fixed before the model is trained. This introduces a paradox. There may be remaining failures to discover. However, the

more training data that is needed, the less the need for—or impact of—the model.

In some cases, training data must be labelled (or even collected) by a human. Again, oracles suffer heavily from this problem. Test verdict oracles require training data where each entry is assigned a verdict. This requires either existing test oracles—reducing the need for a ML-based oracle in the first place—or human labeling of test results. Judging test results is time-consuming and can be erroneous as testers become fatigued [134, 7], making it difficult to produce a significant volume of training data. Metamorphic relation oracles face a similar dilemma, where training data must be labeled based on whether a particular metamorphic relation holds.

For some problems, these issues can be avoided by employing RL instead. RL will learn while interacting with the SUT. In cases where the effectiveness of ML can be measured automatically—e.g., code coverage, performance bottlenecks—RL is a viable solution. However, cases where a ground truth is required—e.g., oracles—are not as amenable to RL. RL also requires many executions of the SUT, which can be an issue if the SUT is computationally expensive or otherwise difficult to execute and monitor, such as when specialized hardware is required for execution.

Otherwise, techniques are required that (1) can enhance training data, (2) that can extrapolate from limited training data, and (3), that can tolerate noise in the training data. Means of generating synthetic training data, like in the work of Nair et al. [130], demonstrate the potential for data augmentation to help in overcoming this limitation. In addition, adversarial learning approaches offer a way to automatically improve the accuracy of a model—reducing the need for a large training dataset. Again, however, such approaches are of limited use in cases where human involvement is required.

RQ6 (Challenges): Supervised learning is limited by the required quantity, quality, and contents of training data—especially when human effort is required. Oracles particularly suffer from these issues. RL and adversarial learning are viable alternatives when data collection and labelling can be automated.

Retraining and Feedback: After training, models generated by supervised learning techniques have a fixed error rate and do not learn from new mistakes made. If

the training data is insufficient or inaccurate, the generated model will be inaccurate. The ability to improve the model based on additional feedback could help account for limitations in the initial training data.

There are two primary means to overcome this limitation—either retraining the model using an enriched training dataset, or adopting a reinforcement learning approach that can adapt its expectations based on feedback. Both means carry challenges. Retraining requires (a) establishing a schedule for when to train the updated model, and (b), an active effort on the part of human testers to enrich and curate the training dataset. Adversarial learning offers an automated means to retrain the model. However, there are still limitations in when adversarial learning can be applied.

Enriching the dataset—as well as the use of RL—requires some kind of feedback mechanism to judge the effectiveness of the predictions made. This can be difficult in some cases, such as test oracles, where human feedback may be required. Human feedback, even on a subset of the decisions made, reduces the cost savings of automation.

RQ6 (Challenges): Models should be retrained over time. How often retraining occurs depends, partially, on the cost to gather and label additional data or on the amount of human feedback required.

Complexity of Studied Systems: Regardless of type of ML, many of the proposed approaches are evaluated on highly simplistic systems, with only a few lines of code or possible function outcomes. While it is intuitive to *start* with simplistic examples to examine the viability of an ML approach, application in the field require accurate predictions for highly complex functions and systems with many branching code paths. If a function is simple, there is likely little need for a predictive model in the first place. Several recent studies feature more thorough evaluations (e.g., [60, 32, 78]), even on industrial systems (e.g., [53, 96]). However, it largely remains to be seen whether the proposed techniques can be used on real-world real-world production code.

Generation of models for arbitrary systems with unconstrained output may be prohibitively difficulty for even the most effective ML techniques. This is particularly the case for expected value oracles, which predict specific outputs. In such cases,

some abstraction should be expected. One possibility to consider is a variable level of abstraction—e.g., a training-time decision to cluster the output into an adjustable number of representative values (e.g., the centroid of each cluster). Training could take place over different settings for this parameter, and an acceptable balance between quality and level-of-detail could be explored.

In any evaluation of a test generation technique, a variety of systems should be considered, not just one or two. The complexity of the systems should vary. This enables the assessment of the scalability of the proposed techniques. Researchers should clearly examine how prediction accuracy, training data requirements (for supervised learning), and time to convergence on an optimal policy (for RL) scale as the complexity of the system increases. This would enable a better understanding of the limitations and applicability of ML-based techniques.

RQ6 (Challenges): Scalability of many ML techniques to real-world systems is not clear. When modeling complex functions, varying degrees of abstraction could be explored if techniques are unable to scale. In all evaluations, a range of systems should be considered, and explicit analysis of scalability (e.g., of accuracy, training, learning rate) should be performed.

Variety, Complexity, and Tuning of ML Techniques: Authors of the sampled publications rarely explain or justify their choice of ML algorithm—often stating that an algorithm worked well previously or that it is "state-of-the-art", if any rationale is offered. It is even rarer that multiple algorithms are compared to determine which is best for a particular task. As the purpose of many research studies is to demonstrate the viability of an idea, the choice of algorithm is not always critically important. However, this choice still has implications, as it may give a false impression of the applicability of an approach and unnecessarily introduce a performance ceiling [135] that could be overcome through consideration of alternative techniques.

One reason for this limitation may be that testing researchers are generally ML *users*, not ML experts. They may lack the domain expertise to know which algorithms to apply. Collaboration with ML researchers may help overcome this challenge. The

use of open-source ML frameworks can also ease this challenge by removing the need for researchers to develop their own algorithms. Rather than needing to understand each algorithm, they could instead compare the performance of multiple available alternatives. This comparison would also lead to a richer evaluation and discussion.

Many of the proposed approaches—especially earlier ones—are based on simple neural networks with few layers. These techniques have strict limitations in the complexity of the data they can model and have been replaced by more sophisticated techniques. Deep learning, which may utilize many hidden layers, may be essential in making accurate predictions for complex systems. Few approaches to date have utilized deep learning, but such approaches are starting to appear, and we would expect more to explore these techniques in the coming years. However, deep learning also introduces steep requirements on the training data that may limit its applicability [136].

