IVVES
Industrial-grade Verification and Validation of Evolving Systems
Labelled in ITEA3, a EUREKA cluster, Call 5

ITEA3 Project Number 18022

D3.1 – State of the art of validation methods and techniques for complex evolving systems

Due date of deliverable: June 30, 2020
Actual date of submission: June 30, 2020

Start date of project: 1 October 2019 **Duration:** 39 months

Organisation name of lead contractor for this deliverable: F-Secure

Author(s): Matvey Pashkovskiy (F-Secure, FIN), Mahshid Helali Moghadam (RISE, SWE), Paul Derckx, Mark van Helvoort (Philips, NLD), Almira Pillay (Sogeti, NLD), Olav Bandmann (Prover, SWE), Tommi Mikkonen (University of Helsinki, FIN), Tanja Vos (The Open University of The Netherlands, NLD), Juan Leandro Sánchez (SII CONCATEL/NETCHECK, ESP)

Status: Final

Version number: V1.0

Submission Date: 30-June-2020

Doc reference: IVVES_Deliverable_D3.1_V1.0.docx

Work Pack./ Task: WP3

Description:
(max 5 lines)

Nature:	☑ **R**=Report, ☐ **P**=Prototype, ☐ **D**=Demonstrator, ☐ **O**=Other		
Dissemination Level:	**PU**	Public	**X**
	PP	Restricted to other programme participants	
	RE	Restricted to a group specified by the consortium	
	CO	Confidential, only for members of the consortium	

DOCUMENT HISTORY

Release	Date	Reason of change	Status	Distribution
V0.1	04/06/2020	Structure is created	Draft	All
V0.2	25/06/2020	Added content	Draft	WP3
V0.3	29/06/2020	List of authors is adjusted	Draft	Submitted to PMT
V1.0	30/06/2020	Approved by PMT, to be submitted to ITEA3	Final	Uploaded to ITEA

Table of Contents

Glossary

Abbreviation / acronym	Description
AFP	Automated Function Points
AI	Artificial Intelligence
APHFW	Average Percentage of Historical Failure with time Window
BOW	Bag-of-Words
CD	Continuous Delivery
CI	Continuous Integration
DBRNN-A	Deep Bidirectional Recurrent Neural Network with Attention
ddmin	Minimizing Delta Debugging
DevOps	software Development (Dev) and IT Operations (Ops)
DNN	Deep Neural Network
E2E	End-to-End
ES	Evolving System
GUI	Graphical User Interface
IDP	Inverse Defect Prediction
ILP	Inductive Logic Programming
LFR	Low Fault Risk
LSTM	Long Short-Term Memory
ML	Machine learning
MR	Metamorphic Relation
NLP	Natural Language Processing
PoC	Proof of Concept
RL	Reinforcement Learning
RBT	Risk-based Testing
RNN	Recurrent Neural Network
QA	Quality Assurance
rSVM	Recurrent Support Vector Machine
SLR	Systematic Literature Review
SUT	Software Under Test
TA	Test Automation
TCP	Test Case Prioritization
TCS	Test Case Selection
TD	Temporal-Difference
UBST	Usage-Based Statistical Testing
UI	User interface

1. Executive Summary

This report describes the state of the art of validation methods and techniques for complex Evolving Systems (ES). It introduces the idea of continuous quality assessment process which spans on entire ES lifecycle and maps methods, techniques and existing tools helping partners to navigate in the domain and apply right approach for the right lifecycle stage.

It is visible from the report that the most expensive stage of ES lifecycle is tests maintenance and a lot of techniques could be applied there thus providing the biggest benefit to companies.
It is important to understand that building the basement for ES development and operation consisting of CI/CD and data collection pipelines is necessary for applying state of the art methods and techniques and even classical engineering solutions often can bring more value and be more efficient in terms of expenses than tools utilizing latest and greatest ML models.

Though a lot of researches have been done in the domain of ES validation and verification it is visible that not many of those got implemented and made available for industry. We agree that one of the main next steps should be focused on addressing very specific problem with selecting and implementing of the approach which will bring the most benefits and cover big market share.

There are three sub-domains that could be considered as main focus areas:

1. Model-based test generation with automatic model building:
    - as it can provide companies with high level end-to-end regression testing suites and requires only basic knowledge and skillset from engineers;
    - some tools are already publicly available, but applicability of those tools is unclear;
2. ML-assisted test generation: tester (testing system) is intelligent and learns the optimal policy (way) to generate the test cases meeting the testing objective:
    - as it can provide automated test generation without access to source code or system model;
    - in some cases, it is able to reuse the gained knowledge (learned policy) in further similar testing situations (transfer learning);
3. Automatic test selection and prioritization as it, when applied, reduces TA infrastructure costs and feedback time allowing teams work in the most efficient manner.

2. Introduction

In order to better understand the holistic picture of current state of the art of validation techniques for complex ES it is important to map ES lifecycle in two-dimensional space where vertical (Y) axis used to measure complexity and horizontal (X) lifecycle stage. Figure 1 displays that mapping. Because software development is iterative process after Operation phase stage Design begins again. It is important understand that techniques used for high complexity software relies on simpler ones, so “Automatic test prioritization” almost impossible to implement having no components in use like “CI / CD pipeline”, “Test results data collection” and “Coverage analysis”. Also, it is obvious from the Figure 1 that most of the work and techniques in continuous quality assurance process are dedicated to tests maintenance stage.

Figure 1: Continuous quality assurance process stages and components

3. State of the art of validation techniques for complex ES

Let's consider the most advanced techniques of validation and verification of complex ES in more details. Mostly state of the art techniques and methods are described below, though several state of the practice techniques with low adoption level are mentioned as well. Methods and techniques are broken down into the sections accordingly described continuous quality process.

3.1 Design

Work on ES quality starts at the very beginning of the its lifecycle, on design stage.
Design stage consist of planning of new features and improvements. As a state of the practice the planning involves manual modelling and architecting which sometimes involves building proof-of-concept (PoC) systems and prioritisation of incoming feedback in form of support cases, surveys, interview reports. Also, during design phase threat modelling methodologies are used to identify possible security issues.

As a state of the art more complex techniques and tools are employed.

3.1.1 Modelling

Along with building PoC modelling could be performed during the design stage. Modelling could be considered as an approach to perform testing before even building the actual software. One of the ways to perform modelling could be verification of the specification written with TLA+ language.

TLA+ is a formal specification language developed by Leslie Lamport. It is used to design, model, document, and verify programs, especially concurrent systems and distributed systems. TLA+ has been described as exhaustively-testable pseudocode, and its use likened to drawing blueprints for software systems; TLA is an acronym for Temporal Logic of Actions.

For design and documentation, TLA+ fulfils the same purpose as informal technical specifications. However, TLA+ specifications are written in a formal language of logic and mathematics, and the precision of specifications written in this language is intended to uncover design flaws before system implementation is underway.

Since TLA+ specifications are written in a formal language, they are amenable to finite model checking. The model checker finds all possible system behaviours up to some number of execution steps, and examines them for violations of desired invariance properties such as safety and liveness. TLA+ specifications use basic set theory to define safety (bad things won't happen) and temporal logic to define liveness (good things eventually happen).

TLA+ is also used to write machine-checked proofs of correctness both for algorithms and mathematical theorems. The proofs are written in a declarative, hierarchical style independent of any single theorem prover backend. Both formal and informal structured mathematical proofs can be written in TLA+; the language is similar to LaTeX, and tools exist to translate TLA+ specifications to LaTeX documents.

Temporal logic of actions (TLA) and TLA+, PlusCAL languages are used by several companies to identify problems in ES design.

At Microsoft, a critical bug was discovered in the Xbox 360 memory module during the process of writing a specification in TLA+.[i] TLA+ was used to write formal proofs of correctness for Byzantine Paxos and components of the Pastry distributed hash table.[ii]

Amazon Web Services has used TLA+ since 2011. TLA+ model checking uncovered bugs in DynamoDB, S3, EBS, and an internal distributed lock manager; some bugs required state traces of 35 steps. Model checking was also used to verify aggressive optimizations. In addition, TLA+ specifications were found to hold value as documentation and design aids.[iii iv]

Microsoft Azure used TLA+ to design Cosmos DB, a globally-distributed database with five different consistency models.[v,vi]

3.1.2 Automatic defect triaging

Defect management processes require defects to be classified, scored/prioritized and allocated to the appropriate development teams. Traditionally still in Defect Review Boards meetings defects are discussed, assessed and decisions are subsequently taken, a time-consuming activity. Agile ways of working require this process to be more efficient and almost "continuous".