Almost all of the proposed approaches utilize a single ML technique. An approach explored in many domains is the use of *ensembles* [137]. In such approaches, models are trained on the same data using a variety of techniques. Each model is asked for a prediction, and then the final prediction is based on the consensus of the ensemble. Ensembles are often able to reach stable, accurate conclusions in situations where a single model may be inaccurate. A small number of studies have applied ensembles [48, 88, 101, 111, 113], but such techniques are rare. Ensembles may be a way to overcome the fragility of some ML approaches.

Many ML techniques have a number of parameters that can be tuned (e.g., the learning rate, number of hidden units, or activation function) [138]. Parameter tuning can significantly impact prediction accuracy and enable significant improvements in the results of even simple ML techniques. The sampled publications do not explore the impact of such tuning. This is an oversight that should be corrected in future work.

RQ6 (Challenges): Researchers rarely justify the choice of ML technique or compare alternatives. The use of open-source ML frameworks can ease comparison. Deep learning and ensemble techniques, as well as hyperparameter tuning, should also be explored.

Lack of Standard Benchmarks: Research benchmarks (e.g., [139, 140]) have enabled sophisticated analyses and comparison of approaches for automated test generation. Such benchmarks usually contain a set of systems prepared for a particular type of research evaluation. Bug benchmarks, in particular, typically contain real faults curated from a variety of systems, along with metadata on those faults. Such benchmarks ease comparing or replicating research, remove bias from system selection, and ensure the real-world effectiveness of techniques.

System or bug benchmarks have been used in a small number of sampled publications (e.g., Defects4J for unit testing [32, 85], F-Droid for Android testing [69, 70]). However, the majority of studies do not use benchmarks. Some studies require their own particular evaluation. However, in cases where evaluation is over-simplistic, or where code or metadata is unavailable, this makes comparison and replication difficult.

Benchmarks are typically tied to particular system types or testing practices. In cases where benchmarks exist—unit, web app, mobile app, and performance testing in particular—we would encourage researchers to use these benchmarks. In other cases, the creation of benchmarks specifically for ML-enhanced test generation research could advance the state-of-the-art in the field, spur new research advances, and enable replication and extension of proposed approaches.

In particular, we recommend the creation of such a benchmark for oracle generation. Such a benchmark should contain a variety of code examples from multiple domains and of varying levels of complexity. Code examples should be paired with the metadata needed to support oracle generation. This would include sample test cases and human-created test oracles, at minimum. Such a benchmark could also include sample training data that could be augmented over time by researchers.

Lack of Replication Package or Open Code: A common dilemma is lack of access to research code and data. Often, the publication itself is not sufficient to allow replication or application of the technique in a new context. This applies to research in ML-enhanced test generation as well, as few authors released code or data. Some publications make use of open-source ML frameworks. This is positive, in that the tools are trustworthy and available. However, there still may not be enough information to enable replication without the authors' own code and data. Further, these frameworks

evolve over time, and the results may differ because the underlying ML technique has changed since the original study was published.

Researchers should include a replication package with their source code, execution scripts, and the versions of external dependencies used when the study was performed. This package should also include training data and the gathered experiment observations used by the authors in their analyses.

RQ6 (Challenges): Research is limited by overuse of simplistic examples, the lack of common benchmarks, and the unavailability of code and data. Researchers should be encouraged to use available benchmarks, and provide replication packages and open code. New benchmarks could be created for ML challenges (e.g., oracle generation).

5. Threats to Validity

External and Internal Validity: Our conclusions are based on the publications sampled. It is possible that we may have omitted important publications or sampled an inadequate number of publications. This can affect internal validity—the evidence we use to make conclusions—and external validity—the generalizability of our findings. SLRs are not required to reflect all publications from a research field. Rather, their selection protocol (search string, inclusion and exclusion criteria) should be sufficient to ensure an adequate sample of the field. We believe that our selection strategy was appropriate. We tested different search strings, and performed a validation exercise to test the robustness of our string. We have used four databases, covering the majority of relevant software engineering venues. Our final set of publications includes 97 primary publications, which we believe is sufficient to make informed conclusions.

Conclusion Validity: The analyses performed are qualitative, and require inference from the authors. This could introduce bias into our conclusions. For example, subjective judgements are required as part of article selection, data extraction, and coding (e.g., categorizing publications based on the oracle type or testing problem). To control

for bias, protocols were discussed and agreed upon by both authors, and independent verification took place on—at least—a sample of all decisions made by either author.

Construct Validity: We used a set of properties to guide data extraction. These properties may have been incomplete or misleading. However, we have tried to establish properties that were appropriate and directly informed by our research questions. These properties were iteratively refined using a selection of papers, and we believe they have allowed us to thoroughly answer the research questions.

6. Conclusions

Automated test generation is a well-studied research topic, but there are critical limitations to overcome. Recently, researchers have begun to use ML to enhance automated test generation. We have characterized emerging research on on this topic through a systematic literature review examining testing practices that have been addressed, the goals of using ML, how ML is integrated into the generation process, which specific ML techniques are applied, how the full test generation process is evaluated, and open research challenges.

Based on 97 publications, we observed that ML supports generation of input and oracles for a variety of testing practices (e.g., system or GUI testing) and oracle types (e.g., verdicts and expected values). During input generation, ML either directly generates input or improves the efficiency or effectiveness of existing generation methods.

The most common types of ML are supervised (59%) and RL (36%). A small number of publications also employ unsupervised (4%) or semi-supervised (2%) learning. Supervised learning is the most common ML for system testing, Combinatorial Interaction Testing, and all forms of oracle generation. Neural networks are the most common supervised techniques, and techniques are evaluated using both traditional testing metrics (e.g., coverage) and ML-related metrics (e.g., accuracy).

RL is the most common ML for GUI, unit, and performance testing. It is effective for practices with scoring functions and when testing requires a sequence of input steps. It is also effective at tuning generation tools. Reinforcement learning techniques are generally based on Q-Learning, and approaches are evaluated using testing met-

rics (often tied to the reward function). Finally, unsupervised learning is effective for filtering tasks such as discarding similar test cases.

The publications show great promise, but there are significant open challenges. Learning is limited by the required quantity, quality, and contents of training data. Models should be retrained over time. Whether techniques will scale to real-world systems is not clear. Researchers rarely justify the choice of ML technique or compare alternatives. Research is limited by the overuse of simplistic examples, the lack of standard benchmarks, and the unavailability of code and data. Researchers should be encouraged to use available benchmarks and provide replication packages and open code. New benchmarks could be created for ML challenges (e.g., oracle generation). We hope that our findings will serve as a roadmap for both researchers and practitioners interested in the use of ML as part of test generation.

7. Acknowledgments

This research was supported by Vetenskapsrådet grant 2019-05275.

References

[1] K. Naik, P. Tripathy, Software testing and quality assurance: theory and practice, John Wiley & Sons, 2011.

[2] G. Hamidi, Reinforcement learning assisted load test generation for e-commerce applications, arXiv preprint arXiv:2007.12094 (2020).

[3] M. Pezze, M. Young, Software Test and Analysis: Process, Principles, and Techniques, John Wiley and Sons, 2006.

[4] G. Gay, M. Staats, M. Whalen, M. Heimdahl, The risks of coverage-directed test case generation, Software Engineering, IEEE Transactions on PP (2015). doi:10.1109/TSE.2015.2421011.

[5] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, J. Benefelds, An industrial evaluation of unit test generation: Finding real faults in a financial application, in: Proceedings of the 39th IEEE/ACM International Conference on

Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP), ICSE 2017, ACM, New York, NY, USA, 2017.

[6] A. Orso, G. Rothermel, Software testing: A research travelogue (2000–2014), in: Proceedings of the on Future of Software Engineering, FOSE 2014, ACM, New York, NY, USA, 2014, pp. 117–132. URL: http://doi.acm.org/10.1145/2593882.2593885. doi:10.1145/2593882.2593885.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: A survey, IEEE transactions on software engineering 41 (2014) 507–525.

[8] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (2013) 1978–2001.

[9] H. Almulla, G. Gay, Learning how to search: Generating exception-triggering tests through adaptive fitness function selection, in: 13th IEEE International Conference on Software Testing, Validation and Verification, 2020.

[10] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, T. Xie, An empirical study of android test generation tools in industrial cases, in: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2018, pp. 738–748.

[11] G. Guizzo, F. Sarro, J. Krinke, S. R. Vergilio, Sentinel: A hyper-heuristic for the generation of mutant reduction strategies, IEEE Transactions on Software Engineering (2020).

[12] V. Dunjko, H. J. Briegel, Machine learning & artificial intelligence in the quantum domain: a review of recent progress, Reports on Progress in Physics 81 (2018) 074001.

[13] J. Kim, M. Kwon, S. Yoo, Generating test input with deep reinforcement learning, in: Proceedings of the 11th International Workshop on Search-Based Soft-

ware Testing, SBST '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 51–58. URL: https://doi.org/10.1145/3194718.3194720. doi:10.1145/3194718.3194720.

[14] T. McDermott, D. DeLaurentis, P. Beling, M. Blackburn, M. Bone, Ai4se and se4ai: A research roadmap, INSIGHT 23 (2020) 8–14. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/inst.12278. doi:https://doi.org/10.1002/inst.12278. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/inst.12278.

[15] A. Fontes, G. Gay, Using machine learning to generate test oracles: A systematic literature review, in: Proceedings of the 1st International Workshop on Test Oracles, TORACLE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 1–10. URL: https://doi.org/10.1145/3472675.3473974. doi:10.1145/3472675.3473974.

[16] E. Alpaydin, Introduction to machine learning, MIT press, 2020.

[17] M. Abadi, M. Isard, D. G. Murray, A computational model for tensorflow: an introduction, in: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2017, pp. 1–7.

[18] N. Ketkar, Introduction to keras, in: Deep learning with Python, Springer, 2017, pp. 97–111.

[19] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, arXiv preprint arXiv:1606.01540 (2016).

[20] R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[21] I. Goodfellow, Y. Bengio, A. Courville, Y. Bengio, Deep learning, volume 1, MIT press Cambridge, 2016.

[22] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, M. P. Guimaraes, Machine learning applied to software testing: A systematic mapping study, IEEE Transactions on Reliability 68 (2019) 1189–1212.

[23] N. Jha, R. Popli, Artificial intelligence for software testing-perspectives and practices, in: 2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT), 2021, pp. 377–382. doi:10.1109/CCICT53244.2021.00075.

[24] C. Ioannides, K. I. Eder, Coverage-directed test generation automated by machine learning – a review, ACM Trans. Des. Autom. Electron. Syst. 17 (2012). URL: https://doi.org/10.1145/2071356.2071363. doi:10.1145/2071356.2071363.

[25] J. M. Balera, V. A. de Santiago Júnior, A systematic mapping addressing hyper-heuristics within search-based software testing, Information and Software Technology 114 (2019) 176–189.

[26] K. Petersen, S. Vakkalanka, L. Kuzniarz, Guidelines for conducting systematic mapping studies in software engineering: An update, Information and Software Technology 64 (2015) 1–18.

[27] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, 2007.

[28] K. Sen, G. Agha, Cute and jcute: Concolic unit testing and explicit path model-checking tools, in: In CAV, Springer, 2006, pp. 419–423.

[29] P. McMinn, Search-based software test data generation: A survey, Software Testing, Verification and Reliability 14 (2004) 105–156.

[30] G. Gay, M. Staats, M. Whalen, M. Heimdahl, Automated oracle data selection support, Software Engineering, IEEE Transactions on PP (2015) 1–1. doi:10.1109/TSE.2015.2436920.

[31] A. Salahirad, H. Almulla, G. Gay, Choosing the fitness function for the job: Automated generation of test suites that detect real faults, Software Testing, Verification and Reliability 29 (2019) e1701. URL: https://onlinelibrary.wiley.com/

doi/abs/10.1002/stvr.1701. doi:10.1002/stvr.1701. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1701, e1701 stvr.1701.

[32] H. Almulla, G. Gay, Learning how to search: Generating exception-triggering tests through adaptive fitness function selection, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 63–73. doi:10.1109/ICST46399.2020.00017.

[33] I. A. Qureshi, A. Nadeem, Gui testing techniques: a survey, International Journal of Future computer and communication 2 (2013) 142.

[34] P. Runeson, A survey of unit testing practices, IEEE Software 23 (2006) 22–29. doi:10.1109/MS.2006.91.