The major challenge is that the defect descriptions and associated information often contain a combination of e.g. free unstructured text, code snippets, and stack trace making the input data highly noisy.

Automatic defect triaging algorithms can be formulated as a classification problem, which takes the reported bug information as the input, mapping it to one of the available developers (class labels). Also, it is possible to do assignment of the severity class and related features.
Manual bug triaging is usually performed using the bug report content, primarily consisting of the summary and description. While additional sources of input have been explored in the literature such as developer profiling from GitHub[vii] and using component information[viii], majority of the research efforts have focused on leveraging the bug report content for triaging[ix,x,xi,xii,xiii,xiv,xv]. The bug report content contains noisy text information including code snippets, and stack trace details. Processing such unstructured and noisy text data is a major challenge in training a classifier.

Natural language processing (NLP) methods like bag-of-words (BOW), bag-of-n-grams, word2vec and more advanced models employing neural networks are used to build classifiers. It is possible that BOW model mis-classifies defects because:

1. BOW feature model considers the sentence as a bag-of-words losing the ordering (context) of words, and
2. the semantic similarity between synonymous words in the sentence are not considered.

Even though a bag-of-n-grams model considers a small context of word ordering, they suffer from high dimensionality and sparse data[xvi]. The semantic similarity between word tokens can be learnt using a skip-gram based neural network model called word2vec[xvii]. This model relies on distributional hypothesis which claims that words that appear in the same context in the sentence share a semantic meaning. Ye et al.,[xviii] built a shared word representation using word2vec for word tokens present in code language and word tokens present in descriptive language. The main disadvantage of word2vec is that it learns a semantic representation of individual word tokens, however, does not consider a sequence of word tokens such as a sentence. An extension of word2vec called paragraph vector[xix] considers the ordering of words, but only for a small context. Recently, recurrent neural network (RNN) based deep learning algorithms have revolutionized the concept of word sequence representation and have shown promising breakthroughs in many applications such as language modelling and machine translation. Lam et al.[xx] used deep neural network (DNN) with rSVM to learn a common representation between source code and the bug reports and used it for effective bug localization. White et al.,[xxi] provided a broad perspective on how deep learning can be used in software repositories to solve some challenging problems. A novel bug report representation approach is proposed using DBRNN-A: Deep Bidirectional Recurrent Neural Network with Attention mechanism and with Long Short-Term Memory units (LSTM)[xxii]. Table 1 presents a list of closely related works on bug triaging arranged in a chronological order (year 2010 to 2018).

Table 1: Summary of various ML based bug triaging approaches available in literature, explaining the features and approach used along with its experimental performance.

Paper	Information used	Feature extracted	Approach	Dataset	Performance
Bhattacharya et al., 2010[viii]	title, description, keywords, product, component, last developer activity	tf-idf + bagof-words	Naive Bayes + Tossing graph	Eclipse# 306,297	Rank#5 accuracy 77.43%
				Mozilla# 549,962	Rank#5 accuracy 77.87%
Tamrawi et al., 2011[xii]	title, description	terms	A fuzzy-set feature for each word	Eclipse# 69829	Rank#5 accuracy 68.00%
Anvik et. Al., 2011[ix]	title, description	normalized tf	Naive Bayes, EM, SVM, C4.5, nearest neighbour, conjunctive rules	Eclipse# 7,233	Rank#3 prec. 60%, recall 3%
				Firefox# 7,596	Rank#3 prec. 51%, recall 24%
Xuan et. Al., 2012[xv]	title, description	tf-idf, developer prioritization	Naive Bayes, SVM	Eclipse# 49,762	Rank#5 accuracy 53.10%
				Mozilla# 30,609	Rank#5 accuracy 56.98%
Shokripour et al. 2013[xi]	title, description, detailed source code info	weighted unigram noun terms	Bug location prediction + developer expertise	JDT-Debug# 85	Rank#5 accuracy 89.41%
				Firefox# 80	Rank#5 accuracy 59.76%
Wang et al., 2014[xiii]	title, description	tf	Active developer cache	Eclipse# 17,937	Rank#5 accuracy 84.45%
				Mozilla# 69,195	Rank#5 accuracy 55.56%
Xuan et. al., 2015[xiv]	title, description	tf	feature selection with Naive Bayes	Eclipse# 50,000	Rank#5 accuracy 60.40%
				Mozilla# 75,000	Rank#5 accuracy 46.46%
Badashian et. al., 2015[vii]	title, description, keyword, project language, tags from stackoverflow, github	Keywords from bug and tags	Social expertise with matched keywords	20 GitHub projects, 7144 bug reports	Rank#5 accuracy 89.43%
Jonsson et. al., 2016[x]	title, description	tf-idf	Stacked Generalization of a classifier ensemble	Industry# 35,266	Rank#1 accuracy 89%
Senthil Mani et al.[xxiii]	title, description	terms	DBRNN-A	Google Chromium# 383,104	Rank#10 accuracy 47%

				Mozilla Core# 314,388	Rank#10 accuracy 43%
				Mozilla Firefox# 162,307	Rank#10 accuracy 56%

3.2 Development and testing

During the development and/or construction stage the product is built (the code is written) and assembled in accordance with the requirements specified in the product, process and material specifications and is deployed and tested within the testing environment. System assessments are conducted in order to correct deficiencies and adapt the system for continued improvement.

3.2.1 Code anomaly detection

Anomaly detection is the process of identifying unexpected items or events in a structure or software, where anomalies are defined as events or behaviours which differ from the norm[xxiv]. Unexpected behaviour of software can lead to numerous risks, one of them being profit loss and loss of customers, other being safety concerns. It is of high importance to detect software anomalies as early as possible in order to mitigate these risks, so software testing and peer reviews have become a must in any development cycle. Even though testing and peer reviews are valuable, they require time and resources, and this is where code anomaly detection brings value. Developing code is the foundation of any software or model and finding anomalies at this, most granular stage, can help in early deviation detection and faster deployment.

Developing code can go wrong for many reasons, the most high-level one being simple misunderstanding of what is required from the stakeholders. In that sense, even healthy code is erroneous. Therefore, it is very important to lay the ground and explain what the expected behaviour of source code is and what would classify as an anomaly. Anomalies can be divided into three types[xxv]:

1. **Point** anomalies, single instances with attributes different than the general population's norm;
2. **Contextual** anomalies, which are context specific, and common in time-series data; and
3. **Collective** anomalies, a set of data instances which can collectively be considered anomalies.

Code anomalies are fragments of code that are not typical within the community or an ecosystem of a given programming language[xxvi]. Erroneous code snippets highlight flaws in language design or indicate problems in software behaviour. Identifying code anomalies at a scale of a programming language means that a large corpus of source code needs to be prepared for digestion by a given ML algorithm, which in turn classifies it. Some approaches to classification by ML are[xxvii]:

1. **Supervised Anomaly Detection**, which requires a labelled dataset containing both normal and anomalous samples to construct a predictive model to classify future data points. The most used algorithms for this purpose are supervised Neural Networks, Support Vector Machines, K-Nearest Neighbours Classifier, etc;
2. **Unsupervised Anomaly Detection**, which requires no training data and has two assumptions about the data:
   a. Only a small percentage of data is anomalous; and
   b. Any anomaly is statistically different from the normal samples.

Based on the above assumptions, the data is then clustered using a similarity measure and the data points which are far off from the cluster are defined as anomalies.
Classifying code as defective can be done on different levels:

1. **Change** log level, where metrics are extracted from the versioning system and most recent files are source of anomalies[xxviii];
2. **Method** level, where methods are the source of anomalies[xxix];
3. **Component** level, where components are the source of anomalies;

4. **File** level, where files are the source of anomalies. Usually, the bigger the file is, the higher the probability of anomalies is[xxx];
5. **Within project**, where a classifier is trained on a set of data from a given project and then used to predict the anomalies in the same project. This level is further divided into inner and cross-version anomaly detection based on which versions of the project are used[xxv];
6. **Cross project**, where a classifier is trained on a previous project and predicts the anomalies in a new one[xxv].

Different code anomalies detected by unsupervised learning method, autoencoder model, in Kotlin programming language are[xxvi]:

1. **Syntax tree anomalies**, nontypical and rare code fragments, divided into:
   a. **Language design anomalies** used to improve the design of programming language itself;
   b. **Compiler anomalies** used as performance tests in compiler correctness;
   c. **Performance anomalies** that point to non-optimal code generation and lack of optimization.
2. **Compiler-induced anomalies**, where complex bytecode was generated through typical syntax tree, caused by:
   a. **Non optimal code generation anomalies** used as tests for bytecode generation;
   b. **Complex functions inlining abnormal code fragments** and being called multiple times when executing code. This can be used to detect and mitigate performance risks.

The summary of methods and tools for code anomaly detection can be found in Table 2.

Table 2: Summary of methods and tools for code anomaly detection

Tool/approach/algorithm	Objective	Method	Reference
GrouMiner tool	Detect anomalous patterns in object interaction in **Java**	Graph-based anomaly detection	xxxi
Mining usage model approach	Detect abnormal usage patterns	Graph-based anomaly detection	xxxii
DIDUCE tool	Dynamic code expression analysis in **Java**	Dynamic code analysis	xxxiii
Feature Envy approach	Identify code patterns indicating architecture flaws	"Code smells"	xxxiv
Supervised learning algorithms	Classify anomalies	Neural Networks, Support Vector Machines, K-Nearest Neighbours Classifier	xxxv, xxxvi, xxxvii, xxxviii, xxxix, xl
Unsupervised learning algorithms	Cluster anomalies	Autoencoders	xxv, xxvi

3.2.2 Formal verification

In general, the term "formal methods" refers to "mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems"[xli]. Formal verification consists of mathematically proving properties of mathematical models of systems. This definition covers a wide range of areas and techniques, but in this context, we will focus on model checking[xlii] of safety properties applied to embedded systems.

Model checking is a family of techniques used to verify properties of finite state systems. The systems are modelled using temporal logic and then the properties are checked to hold over the entire (finite) state space. These techniques have seen a significant boost in performance and usability over the last 15 years[xliii] and are now more broadly used in the industry.

In order to get a wider adoption in the industry, performance is one of the key issues that needs to be addressed. This area has been a very active field of research[xliv xlv], and more recently, inspired by the impressive results of ML (deep learning in particular), there has been a growing interest in applying ML to formal verification. So far, there are some promising results[xlvi], but this new direction is still in its infancy.

3.2.3 Risk-based testing

The aim of risk-based testing (RBT) approaches is to ensure that appropriate testing activities are identified and prioritized based on risk[xlvii]. Furthermore, we may use testing to support risk analysis and risk analysis to support testing. Fundamentally, the goal of RBT is to reduce the risk of failure to the business and increase customer satisfaction.

Several RBT approaches were proposed in academia (e.g., xlviii, xlix, l), and industry (e.g., li, lii, liii, liv, lv). Moreover, the international standard ISO/IEC/IEEE 29119 Software Testing[lvi] on testing techniques, processes, and documentation even explicitly considers risks as an integral part of the test planning process. To a degree, proposed approaches are overlapping and include common elements, but they all also have their specifics.

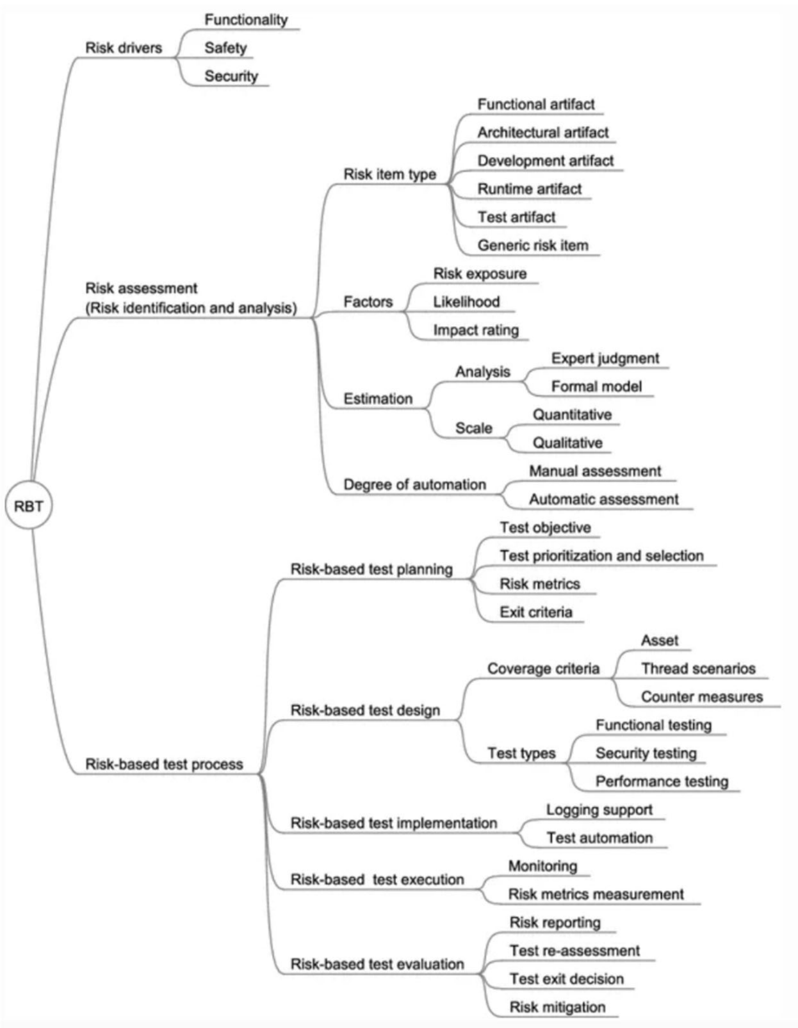

Figure 2: RBT taxonomy

To create order to the practice of risk-based testing, Felderer and Schieferdecker[lvii] proposed a taxonomy for RBT (see Figure 2). The authors also introduce each item in the taxonomy at a detailed level. For brevity, the definitions are omitted here, since to a large degree the discussion is not closely related to IVVES project as such. However, an important observation is that terms “ML” and “AI” are never mentioned in the paper. Hence, IVVES clearly can introduce new elements to RBT with its application of ML and AI in many of the identified taxonomy items. Furthermore, as many of the items are such that considerable amount of data exists in companies, IVVES can also study utilising such data for AI/ML supported RBT.

A systematic literature review (SLR) by Erdogan et al.[lviii] has studied RBT by surveying the literature on the combined use of risk analysis and testing. First, the paper identifies the existing approaches using an SLR. Then, the authors have classified the approaches and discussed with respect to main goal, context of use and maturity level of each approach. The authors found 8 categories:

- Approaches addressing the combination of risk analysis and testing at a general level;
- Approaches with main focus on model-based risk estimation;
- Approaches with main focus on test-case generation;
- Approaches with main focus on test-case analysis;
- Approaches based on automatic source code analysis;
- Approaches targeting specific programming paradigms;
- Approaches targeting specific applications;
- Approaches aiming at measurement in the sense that measurement is the main issue.

Here, topics that have been explicitly mentioned by authors and also are interesting from IVVES perspective include:

- Test prioritization;
- Model-based testing;
- Test case code generation;
- Test case analysis;
- Automatic source code analysis.

To summarize, RBT testing in the large has not yet adopted ML/AI features. To some extend this can be addressed to the origins of the approach, where risk analysis and testing both are to be taken into account and many risk analysis approaches are manual in nature. However, it has been shown that software errors are not randomly distributed in software projects, but that certain parts are more likely to contain bugs than some other parts[lix]. Therefore, methods to identify parts of software projects that are prone to errors – as well as parts that are safe from errors[lx] – are interesting research directions in the scope of project IVVES. In fact, we believe that there are several low-hanging results that do not need ML/AI features at all but can be simply solved by a closer connection between development and testing.