[35] M. H. Moghadam, Machine learning-assisted performance testing, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 1187–1189. URL: https://doi.org/10.1145/3338906.3342484. doi:10.1145/3338906.3342484.

[36] C. Nie, H. Leung, A survey of combinatorial testing, ACM Comput. Surv. 43 (2011). URL: https://doi.org/10.1145/1883612.1883618. doi:10.1145/1883612.1883618.

[37] H. Ebadi, M. H. Moghadam, M. Borg, G. Gay, A. Fontes, K. Socha, Efficient and effective generation of test cases for pedestrian detection - search-based software testing of baidu apollo in svl, in: 2021 IEEE International Conference on Artificial Intelligence Testing (AITest), 2021, pp. 103–110. doi:10.1109/AITEST52744.2021.00030.

[38] B. Hardin, U. Kanewala, Using semi-supervised learning for predicting metamorphic relations, in: Proceedings of the 3rd International Workshop on Metamorphic Testing, MET '18, Association for Computing Machinery, New

York, NY, USA, 2018, p. 14–17. URL: https://doi.org/10.1145/3193977.3193985. doi:10.1145/3193977.3193985.

[39] D. Araiza-Illan, A. G. Pipe, K. Eder, Intelligent agent-based stimulation for testing robotic software in human-robot interactions, in: Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering, MORSE '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 9–16. URL: https://doi.org/10.1145/3022099.3022101. doi:10.1145/3022099.3022101.

[40] D. Baumann, R. Pfeffer, E. Sax, Automatic generation of critical test cases for the development of highly automated driving functions, in: 2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring), 2021, pp. 1–5. doi:10.1109/VTC2021-Spring51267.2021.9448686.

[41] M. Buzdalov, A. Buzdalova, Adaptive selection of helper-objectives for test case generation, in: 2013 IEEE Congress on Evolutionary Computation, 2013, pp. 2245–2250. doi:10.1109/CEC.2013.6557836.

[42] M. Esnaashari, A. H. Damia, Automation of software test data generation using genetic algorithm and reinforcement learning, Expert Systems with Applications 183 (2021) 115446. URL: https://www.sciencedirect.com/science/article/pii/S0957417421008605. doi:https://doi.org/10.1016/j.eswa.2021.115446.

[43] S. Huurman, X. Bai, T. Hirtz, Generating api test data using deep reinforcement learning, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20, Association for Computing Machinery, New York, NY, USA, 2020, p. 541–544. URL: https://doi.org/10.1145/3387940.3392214. doi:10.1145/3387940.3392214.

[44] S. Reddy, C. Lemieux, R. Padhye, K. Sen, Quickly generating diverse valid test inputs with reinforcement learning, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE

'20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1410–1421. URL: https://doi.org/10.1145/3377811.3380399. doi:10.1145/3377811.3380399.

[45] Y. Deng, G. Lou, X. Zheng, T. Zhang, M. Kim, H. Liu, C. Wang, T. Y. Chen, Bmt: Behavior driven development-based metamorphic testing for autonomous driving models, in: 2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET), 2021, pp. 32–36. doi:10.1109/MET52542.2021.00012.

[46] F. Bergadano, Test case generation by means of learning techniques, SIGSOFT Softw. Eng. Notes 18 (1993) 149–162. URL: https://doi.org/10.1145/167049.167074. doi:10.1145/167049.167074.

[47] C. Budnik, M. Gario, G. Markov, Z. Wang, Guided test case generation through ai enabled output space exploration, in: Proceedings of the 13th International Workshop on Automation of Software Test, AST '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 53–56. URL: https://doi.org/10.1145/3194733.3194740. doi:10.1145/3194733.3194740.

[48] R. Eidenbenz, C. Franke, T. Sivanthi, S. Schoenborn, Boosting exploratory testing of industrial automation systems with ai, 2021, pp. 362–371. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85107943033&doi=10.1109%2fICST49551.2021.00048&partnerID=40&md5=3290a5a5dd10e18af82b047a12502bce. doi:10.1109/ICST49551.2021.00048, cited By 0.

[49] Z. Gao, W. Dong, R. Chang, C. Ai, The stacked seq2seq-attention model for protocol fuzzing, in: 2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT), 2019, pp. 126–130. doi:10.1109/ICCSNT47585.2019.8962499.

[50] K. Kikuma, T. Yamada, K. Sato, K. Ueda, Preparation method in automated

test case generation using machine learning, in: Proceedings of the Tenth International Symposium on Information and Communication Technology, SoICT 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 393–398. URL: https://doi.org/10.1145/3368926.3369679. doi:10.1145/3368926.3369679.

[51] M. Kıraç, B. Aktemur, H. Sözer, C. Gebizli, Automatically learning usage behavior and generating event sequences for black-box testing of reactive systems, Software Quality Journal 27 (2019) 861–883. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85059878995&doi=10.1007%2fs11219-018-9439-1&partnerID=40&md5=291bd6c82252e9f7c1b0f230e23ec10c. doi:10.1007/s11219-018-9439-1, cited By 0.

[52] K. Meinke, H. Khosrowjerdi, Use case testing: A constrained active machine learning approach, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 12740 LNCS (2021) 3–21. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85111470675&doi=10.1007%2f978-3-030-79379-1_1&partnerID=40&md5=1417ef71a9d511de0e433ada57e32cbd. doi:10.1007/978-3-030-79379-1_1, cited By 0.

[53] A. G. Mirabella, A. Martin-Lopez, S. Segura, L. Valencia-Cabrera, A. Ruiz-Cortés, Deep learning-based prediction of test input validity for restful apis, in: 2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest), 2021, pp. 9–16. doi:10.1109/DeepTest52559.2021.00008.

[54] P. Papadopoulos, N. Walkinshaw, Black-box test generation from inferred models, in: Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE '15, IEEE Press, 2015, p. 19–24.

[55] S. L. Shrestha, Automatic generation of simulink models to find bugs in a cyber-physical system tool chain using deep learning, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, ICSE '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 110–112. URL: https://doi.org/10.1145/3377812.3382163. doi:10.1145/3377812.3382163.

[56] K. Ueda, H. Tsukada, Accuracy improvement by training data selection in automatic test cases generation method, 2021, pp. 438–442. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85106413235&doi=10.1109%2fICIET51873.2021.9419636&partnerID=40&md5=546d89ab1c6873a299a9f2297417bed3. doi:10.1109/ICIET51873.2021.9419636, cited By 0.