3.2.4 Automatic tests creation

The earliest applications of ML to software testing date back to pioneering work of Budd[lxi] and Weyuker[lxii]. During the 1990s and early 2000s, Inductive Logic Programming (ILP) was considered as a model learning paradigm for model-based test case generation[lxiii] but it has been unclear what range of behaviours can be learned by ILP. Recently, alternative modelling and inference approaches have been considered, such as learning algebraic specifications[lxiv] and learning decision trees[lxv]. Not all such approaches automate the important test oracle step (i.e. test verdict generation), e.g. Briand et al.[lxvi] argues to keep the human in the loop. However, when test suites are large (e.g. > 1 million test cases) it seems clear that automation of the oracle step is also necessary. This is currently being tackled by ML-based methods such as metamorphic testing.

An emerging approach to inference of computational models using ML in recent years is based on active automaton learning (aka. regular inference)[lxvii]. Recently attention has turned to more widely applicable classes of computational models, such as non-deterministic finite automata, timed automata, probabilistic automata, hybrid automata and generalized automata. This ML approach has been applied to software engineering problems such as testing, software documentation, reverse engineering and interface synthesis. In software testing, the approach has been applied to unit testing[lxviii], integration testing[lxix] and system testing[lxx]. This ML approach has also been combined with model checking to perform learning-based testing as well as model-based testing through model inference[lxxi]. Another recent approach to harnessing ML for software testing is given by metamorphic testing[lxxii]. In this approach, graph kernels and support vector machines (SVMs) are used to reverse engineer software testing requirements (aka. metamorphic relations) automatically from code. This represents a significant step towards fully autonomous requirements-based software testing. Metamorphic testing has even been applied to test neural networks themselves. Deep neural networks (DNNs) represent one of the most promising topics in the area of ML. However, the non-explainability in connecting predicted outcomes to learned features makes the DNN model a black box. Usually there is no oracle for testing DNN performance without human intervention. One of the promising methods to mitigate this oracle problem is metamorphic testing by Xie[lxxiii] and Tian[lxxiv]. Here, metamorphic testing operates by checking the system under test (a DNN) against a relation (such as an inequality) between different pairs of DNN predicted outputs. Such a relation is termed a metamorphic relation (MR). The MR specifies how the output would vary, according to changes made to the input. MRs provide a powerful technique to create domain related test cases without any human expert support and could provide a viable option in validating deep learning models.

Model-free ML-assisted tests generation

According to the existing studies in the literature, model-driven techniques or the techniques relying on source code and declarative specifications are common approaches to generate test cases to accomplish the testing objective. However, drawing a precise and well-detailed model which gives the details of the system requires a big endeavor, in particular for complex systems. Moreover, other artifacts such as source code which are also used as underlying tools in many existing techniques, might not be accessible all the time. Therefore, within the scope of black-box testing, there is room for serving ML techniques to generate test cases.

Different types of ML including supervised and unsupervised learning algorithms have been used frequently to tackle different challenges in software testing. In addition to common supervised and unsupervised ML, reinforcement learning (RL)[lxxv] is a fundamental category of learning algorithms which are mainly intended to solve decision-making problems. RL algorithms find the optimal way to make decisions. RL is a different learning paradigm from the supervised and unsupervised learning, which is based on interaction with the environment/system of the problem. There is no supervisor at play in RL and the learning does not occur based on a training data set. Instead, the agent goes through the system and learns the optimal way of decision making through interaction with the system. Basically, at each step of the interaction, the agent observes (senses) the system, takes a possible action to reach the intended objective and receives a reward signal from the environment showing the effectiveness of the applied action to accomplish the intended objective of the agent (See Figure 3 in which the system/environment that the agent interacts with is software under test).

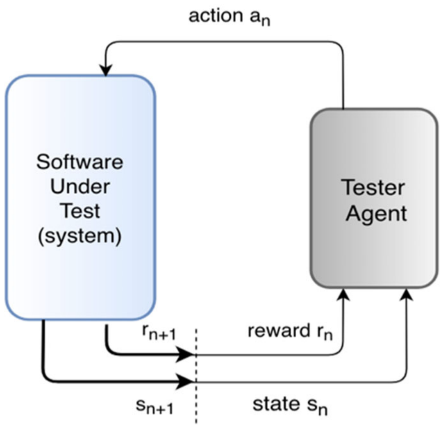

Figure 3. Reinforcement learning cycle between agent (tester) and system (SUT)

Regarding the potential of RL and the issues of common solution techniques, RL techniques in particular model-free RLs could play an interesting role in addressing the challenges of test case generation. Model-free RLs are a subset of RL algorithms which can learn the optimal way to solve a problem (i.e., to accomplish an objective) from the interaction with the system without need to access or build a model of the system. These RL algorithms are not intended to explicitly build or learn a model of the system to understand how it works. The purpose of these algorithms is learning the optimal behaviour, i.e., understanding how to behave to achieve as much reward as possible through multiple experiences of interaction with the system. Monte Carlo learning and Temporal-Difference (TD) learning including Q-learning algorithms are well-known model-free RL algorithms[lxxv].

With respect to the potential of model-free RL to address the related challenges in testing, it is proposed that if the optimal policy (way) for accomplishing the intended objective in the testing could be learned by the tester system instead, then the intended task could be possible without need to access source code or system models. Moreover, once the optimal policy is learned, the learned policy could be reused in further similar testing situations[lxxvi, lxxvii].

The capability of knowledge formation during the learning, storing the gained knowledge and reusing the knowledge in further similar testing situations are the important features in using RL-assisted approaches that could lead to efficiency improvement in comparison to other common approaches such as the ones based on ordinary search techniques[lxxviii].

RL algorithms have been applied to address the testing challenges such as test case generation, particularly in performance testing domain. For example, using RL together with symbolic execution to find the worst-case execution path within a SUT in[lxxix], a feedback-driven learning technique which extracts some rules from the execution traces to find the performance bottlenecks, i.e., the method calls which their execution highly affects the performance[lxxx], using RL to find a sequence of input values resulting in performance degradation[lxxxi], and using RL to build a smart performance testing framework which mainly generates the platform-based test conditions[lxxxii, lxxvi, lxxvii].

3.2.5 Automatic tests selection and prioritization

Test case selection and prioritization automation is fundamental for CI. The objective is to shift the responsibility for testing from human testers and developers to enhanced, ML-enabled tools. This would enable, eventually, that validation techniques and bug-checking are done without user intervention. Complex ES are deriving in test suites to exponentially grow, and the time spent for validation is impacting the CI pipeline.

Test case selection (TCS) and test case prioritization (TCP) ML-enabled techniques are growing rapidly, and there are promising studies and research activities that have been taken into account during the analysis of the state of the art. In fact, some publications have remarked that ML-based TCP surpasses the traditional coverage-based approaches.

The goal of TCS is to provide a subset of the test suite to test a modified model. In parallel, TCP focuses on re-ordering the test in the suite. The tests that in theory are more likely to find a fault or bug are fired firstly. When the potential fault prediction is similar in two candidate tests, other variables (as the fastest, the less performance-demanding…) are taken into account. However, the prioritization criteria can't be easily defined in ML-powered ES.

ML-based TCP and TCS techniques

Regarding the application of ML-based approaches, Busjaeger and Xie[lxxxiii] provided a solution that integrates multiple existing techniques. For this, they used a systematic framework of ML to rank. They applied multiple heuristic techniques: test coverage of modified code, textual similarity between tests and changes, test-failure or fault history and test age. The evaluation on a large dataset indicated that it outperformed previous approaches. Some key outcomes were identified when evaluating the results:

They recall achieved, when selecting 3% of the top tests from the prioritized test suite was close to 75%. As future work, the inclusion of new features for the evaluation where outlined. The approach is focused in real domains and take into account challenges of industrial environments.

Test suite reductions applied to an actual live system, with realistic setup, were also done by Beszédes et al.,[lxxxiv] providing a suite reduction of 51% with over 75% of recall on average. This was much more improved by adding a prioritization step, reaching a reduction over 90% in further tests. The most important tools used in the toolchain for the live version were: Procedure level coverage measurement, Identification of changes made to the source code and Coverage database and database update.

An exhaustive summary is provided by Mulkahainen M.,[lxxxv] where the ML-based techniques remarked where: RandomForest, RandomForest (U), LogReg and XGBoost. When compared with heuristics methods, his study states that ML techniques gradually increase the performance, reaching in some cases better results and that Unlimited RandomForest was the most effective incremental learning-based test case selection. Regarding TCP, five techniques were remarked: RandomForest, MLP, XGBoost, Naïve Bayes and LogReg. The conclusion of the analysis of these techniques, when compared with traditional ones, is that the incremental learning techniques outperformed traditional statement coverage-based prioritization techniques in fault detection rates, when a failing test is assumed to reveal one unique fault.

Reinforcement Learning-based TCS and TCP

Regarding the application of RL to test case prioritization, there are some key concepts to be taken into account. A novel reward function proposed by Wu, Zhaolin, et al.,[lxxxvi] provides a reference for test case prioritization to save computing resources in CI based on RL. In this case, a novel reward function is proposed, by using partial historical information of test cases effectively for fast feedback and cost reduction. The approach is focusing in reduce the huge cost in terms of time and resource availability related to the linear growth observed in both, code committing rate and test suite scales, due to higher complexity and the need of shorter CI cycles. Wu, Zhaolin, et al.[lxxxvi] defined the Average Percentage of Historical Failure with time Window (APHFW), as a novel reinforcement learning reward function, that utilizes a time window to filter recent historical information to calculate reward value.

Spieker et al.[lxxxvii] have successfully applied also RL together with a multi-layered perceptron to predict failing test cases based on test history. They presented RETECS, a novel lightweight method for test case prioritization and selection in CI. RETECS is an adaptive approach, that learns indicators for failing test cases during its runtime by analysing test cases, test results, its own actions and its defects. The evaluation of RETECS in three industrial use cases suggested that a much more effective strategy, compared with basic deterministic prioritization methods could be achieved after an initial learning phase.

3.2.6 Automatic test suite reduction

One of the approaches to timewise optimization of feedback loop from TA pipeline is reduction of the size of the test suite. In order to maintain the suite quality test coverage is an important metric to track while the reduction. It is possible to decrease number of tests in the suite and keep the coverage on the same level if overlapping tests exist. To mitigate risks while further tests reduction more advanced techniques employing code analysis and defect predictions could be used.

To support development teams in this activity, defect prediction has been developed and studied extensively in the last decades[lxxxviii, lxxxix, xc]. Defect prediction identifies code regions that are likely to contain a fault and should therefore be tested[xci, xcii].

Another view on defect prediction is inverse defect prediction (IDP)[xciii]. The idea behind IDP is to identify code artifacts (e.g., methods) that are so trivial that they contain hardly any faults and thus can be deferred or ignored in testing. Like traditional defect prediction, IDP also uses a set of metrics that characterize artifacts, applies transformations to pre-process metrics, and uses a ML classifier to build a prediction model. The difference rather lies in the predicted classes. While defect prediction classifies an artifact either as buggy or non-buggy, IDP identifies methods that exhibit a low fault risk (LFR) with high certainty and does not make an assumption about the remaining methods, for which the fault risk is at least medium or cannot be reliably determined. As a consequence, the objective of the prediction also differs. Defect prediction aims to achieve a high recall, such that as many faults as possible can be detected, and a high precision, such that only few false positives occur. In contrast, IDP aims to achieve high precision to ensure that low-fault-risk methods contain indeed hardly any faults, but it does not necessarily seek to predict all non-faulty methods. Still, IDP needs to achieve a certain recall such that a reasonable reduction potential arises when treating LFR methods with a lower priority in QA activities.

The results of our empirical study[xciii] show that only very few low-fault-risk methods actually contain a fault, and thus, they indicate that IDP can successfully identify methods that are not fault-prone. On average, 31.7% of the methods matched by the strict classifier contain only 6.0% of all faults, resulting in a considerable fault-density reduction for the matched methods. Results show that the IDP approach can be used to identify methods that are, due to the “triviality” of their code, less likely to contain any faults. Hence, these methods require less focus during quality-assurance activities. Depending on the criticality of the system and the risk one is willing to take, the development of tests for these methods can be deferred or even omitted in case of insufficient available test resources.

3.2.7 Automatic root cause analysis

TA pipeline fails requires developers to start investigation to identify root cause. One of the main stages during investigation is debugging of the failed test case and code under the test. Debugging falls into three phases: reproducing a failure, finding the root cause of the failure, and correcting the error such that the failure no longer occurs. While failure reproduction and correction are important issues, it is the second phase, finding the root cause, which is the most significant. Early studies have shown that finding the root cause accounts for 95% of the whole debugging effort[xciv].

There are two reasons why tests can fail:

- External - application environment or infrastructure problems;
- Internal - errors in the code of the application and tests.

External issues

There are number of solutions on the market offering fails root cause analysis focused on application environment problems, for example: NewRelic[xcv], StackSlate[xcvi], Dynatrace[xcvii]. In order to perform that kind of analysis installing sensors on different levels of the infrastructure is required. The sensors then collect different metrics, combine and analyse them presenting the overall picture of systems state. This approach also allows to avoid alert storms deluging developers with cascades of individual alerts.

Internal issues

Speaking of internal problems, it is possible to split it into two levels: unit tests and UI tests level which is sometimes called end-to-end (E2E) tests level.

Unit tests level

One of the approaches to identify root cause on unit test level in the code is delta debugging— an automated debugging method that relies on systematic testing to prove and isolate failure causes—circumstances such as the program input, changes to the program code, or executed statements. Basically, delta debugging sets up subsets of the original circumstances, and tests these configurations whether the failure still occurs. Eventually, delta debugging returns a subset of circumstances where every single circumstance is relevant for producing the failure.

Delta debugging automates the most time-consuming debugging issue: determining the relevant problem circumstances. Relevant circumstances include the program input, changes to the program code, or executed statements. All that is required is an automated test.

Delta debugging comes at a price: Although the minimizing delta debugging algorithm (ddmin) algorithm guarantees 1-minimality, the worst-case quadratic complexity is a severe penalty for real-world programs— especially considering program runs with billions of executed statements[xcviii],[xcix].

GUI/E2E tests level

E2E tests, which include interaction with GUI, relies on usage of controls like buttons, input fields and others. ES GUIs, especially Web GUIs, are subjects for frequent changes and thus tests should be always kept aligned with recent changes. It is not always the case. That is why number of solutions appear on the market helping developers and quality assurance engineers in root cause analysis: AppliTools[c], Functionize Visual Testing[ci]

3.2.8 Automatic tests healing

Two categories of test failings could be identified: random failings and failings caused by errors in the environment or in the code.

Flaky tests

Tests which could fail or pass from one test run to another for the same configuration are called “flaky” tests. Such behaviour could be harmful to developers because test failures do not always indicate bugs in the code. Our test suite should act like a bug detector. Non-determinism can plague any kind of test, but it’s particularly prone to affect tests with a broad scope, such as acceptance, functional/UI tests. Some common reasons a test could be flaky:

- Concurrency;
- Caching;
- Tests setup—Cleanup state;
- Dynamic UI contents;
- Infrastructure or 3rd party systems issues.

In order to identify those tests basic statistical methods could be applied. Also, supervised classification ML models could be used. After the identification of those kind of tests they should be subjected for refactoring, while, in a mean time, separate routine could be introduced to rerun failed flaky tests and save developers time on investigation of those cases.

Failing tests

It was already highlighted in the “Automatic root cause analysis” that two levels could be considered independently while speaking of failed tests: unit tests and GUI/E2E tests level.