[57] M. Utting, B. Legeard, F. Dadeau, F. Tamagnan, F. Bouquet, Identifying and generating missing tests using machine learning on execution traces, in: 2020 IEEE International Conference On Artificial Intelligence Testing (AITest), 2020, pp. 83–90. doi:10.1109/AITEST49225.2020.00020.

[58] R. Zhao, S. Lv, Neural-network based test cases generation using genetic algorithm, in: 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007), 2007, pp. 97–100. doi:10.1109/PRDC.2007.63.

[59] J. Zhu, L. Wang, Y. Gu, X. Lin, Learning to restrict test range for compiler test, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2019, pp. 272–274. doi:10.1109/ICSTW.2019.00064.

[60] Z. Chen, Z. Chen, Z. Shuai, G. Zhang, W. Pan, Y. Zhang, J. Wang, Synthesize solving strategy for symbolic execution, 2021, pp. 348–360. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85111422819&doi=10.1145%2f3460319.3464815&partnerID=40&md5=e4d913f59fa100cc794de8989a1e2e60. doi:10.1145/3460319.3464815, cited By 0.

[61] C. Paduraru, M. Paduraru, A. Stefanescu, Riverfuzzrl - an open-source tool to experiment with reinforcement learning for fuzzing, 2021, pp. 430–435. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85107918441&doi=10.1109%2fICST49551.2021.00055&partnerID=40&md5=9f99d05a66b7393caae98aa3ea6a0bd7. doi:10.1109/ICST49551.2021.00055, cited By 0.

[62] S. Luo, H. Xu, Y. Bi, X. Wang, Y. Zhou, Boosting symbolic execution via constraint solving time prediction (experience paper), in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 336–347. URL: https://doi.org/10.1145/3460319.3464813. doi:10.1145/3460319.3464813.

[63] N. Majma, S. M. Babamir, Software test case generation test oracle design using neural network, in: 2014 22nd Iranian Conference on Electrical Engineering (ICEE), 2014, pp. 1168–1173. doi:10.1109/IranianCEE.2014.6999712.

[64] K. Mishra, S. Tiwari, A. Misra, Combining non revisiting genetic algorithm and neural network to generate test cases for white box testing, Advances in Intelligent and Soft Computing 124 (2011) 373–380. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84855228800&doi=10.1007%2f978-3-642-25658-5_46&partnerID=40&md5=0f21d2455ee273d6f3954f310f7d02a3. doi:10.1007/978-3-642-25658-5_46, cited By 1.

[65] D. Adamo, M. K. Khan, S. Koppula, R. Bryce, Reinforcement learning for android gui testing, in: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 2–8. URL: https://doi.org/10.1145/3278186.3278187. doi:10.1145/3278186.3278187.

[66] S. Ariyurek, A. Betin-Can, E. Surer, Automated video game testing using synthetic and humanlike agents, IEEE Transactions on Games 13 (2021) 50–67. doi:10.1109/TG.2019.2947597.

[67] M. Brunetto, G. Denaro, L. Mariani, M. Pezzè, On introducing automatic test case generation in practice: A success story and lessons learned, Journal of Systems and Software 176 (2021) 110933. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000303. doi:https://doi.org/10.1016/j.jss.2021.110933.

[68] W. Choi, G. Necula, K. Sen, Guided gui testing of android apps with minimal restart and approximate learning, ACM SIGPLAN Notices 48 (2013) 623–639. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84888802373&doi=10.1145%2f2544173.2509552&partnerID=40&md5=476dda06dd32a8e6289ecf3a83dc01b1. doi:10.1145/2544173.2509552, cited By 95.

[69] E. Collins, A. Neto, A. Vincenzi, J. Maldonado, Deep reinforcement learning based android application gui testing, in: Brazilian Symposium on Software Engineering, SBES '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 186–194. URL: https://doi.org/10.1145/3474624.3474634. doi:10.1145/3474624.3474634.

[70] C. Degott, N. P. Borges Jr., A. Zeller, Learning user interface element interactions, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 296–306. URL: https://doi.org/10.1145/3293882.3330569. doi:10.1145/3293882.3330569.

[71] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, Y. Donmez, Qbe: Qlearning-based exploration of android applications, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018, pp. 105–115. doi:10.1109/ICST.2018.00020.

[72] Y. Koroglu, A. Sen, Functional test generation from ui test scenarios using reinforcement learning for android applications, Software Testing Verification and Reliability (2020). URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85092015479&doi=10.1002%2fstvr.1752&partnerID=40&md5=f43056f29da7195d72ff0e731caf5989. doi:10.1002/stvr.1752, cited By 0.

[73] Y. Koroglu, A. Sen, Functional test generation from ui test scenarios using reinforcement learning for android applications, Software Testing Verification and Reliability 31 (2021). URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85092015479&doi=10.1002%2fstvr.1752&partnerID=40&md5=f43056f29da7195d72ff0e731caf5989. doi:10.1002/stvr.1752, cited By 0.

[74] L. Mariani, M. Pezze, O. Riganelli, M. Santoro, Autoblacktest: Automatic black-box testing of interactive applications, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 81–90. doi:10.1109/ICST.2012.88.

[75] M. Pan, A. Huang, G. Wang, T. Zhang, X. Li, Reinforcement learning based curiosity-driven testing of android applications, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 153–164. URL: https://doi.org/10.1145/3395363.3397354. doi:10.1145/3395363.3397354.

[76] T. A. T. Vuong, S. Takada, A reinforcement learning based approach to automated testing of android applications, in: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018, Association for Computing Machinery, New

York, NY, USA, 2018, p. 31–37. URL: https://doi.org/10.1145/3278186.3278191. doi:10.1145/3278186.3278191.

[77] H. Yasin, S. Hamid, R. Yusof, Droidbotx: Test case generation tool for android applications using q-learning, Symmetry 13 (2021) 1–30. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85101268101&doi=10.3390%2fsym13020310&partnerID=40&md5=cd764a006ecffc3e11a8f98718b7c23c. doi:10.3390/sym13020310, cited By 0.