Unit tests level

The cost of debugging and maintaining software has continued to rise, even while hardware and many software costs fall. In 2006, one Mozilla developer noted, “everyday, almost 300 bugs appear [...] far too much for only the Mozilla programmers to handle”[cii]. The situation has hardly improved in the intervening years, as bugzilla.mozilla.org indicates similar rates of bugs reported in 2013. A 2013 study estimated the global cost of debugging at $312 billion, with software developers spending half their time debugging[ciii]. Since there are not enough developer resources to repair all of these defects before deployment, it is well known that programs ship with both known and unknown bugs[civ].

In response to this problem, many companies offer bug bounties that pay outside developers for candidate repairs to their open source code. Well-known companies such as Mozilla ($3,000/bug)[cv], Google ($500/bug)[cvi], and Microsoft ($10,000/bug)[cvii], offer significant rewards for security fixes, reaching thousands of dollars and engaging in bidding wars[cviii]. While many bug bounties simply ask for defect reports, other companies, such as Microsoft, reward defensive ideas and patches as well (up to $50,000/fix)[cix].

The abundance and success of these programs suggests that the need for repairs is so pressing that some companies must consider outside, untrusted sources, even though such reports must be manually reviewed, most are rejected, and most accepted repairs are for low-priority bugs[cx]. A technique for automatically generating patches, even if those patches require human evaluation before deployment, could fit well into this paradigm, with potential to greatly reduce the development time and costs of software debugging.

The importance of defects in software engineering practice is reflected in software engineering research. Since 2009, when automated program repair was demonstrated on real-world problems (PACHIKA[cxi], ClearView[cxii], GenProg[cxiii]), interest in the field has grown steadily, with multiple novel techniques proposed (e.g., Debroy and Wong[cxiv], AutoFix-E[cxv], ARMOR[cxvi, cxvii], AFix[cxviii], AE[cxix], Coker and Hafiz[cxx], PAR[cxxi], SemFix[cxxii], TrpAutoRepair[cxxiii], Monperrus[cxxiv], Gopinath et al.[cxxv], MintHint[cxxvi], etc.). Some of these methods produce multiple candidate repairs, and then validate them using test cases, such as by using stochastic search or methods based on search-based software engineering[cxxvii] (e.g., GenProg, PAR, AutoFix-E, ClearView, Debroy and Wong, TrpAutoRepair). Others use techniques such as synthesis or constraint solving to produce smaller numbers of patches that are correct by construction (e.g., Gopinath et al., AFix, etc.) relative to inferred or human-provided contracts or specifications.

Several recent studies have established the potential of these techniques to reduce costs and improve software quality, while raising new questions about the acceptability of automatically generated patches to humans. See, for example, the systematic study of GenProg, which measured cost in actual dollars[cxxviii] and related studies that assess the acceptability of automatically generated patches[cxxi, cxxix].

An attempt was made to build a general benchmark for assessing the quality automatically generated patches and two datasets were presented, MANYBUGS and INTROCLASS, consisting between them of 1,183 defects in 15 C programs. Each dataset is designed to support the comparative evaluation of automatic repair algorithms asking a variety of experimental questions. The datasets have empirically defined guarantees of reproducibility and benchmark quality, and each study object is categorized to facilitate qualitative evaluation and comparisons by category of bug or program. Baseline experimental results were presented in the Table 3 and 4 on both datasets for three existing repair methods, GenProg, AE, and TrpAutoRepair, to reduce the burden on researchers who adopt these datasets for their own comparative evaluations[cxxx]. The average number of test suite executions in runs leading to a repair is presented as “fitness evaluations” in the figures. This measurement serves as a compute- and scenario-independent measure of efficiency, which is typically dominated by test suite execution time.

Table 3: MANYBUGS: Baseline results of running GenProg v2.2, TrpAutoRepair, and AE v3.0 on the 185 defects of the MANYBUGS benchmark. For each of the repair techniques, we report the number of defects repaired per program; the average time to repair in minutes (GenProg and TrpAutoRepair were run on 10 seeds per scenario, with each run provided a 12-hour timeout; AE is run once per scenario, with a 60-hour timeout); and the number of fitness evaluations to a repair, which serves as a compute- and scenario-independent measure of repair time (typically dominated by test suite execution time and thus varies by test suite size). Complete results, including individual log files for each defect, are available for download with the dataset.

Program	GenProg			TrpAutoRepair			AE		
	Defects repaired	Time (min)	Fitness evals	Defects repaired	Time (min)	Fitness evals	Defects repaired	Time (min)	Fitness evals
fbc	1/3	133	79.0	0/3	-	-	1/3	7	1.7
gmp	1/2	13	7.2	1/2	18	2.4	1/2	739	63.3

gzip	1/5	240	130.7	1/5	107	56.7	2/5	84	1432.0
libtiff	17/24	27	20.8	17/24	16	2.9	17/24	24	3.0
lighttpd	5/9	79	44.1	4/9	33	14.9	4/9	22	11.2
php	54/104	181	5.2	56/104	180	1.1	53/104	441	1.1
python	2/15	110	12.9	2/15	144	1.4	3/15	529	7.6
valgrind	4/15	193	24.0	4/15	133	1.5	0/15	-	-
wireshark	5/8	140	14.3	5/8	44	2.6	5/8	574	66.5

Table 4: INTROCLASS: Baseline results of running GenProg v2.2, TrpAutoRepair, and AE v3.0 on the 845 white-box-based defects, and 778 white-boxbased defects of the INTROCLASS benchmark. For each of the repair techniques, we report the number of defects repaired per program; the average time to repair in second (all three techniques were given timeouts); and the number of fitness evaluations needed to produce a repair. Complete results, including individual log files for each defect, are available for download with the dataset.

Program	GenProg			TrpAutoRepair			AE		
	Defects repaired	Time (min)	Fitness evals	Defects repaired	Time (min)	Fitness evals	Defects repaired	Time (min)	Fitness evals
White-box-based defects									
checksum	3/49	343	132	1/49	10	5	1/49	4	1
digits	99/172	191	102	46/172	32	13	50/172	11	3
grade	3/224	152	160	2/224	26	23	2/224	25	25
median	63/152	107	114	26/152	19	25	16/152	4	2
smallest	118/118	23	23	118/118	15	11	92/118	4	2
syllables	6/130	284	157	9/130	36	56	5/130	9	6
Black-box-based defects									
checksum	8/29	517	307	0/29	-	-	0/29	-	-
digits	30/91	162	77	19/91	24	15	17/91	6	6
grade	2/226	141	156	2/226	30	27	2/226	24	25
median	108/168	44	59	93/168	20	20	58/168	4	1
smallest	120/155	102	86	119/155	24	21	71/155	5	4
syllables	19/109	96	117	14/109	39	54	11/109	3	2

Also, a collection of reproducible bugs and a supporting infrastructure with the goal of advancing software engineering research was created[cxxxi] together with data and scripts that extend the ManyBugs version beta-2.1 and Defects4J version 1.1.0 benchmarks to enable the evaluation of automated program repair's applicability to defects, For example, these data enable evaluating if automated repair techniques are able to produce patches for defects considered hard or important by developers[cxxxii].

GUI/E2E tests level

For E2E tests, which include interaction with GUI automatic healing often converges to identification of the right element for interaction or assessment (visibility, text checking etc). For that purposes smart runners exists. One of the examples of that smart runner service is Functionize. It is declared[cxxxiii] that Functionize platform has abilities to:

- Identify changes in the test execution comparing to previous test runs;
- Suggest a solution to fix the failing test;
- Automatically validate the suggestion.

3.3 Operation

At this point, the development cycle is almost finished. The application is done and being used in the field. The Operation phase is still important, though. In this phase, users discover bugs that weren't found during testing. These errors need to be resolved, which can spawn new development cycles.

In addition to bug fixes, models like Iterative development plan additional features in future releases. For each new release, a new Development Cycle can be launched.

3.3.1 Analytics and monitoring

The state of the industry today requires fast deployment cycles and continuous testing for a company to keep up. This means that time to market needs to be optimized without damaging the quality of software or model, as users expect the software to be updated and enhanced quickly[cxxxiv].

To optimize the deployment cycle, it is necessary to test in a smart and planned way. Analytics and monitoring can help with that, giving insight into the process. It has been recorded that using project-level analytics has improved productivity by 28%[cxxxv], by offering solution to the problem of determining how to reduce the scope of effort, making development smarter and more efficient. This is done by doing analysis through the entire development cycle, starting from database structure and ending in user experience. The points of analysis are defined through use of Function Points[cxxxvi], units of measure that express business functionality provided to the user by an information system. In order to quantify business functionality, user requirements are considered, to be concrete, the output of a system, inquiries, inputs, internal files and external interfaces. These requirements are then assigned a specific number of function points. An automated approach to assigning these points has been standardized through Automated Function Points (AFPs) ISO Standards, which include, among others: FiSMA[cxxxvii], IFPUG[cxxxviii], Nesma[cxxxix]. These standards are mostly user oriented and none of them include algorithmic complexity. FiSMA has tried to combat this by using engineering function points (operators and Booleans are counted) and weighted micro function points (newer model that adjusts function points based on complexity[cxl]).

Analytics insights are presented inside the company or project through monitoring dashboards. It is important to align all analysis results in an efficient and understandable way. Microsoft Power BI[cxli] can be utilized for this, creating dashboards with heat maps, bar plots and similar. Some examples of possible dashboards used in projects are:

- **Productivity analysis dashboard** can be visualised using AFPs measuring the size and effort in maintaining software through story points (estimation of story points in agile way of work can give insight into the effort put into software maintenance[cxlii], lines of code or functional size (software metric used to measure the effort needed to maintain software by counting the number of lines in source code or looking at the functional size of it), code review defects, code coverage and many more;
- **Structural quality dashboard** can also be visualised using AFPs, to measure the impact of DevOps transformation practices. Metrics used are defect ratio, dollar spend, cycle release time, build count, and other. Analysing structural quality is independent of programming language used and focuses on integration of building blocks and overall structural integrity of software in each project;
- **User analytics dashboard**, constructed by collecting user feedback when handling software, can be very useful in driving focus and effort.

These dashboards can help higher management gain an overview into efforts put in the development cycle, the quality of software that has been developed and user feedback.

3.3.2 Real usage-based testing

Real usage of software can be regarded as a form of usage-based testing, at least under certain conditions. In the clearest case, if defects are detected by clients, some information about them is reported to software vendors, and integrated fixes may be created and delivered to all the clients to avert such defects. The situation is less clear when analytics and monitoring discussed above are used to detect possible errors. However, if errors are found only after deploying the software to end users, there in any case needs to be updates, which can be annoying and costly for the end users.

However, it is also possible to mimic the behaviour of end users. This technique is commonly referred to as usage-based statistical testing (UBST)[cxliii]. UBST is considered means to cost-effectively improve the quality of software delivered into systems integration was a driving criterion for the program. UBST provides the capability to increase the number of test cases executed on the software and to focus the testing on expected usage scenarios[cxliv]. The techniques provide quantitative methods for measuring and reporting testing progress, and support managing the testing process. Hence such data can also be applied in the scope of IVVES.

In the technical sense, in UBST, the testing environment resembles the actual operational environment for the software in the field. Furthermore, the overall testing sequence is similar to real-life usage scenarios, sequences, and templates of actual software usage by the target clients. As the huge quantity of clients and diverse usage templates cannot be captured in an implementation set of test cases, statistical sampling is required. Obviously, there is a link to monitoring and analysis capabilities, as they provide important input for designing for UBST. This has inspired researchers and practitioners to use the approach in the context of web applications in particular (e.g. cxlv, cxlvi), where tracing user actions is often easier than when dealing with installable software. However, also synthetic data can be used to support the approach[cxlvii].

Usage-based statistical testing is commonly appropriate to the final phase of software testing. It can be also used as a part of acceptance testing right before product release, in which case stopping testing is of equal worth to the product release. While less common, it is also possible to apply UBST to integration and system testing, if data and knowledge of actual client usage situations is available. This can support reaching effectual reliability goals before product release.

3.4 Summary

A concise summary of methods and techniques in different phases of the continuous quality assurance process are presented in Table 5. Where:

- Test level are:
  - Req. – requirements;
  - Unit – unit tests;
  - Int. – integration tests;
  - E2E – end-to-end tests.
- States are:
  - P – state of the practice;
  - A – state of the art.
- Adoption levels:
  - Low – technique or approach is developed during research project and no or only very few companies using it;
  - Medium – software implementing the method is available and used by some companies;
  - High – different tools implementing the same approach is available for different technology stacks and widely used by companies. De facto being the state of the practice.

Table 5: Stages, techniques and tools summary

Technique	Tests level	State	Adoption level	Tools (if available)
Design				
New features				
Modelling				
Threat Modelling	Req.	P	High	- STRIDE - P.A.S.T.A. - Trike - VAST
TLA	Req.	A	Low	- TLA toolbox
Improvements				
Automatic bugs triaging	Req., E2E	A	Medium	- CERT Triage tool / Exploitable
Development and Testing				
Development				
Static code analysis	Unit	P	High	- SonarQube - Language specific IDEs, linters and analysis tools
Code anomaly detection	Unit	A	Low	- REPD
Formal Verification	Req.	P	Medium	- Uppaal - PRISM - Rebeca (Afra)
Risk-based testing	Unit, Int., E2E	P	Medium	
Tests creation				
Automatic tests creation				
Fuzzing	Int., E2E	A	Medium	- LibFuzzer etc - American Fuzzy Loop - AddressSanitizer, ThreadSanitizer, MemorySanitizer - OssFuzz
Metamorphic testing	Unit, Int., E2E	A	Low	
Search-based testing	Unit, Int., E2E	A	Medium	- EvoSuite - Randoop - Microsoft IntelliTest - DiffBlue Cover
Model-based testing	E2E	A	Medium	- Test Modeller

				- APOGEN with Crawljax - ALEX
ML-based testing (model free reinforcement learning)	Unit, E2E	A	Low	- RELOAD - SaFReL
Tests maintenance				
Automatic test selection and prioritization	Unit, Int., E2E	A	Low	- TestArchiver and ChangeEngine by SALabs
Automatic root cause analysis	Unit, Int., E2E	A	Low	- Functionize platform - Delta debugging tools
Automatic test suite reduction	Unit, Int., E2E	A	Low	
Automatic healing	Unit, Int., E2E	A	Low	- Functionize platform
Operation				
Analytics and monitoring	E2E	P	High	- AWS CloudWatch - New Relic - Kibana - Google Analytics - Matomo
Real usage-based testing	E2E, Int.	A	Low	

4. Conclusions

This report presents the state of the art of validation methods and techniques for complex ES. The main contributions are:

- A mapping of the validation methods and techniques with the continuous quality assurance process
- A concise summary of methods and techniques in different phases of the continuous quality process
- A classification of validation methods and tools by test and adoption levels

The main conclusions are:

- The huge gap between academic researches and industry state of the practice and art exists;
- Often academic research results:
  - has limited application;
  - requires strong expert knowledge, skills and considerable effort to be applied in the industry.
- Companies developing mission critical systems can afford applying expensive state of the art techniques for their validation and verification.

The findings of this report suggest applying research results to produce tools that could be applied with reasonable effort by avoiding too expensive for implementation and maintenance methods and limiting the scope of addressed problem.

Three sub-domains could be considered as main focus areas for the project next steps:

1. Model-based test generation with automatic model building:
   - as it can provide companies with high level end-to-end regression testing suites and requires only basic knowledge and skillset from engineers
   - some tools are already publicly available, but applicability of those tools is unclear;
2. ML-assisted test generation: tester (testing system) is intelligent and learns the optimal policy (way) to generate the test cases meeting the testing objective:
   - as it can provide automated test generation without access to source code or system model
   - in some cases, it is able to reuse the gained knowledge (learned policy) in further similar testing situations (transfer learning);
3. Automatic test selection and prioritization as it, when applied, reduces TA infrastructure costs and feedback time allowing teams work in the most efficient manner.