[78] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, Y. Liu, Automatic web testing using curiosity-driven reinforcement learning, 2021, pp. 423–435. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85112295528&doi=10.1109%2fICSE43902.2021.00048&partnerID=40&md5=b8f8e252f1c7f4df056b50481ed991ea. doi:10.1109/ICSE43902.2021.00048, cited By 1.

[79] Y. Li, Z. Yang, Y. Guo, X. Chen, Humanoid: A deep learning-based approach to automated black-box android app testing, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE '19, IEEE Press, 2019, p. 1070–1073. URL: https://doi.org/10.1109/ASE.2019.00104. doi:10.1109/ASE.2019.00104.

[80] D. Santiago, P. J. Clarke, P. Alt, T. M. King, Abstract flow learning for web application test generation, in: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 49–55. URL: https://doi.org/10.1145/3278186.3278194. doi:10.1145/3278186.3278194.

[81] D. Santiago, J. Phillips, P. Alt, B. Muras, T. King, P. Clarke, Machine learning and constraint solving for automated form testing, volume 2019-October, 2019, pp. 217–227. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.

0-85081113091&doi=10.1109%2fISSRE.2019.00030&
partnerID=40&md5=1be86067a371b94f0bd32074f2005a34.
doi:10.1109/ISSRE.2019.00030, cited By 0.

[82] M. M. Kamal, S. M. Darwish, A. Elfatatry, Enhancing the automation of gui testing, in: Proceedings of the 2019 8th International Conference on Software and Information Engineering, ICSIE '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 66–70. URL: https://doi.org/10.1145/3328833.3328842. doi:10.1145/3328833.3328842.

[83] N. Walkinshaw, G. Fraser, Uncertainty-driven black-box test data generation, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 253–263. doi:10.1109/ICST.2017.30.

[84] I. Hooda, R. Chhillar, Test case optimization and redundancy reduction using ga and neural networks, International Journal of Electrical and Computer Engineering 8 (2018) 5449–5456. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85066159186&doi=10.11591%2fijece.v8i6.pp5449-5456&partnerID=40&md5=1f455167b829969e3bc6e626b3b5d65d. doi:10.11591/ijece.v8i6.pp5449-5456, cited By 3.

[85] H. Almulla, G. Gay, Generating diverse test suites for gson through adaptive fitness function selection, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 12420 LNCS (2020) 246–252. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85092933212&doi=10.1007%2f978-3-030-59762-7_18&partnerID=40&md5=f1ae2eee34d85dd191295cd2ed4ee57a. doi:10.1007/978-3-030-59762-7_18, cited By 0.

[86] A. Groce, Coverage rewarded: Test input generation via adaptation-based programming, in: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, IEEE Computer Soci-

ety, USA, 2011, p. 380–383. URL: https://doi.org/10.1109/ASE.2011.6100077. doi:10.1109/ASE.2011.6100077.

[87] W. He, R. Zhao, Q. Zhu, Integrating evolutionary testing with reinforcement learning for automated test generation of object-oriented software, Chinese Journal of Electronics 24 (2015) 38–45. doi:10.1049/cje.2015.01.007.

[88] E. Hershkovich, R. Stern, R. Abreu, A. Elmishali, Prioritized test generation guided by software fault prediction, in: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2021, pp. 218–225. doi:10.1109/ICSTW52544.2021.00045.

[89] S. Ji, Q. Chen, P. Zhang, Neural network based test case generation for data-flow oriented testing, in: 2019 IEEE International Conference On Artificial Intelligence Testing (AITest), 2019, pp. 35–36. doi:10.1109/AITest.2019.00-11.

[90] T. Ahmad, A. Ashraf, D. Truscan, I. Porres, Exploratory performance testing using reinforcement learning, in: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2019, pp. 156–163. doi:10.1109/SEAA.2019.00032.

[91] J. Koo, C. Saumya, M. Kulkarni, S. Bagchi, Pyse: Automatic worst-case test generation by reinforcement learning, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 136–147. doi:10.1109/ICST.2019.00023.

[92] M. Helali Moghadam, M. Saadatmand, M. Borg, M. Bohlin, B. Lisper, Machine learning to guide performance testing: An autonomous test framework, in: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2019, pp. 164–167. doi:10.1109/ICSTW.2019.00046.

[93] A. Sedaghatbaf, M. H. Moghadam, M. Saadatmand, Automated performance testing based on active deep learning, in: 2021 IEEE/ACM International Confer-

ence on Automation of Software Test (AST), 2021, pp. 11–19. doi:10.1109/AST52587.2021.00010.

[94] Q. Luo, D. Poshyvanyk, A. Nair, M. Grechanik, Forepost: A tool for detecting performance problems with feedback-driven learning software testing, in: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 593–596. URL: https://doi.org/10.1145/2889160.2889164. doi:10.1145/2889160.2889164.

[95] H. Schulz, D. Okanović, A. van Hoorn, P. Tůma, Context-tailored workload model generation for continuous representative load testing, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering, ICPE '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 21–32. URL: https://doi.org/10.1145/3427921.3450240. doi:10.1145/3427921.3450240.

[96] Y. Jia, M. B. Cohen, M. Harman, J. Petke, Learning combinatorial interaction test generation strategies using hyperheuristic search, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, IEEE Press, 2015, p. 540–550.

[97] L. Mudarakola, J. Sastry, C. Vudatha, Generating test cases for testing web sites through neural networks and input pairs, International Journal of Applied Engineering Research 9 (2014) 11819–11831. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84925945679&partnerID=40&md5=b04e683a64af28552e91ef9f0ae6fce7, cited By 7.

[98] L. Mudarakola, J. Sastry, A neural network based strategy (nnbs) for automated construction of test cases for testing an embedded system using combinatorial techniques, International Journal of Engineering and Technology(UAE) 7 (2018) 74–81. URL: https://www.scopus.com/inward/

record.uri?eid=2-s2.0-85067270771&partnerID=40&md5=397c33d415fed6913c48d4bb4751dc7a, cited By 6.

[99] R. Patil, V. Prakash, Neural network based approach for improving combinatorial coverage in combinatorial testing approach, Journal of Theoretical and Applied Information Technology 96 (2018) 6677–6687. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056233628&partnerID=40&md5=4f989bf779060baabae4e8302a603f91, cited By 2.

[100] C. Duy Nguyen, P. Tonella, Automated inference of classifications and dependencies for combinatorial testing, 2013, pp. 622–627. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84893575078&doi=10.1109%2fASE.2013.6693123&partnerID=40&md5=102d724421c89c7ae1f70e2a20020355. doi:10.1109/ASE.2013.6693123, cited By 1.

[101] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, M. Souza, A machine learning approach to generate test oracles, in: Proceedings of the XXXII Brazilian Symposium on Software Engineering, SBES '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 142–151. URL: https://doi.org/10.1145/3266237.3266273. doi:10.1145/3266237.3266273.

[102] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, W. Yang, Glib: Towards automated test oracle for graphically-rich applications, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 1093–1104. URL: https://doi.org/10.1145/3468264.3468586. doi:10.1145/3468264.3468586.

[103] F. Gholami, N. Attar, H. Haghighi, M. V. Asl, M. Valueian, S. Mohamadyari, A classifier-based test oracle for embedded software, in: 2018 Real-Time and

Embedded Systems and Technologies (RTEST), 2018, pp. 104–111. doi:10.1109/RTEST.2018.8397165.

[104] H. Khosrowjerdi, K. Meinke, A. Rasmusson, Learning-based testing for safety critical automotive applications, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10437 LNCS (2017) 197–211. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85029520480&doi=10.1007%2f978-3-319-64119-5_13&partnerID=40&md5=a74bc1966cd142e83070da2e7cc1bb37. doi:10.1007/978-3-319-64119-5_13, cited By 7.

[105] H. Khosrowjerdi, K. Meinke, Learning-based testing for autonomous systems using spatial and temporal requirements, in: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, MASES 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 6–15. URL: https://doi.org/10.1145/3243127.3243129. doi:10.1145/3243127.3243129.

[106] W. Makondo, R. Nallanthighal, I. Mapanga, P. Kadebu, Exploratory test oracle using multi-layer perceptron neural network, in: 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2016, pp. 1166–1171. doi:10.1109/ICACCI.2016.7732202.

[107] S. Shahamiri, W. Wan Kadir, S. Bin Ibrahim, An automated oracle approach to test decision-making structures, volume 5, 2010, pp. 30–34. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-77958584496&doi=10.1109%2fICCSIT.2010.5563989&partnerID=40&md5=732483e9576df4cfb81151cf2f666730. doi:10.1109/ICCSIT.2010.5563989, cited By 10.

[108] F. Tsimpourlas, A. Rajan, M. Allamanis, Supervised learning over test executions as a test oracle, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, Association for Computing Machinery, New

York, NY, USA, 2021, p. 1521–1531. URL: https://doi.org/10.1145/3412841.3442027. doi:10.1145/3412841.3442027.

[109] D. J. Richardson, S. L. Aha, T. O'Malley, Specification-based test oracles for reactive systems, in: Proc. of the 14th Int'l Conf. on Software Engineering, Springer, 1992, pp. 105–118.

[110] K. K. Aggarwal, Y. Singh, A. Kaur, O. P. Sangwan, A neural net based approach to test oracle, SIGSOFT Softw. Eng. Notes 29 (2004) 1–6. URL: https://doi.org/10.1145/986710.986725. doi:10.1145/986710.986725.

[111] A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, M. Arratibel, Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms, in: 2021 IEEE/ACM International Conference on Automation of Software Test (AST), 2021, pp. 30–39. doi:10.1109/AST52587.2021.00012.

[112] J. Ding, D. Zhang, A machine learning approach for developing test oracles for testing scientific software, volume 2016-January, 2016, pp. 390–395. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84988431007&doi=10.18293%2fSEKE2016-137&partnerID=40&md5=ab9dd29e1f7369a3a2e41933a169d76e. doi:10.18293/SEKE2016-137, cited By 4.

[113] A. Gartziandia, A. Arrieta, A. Agirre, G. Sagardui, M. Arratibel, Using regression learners to predict performance problems on software updates: A case study on elevators dispatching algorithms, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 135–144. URL: https://doi.org/10.1145/3412841.3441894. doi:10.1145/3412841.3441894.

[114] H. Jin, Y. Wang, N. Chen, Z. Gou, S. Wang, Artificial neural network for automatic test oracles generation, in: 2008 International Conference on

Computer Science and Software Engineering, volume 2, 2008, pp. 727–730. doi:10.1109/CSSE.2008.774.

[115] A. Monsefi, B. Zakeri, S. Samsam, M. Khashehchi, Performing software test oracle based on deep neural network with fuzzy inference system, Communications in Computer and Information Science 891 (2019) 406–417. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85075817713&doi=10.1007%2f978-3-030-33495-6_31&partnerID=40&md5=e74f5939bdd085a046c2f882260287fa. doi:10.1007/978-3-030-33495-6_31, cited By 0.

[116] O. P. Sangwan, P. K. Bhatia, Y. Singh, Radial basis function neural network based approach to test oracle, SIGSOFT Softw. Eng. Notes 36 (2011) 1–5. URL: https://doi.org/10.1145/2020976.2020992. doi:10.1145/2020976.2020992.

[117] S. Shahamiri, W. Kadir, S. Ibrahim, S. Hashim, An automated framework for software test oracle, Information and Software Technology 53 (2011) 774–788. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-79955055107&doi=10.1016%2fj.infsof.2011.02.006&partnerID=40&md5=a86d6be1213fba799db8e24df632367c. doi:10.1016/j.infsof.2011.02.006, cited By 31.

[118] S. Shahamiri, W. Wan-Kadir, S. Ibrahim, S. Hashim, Artificial neural networks as multi-networks automated test oracle, Automated Software Engineering 19 (2012) 303–334. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84863642080&doi=10.1007%2fs10515-011-0094-z&partnerID=40&md5=92dcec011b1a06b3275252dd2d1449cf. doi:10.1007/s10515-011-0094-z, cited By 23.

[119] A. Singhal, A. Bansal, A. Kumar, An approach to design test oracle for aspect oriented software systems using soft computing approach, International

Journal of Systems Assurance Engineering and Management 7 (2016) 1–5. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84957956401&doi=10.1007%2fs13198-015-0402-2&partnerID=40&md5=99e4f57c672249b63483b7fc06ded311. doi:10.1007/s13198-015-0402-2, cited By 1.

[120] M. Vanmali, M. Last, A. Kandel, Using a neural network in the software testing process, International Journal of Intelligent Systems 17 (2002) 45–62. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0036157249&doi=10.1002%2fint.1002&partnerID=40&md5=81a29cc673c51e8f659a1676551ec393. doi:10.1002/int.1002, cited By 63.

[121] Vineeta, A. Singhal, A. Bansal, Generation of test oracles using neural network and decision tree model, in: 2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence), 2014, pp. 313–318. doi:10.1109/CONFLUENCE.2014.6949311.

[122] M. Ye, B. Feng, L. Zhu, Y. Lin, Neural networks based automated test oracle for software testing, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 4234 LNCS - III (2006) 498–507. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-33750708220&partnerID=40&md5=858633bde3ccbdac0195bd004eb4141e, cited By 9.

[123] R. Zhang, Y.-W. Wang, M.-Z. Zhang, Automatic test oracle based on probabilistic neural networks, Advances in Intelligent Systems and Computing 752 (2019) 437–445. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85053258403&doi=10.1007%2f978-981-10-8944-2_50&partnerID=40&md5=81b1448d0ff5e3381e84522956705caa. doi:10.1007/978-981-10-8944-2_50, cited By 0.

[124] D. J. Hiremath, M. Claus, W. Hasselbring, W. Rath, Automated identification of

metamorphic test scenarios for an ocean-modeling application, in: 2020 IEEE International Conference On Artificial Intelligence Testing (AITest), 2020, pp. 62–63. doi:10.1109/AITEST49225.2020.00016.

[125] D. J Hiremath, M. Claus, W. Hasselbring, W. Rath, Towards automated metamorphic test identification for ocean system models, in: 2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET), 2021, pp. 42–46. doi:10.1109/MET52542.2021.00014.

[126] H. Spieker, A. Gotlieb, Adaptive metamorphic testing with contextual bandits, Journal of Systems and Software 165 (2020) 110574. URL: http://www.sciencedirect.com/science/article/pii/S0164121220300558. doi:https://doi.org/10.1016/j.jss.2020.110574.

[127] U. Kanewala, J. Bieman, Using machine learning techniques to detect metamorphic relations for programs without test oracles, 2013, pp. 1–10. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84893326644&doi=10.1109%2fISSRE.2013.6698899&partnerID=40&md5=947a8b49ea54dd44a9656fa5110480fa. doi:10.1109/ISSRE.2013.6698899, cited By 30.

[128] U. Kanewala, J. Bieman, A. Ben-Hur, Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels, Software Testing Verification and Reliability 26 (2016) 245–269. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84963615021&doi=10.1002%2fstvr.1594&partnerID=40&md5=89589cb8ce1bf35471174edb2e7e6df5. doi:10.1002/stvr.1594, cited By 35.

[129] O. Korkmaz, C. Yilmaz, Sysmodis: A systematic model discovery approach, 2021, pp. 67–76. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85108026461&doi=10.1109%2fICSTW52544.2021.00023&partnerID=40&

md5=cc2b4342dd8c026e3268914222b5fd66. doi:10.1109/ICSTW52544.2021.00023, cited By 0.

[130] A. Nair, K. Meinke, S. Eldh, Leveraging mutants for automatic prediction of metamorphic relations using machine learning, in: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, MaLTeSQuE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 1–6. URL: https://doi.org/10.1145/3340482.3342741. doi:10.1145/3340482.3342741.

[131] G. Shu, D. Lee, Testing security properties of protocol implementations - a machine learning based approach, in: 27th International Conference on Distributed Computing Systems (ICDCS '07), 2007, pp. 25–25. doi:10.1109/ICDCS.2007.147.

[132] P. Zhang, X. Zhou, P. Pelliccione, H. Leung, Rbf-mlmr: A multi-label metamorphic relation prediction approach using rbf neural network, IEEE Access 5 (2017) 21791–21805. doi:10.1109/ACCESS.2017.2758790.

[133] J. Chen, L. Zhu, T. Y. Chen, D. Towey, F.-C. Kuo, R. Huang, Y. Guo, Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering, Journal of Systems and Software 135 (2018) 107–125. URL: https://www.sciencedirect.com/science/article/pii/S0164121217302170. doi:https://doi.org/10.1016/j.jss.2017.09.031.

[134] P. McMinn, M. Stevenson, M. Harman, Reducing qualitative human oracle costs associated with automatically generated test data, in: Proceedings of the First International Workshop on Software Test Output Validation, STOV '10, ACM, New York, NY, USA, 2010, pp. 1–4. URL: http://doi.acm.org/10.1145/1868048.1868049. doi:10.1145/1868048.1868049.

[135] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, Y. Jiang, Implications of ceiling effects in defect predictors, in: Proceedings of the

4th International Workshop on Predictor Models in Software Engineering, PROMISE '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 47–54. URL: https://doi.org/10.1145/1370788.1370801. doi:10.1145/1370788.1370801.

[136] L. Taylor, G. Nitschke, Improving deep learning using generic data augmentation, 2017. arXiv:1708.06020.

[137] E. Kocaguneli, T. Menzies, J. W. Keung, On the value of ensemble effort estimation, IEEE Transactions on Software Engineering 38 (2012) 1403–1416. doi:10.1109/TSE.2011.111.

[138] L. L. Minku, A novel online supervised hyperparameter tuning procedure applied to cross-company software effort estimation, Empirical Software Engineering 24 (2019) 3153–3204. URL: https://doi.org/10.1007/s10664-019-09686-w. doi:10.1007/s10664-019-09686-w.

[139] G. Gay, R. Just, Defects4j as a challenge case for the search-based software engineering community, in: A. Aleti, A. Panichella (Eds.), Search-Based Software Engineering, Springer International Publishing, Cham, 2020, pp. 255–261.

[140] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, M. R. Prasad, Bugs.jar: A large-scale, diverse dataset of real-world java bugs, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 10–13. URL: https://doi.org/10.1145/3196398.3196473. doi:10.1145/3196398.3196473.