5. References

[i] Leslie Lamport. Thinking for Programmers (Technical talk). San Francisco: Microsoft.
[ii] Cousineau, Denis; Doligez, Damien; Lamport, Leslie; Merz, Stephan; Ricketts, Daniel; Vanzetto, Hernán. TLA+ Proofs (PDF). FM 2012: Formal Methods. Lecture Notes in Computer Science. 7436. Springer Berlin Heidelberg. pp. 147–154. doi:10.1007/978-3-642-32759-9_14. ISBN 978-3-642-32758-2.
[iii] Newcombe, Chris; Rath, Tim; Zhang, Fan; Munteanu, Bogdan; Brooker, Marc; Deardeuff, Michael (29 September 2014). "Use of Formal Methods at Amazon Web Services" (PDF). Amazon.
[iv] Chris, Newcombe (2014). Why Amazon Chose TLA+. Lecture Notes in Computer Science. 8477. Springer Berlin Heidelberg. pp. 25–39. doi:10.1007/978-3-662-43652-3_3. ISBN 978-3-662-43651-6.
[v] Lardinois, Frederic. "With Cosmos DB, Microsoft wants to build one database to rule them all". TechCrunch.
[vi] Leslie Lamport. Foundations of Azure Cosmos DB with Dr. Leslie Lamport (Recording of interview). Microsoft Azure.
[vii] Ali Sajedi Badashian, Abram Hindle, and Eleni Stroulia. Crowdsourced bug triaging. In International Conference on Software Maintenance and Evolution, pages 506–510. IEEE, 2015.
[viii] Pamela Bhattacharya and Iulian Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In International Conference on Software Maintenance, pages 1–10, 2010.
[ix] John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Transactions on Software Engineering and Methodology, 20(3):10, 2011.
[x] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. Empirical Software Engineering, 21(4):1533–1578, 2016.
[xi] Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In Working Conference on Mining Software Repositories, pages 2–11, 2013.
[xii] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Fuzzy set-based automatic bug triaging: Nier track. In International Conference on Software Engineering, pages 884–887, 2011.
[xiii] Song Wang, Wen Zhang, and Qing Wang. Fixercache: unsupervised caching active developers for diverse bug triage. In ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, page 25, 2014.
[xiv] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. Towards effective bug triage with software data reduction techniques. IEEE Transactions on Knowledge and Data Engineering, 27(1):264–280, 2015.
[xv] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. Developer prioritization in bug repositories. In International Conference on Software Engineering, pages 25–35, 2012.
[xvi] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In International Conference on Software Engineering, pages 837–847, 2012.
[xvii] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013.
[xviii] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In International Conference on Software Engineering, pages 404–415, 2016.
[xix] Quoc V Le and Tomas Mikolov. Distributed representations of sentences and documents. In International Conference on Machine Learning, volume 14, pages 1188–1196, 2014.
[xx] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In International Conference on Automated Software Engineering, pages 476–481, 2015.
[xxi] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. Toward deep learning software repositories. In Working Conference on Mining Software Repositories, pages 334–345, 2015.
[xxii] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In International Conference on Frontiers in Handwriting Recognition, pages 285–290, 2014.

[xxiii] Senthil Mani, Anush Sankaran, Rahul Aralikatte. DeepTriage: Exploring the Effectiveness of Deep

Learning for Bug Triaging. The ACM India Joint International Conference on. Data Science & Management of Data. CoDS-COMAD '19, 2019

[xxiv] Susan Li, Anomaly Detection for Dummies, 2019, https://towardsdatascience.com/anomaly-detection-for-dummies-15f148e559c1

[xxv] Afric, Petar & Sikic, Lucija & Kurdija, Adrian & Silic, Marin. (2020). REPD: Source Code Defect Prediction as Anomaly Detection. Journal of Systems and Software. 10.1016/j.jss.2020.110641.

[xxvi] Timofey Bryksin, Victor Petukhov, Ilya Alexin, Stanislav Prikhodko, Alexey Shpilman, Vladimir Kovalenko, and Nikita Povarov. 2020. Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler. In17thInternational Conference on Mining Software Repositories (MSR '20), October5–6, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.338744

[xxvii] Alind Gupta, Machine Learning for Anomaly Detection, https://www.geeksforgeeks.org/machine-learning-for-anomaly-detection/

[xxviii] D'Ambros, M., Lanza, M. & Robbes, R. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empir Software Eng 17, 531–577 (2012). https://doi.org/10.1007/s10664-011-9173-9

[xxix] E. Giger, M. D'Ambros, M. Pinzger and H. C. Gall, "Method-level bug prediction," Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Lund, 2012, pp. 171-180, doi: 10.1145/2372251.2372285.

[xxx] M. Yan, Y. Fang, D. Lo, X. Xia and X. Zhang, "File-Level Defect Prediction: Unsupervised vs. Supervised Models," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, 2017, pp. 344-353, doi: 10.1109/ESEM.2017.48.

[xxxi] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, andTien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (Amsterdam, The Netherlands) (ESEC/FSE '09). ACM, New York, NY, USA, 383–392. https://doi.org/10.1145/1595696.1595767

[xxxii] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (Dubrovnik, Croatia) (ESEC-FSE '07). ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1287624.1287632

[xxxiii] Sudheendra Hangal and Monica S. Lam. 2002. Tracking Down Software Bugs Using Automatic Anomaly Detection. In Proceedings of the 24th International Conference on Software Engineering (Orlando, Florida) (ICSE '02). ACM, New York, NY, USA, 291–301. https://doi.org/10.1145/581339.581377

[xxxiv] Willian N. Oizumi, Alessandro F. Garcia, Thelma E. Colanzi, Manuele Ferreira, and Arndt V. Staa. 2015. On the relationship of code-anomaly agglomerations and architectural problems. Journal of Software Engineering Research and Development 3, 1 (10 Jul 2015), 11. https://doi.org/10.1186/s40411-015-0025-y

[xxxv] A. E. Hassan, "Predicting faults using the complexity of code changes," in 2009 IEEE 31st International Conference on Software Engineering, May 2009, pp. 78–88.

[xxxvi] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Transactions on Software Engineering, vol. 26, no. 7, pp. 653–661, July 2000.

[xxxvii] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov 2013, pp. 279–289.

[xxxviii] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276–1304, Nov 2012.

[xxxix] T. M. Khoshgoftaar and N. Seliya, "Software quality classification modelling using the sprint decision tree algorithm," in 14th IEEE International Conference on Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings., Nov 2002, pp. 365–374.

[xl] T. M. Khoshgoftaar, X. Yuan, and E. B. Allen, "Balancing misclassification rates in classification-tree models of software quality," Empirical Software Engineering, vol. 5, no. 4, pp. 313–330, Dec 2000. [Online]. Available: https://doi.org/10.1023/A:1009896203228

[xli] NASA Langley "Formal Methods". url: http://shemesh.larc.nasa.gov/fm/fm-what.html

[xlii] C. Baier and J.-P. Katoen. "Principles of Model Checking" (Representation and Mind Series). The MIT Press, 2008.

[xliii] Beyer D., Lemberger T. (2017) "Software Verification: Testing vs. Model Checking". In: Strichman O., Tzoref-Brill R. (eds) Hardware and Software: Verification and Testing. HVC 2017. Lecture Notes in Computer Science, vol 10629. Springer, Cham

[xliv] Cordeiro, Lucas & Filho, Eddie & Bessa, Iury. (2019). "A Survey on Automated Symbolic Verification and its Application for Synthesizing Cyber-Physical Systems". IET Cyber-Physical Systems: Theory & Applications. 10.1049/iet-cps.2018.5006.

[xlv] Marwa Hachicha, Riadh Ben Halima, Ahmed Hadj Kacem, "Formal Verification approaches of Self-adaptive Systems: A Survey", Procedia Computer Science, Volume 159, 2019, Pages 1853-1862, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2019.09.357.

[xlvi] Amrani, Moussa, Levi Lucio and Adrien Bibal. “ML + FV = ♡? A Survey on the Application of Machine Learning to Formal Verification.” ArXiv abs/1806.03600 (2018)

[xlvii] Alam, M. M., & Khan, A. I. (2013). Risk-based testing techniques: a perspective study. International Journal of Computer Applications, 65(1).

[xlviii] Bai, X., Kenett, R.S., Yu, W.: Risk assessment and adaptive group testing of semantic web services. Int. J. Softw. Eng. Knowl. Eng. 22(05), 595–620 (2012)

[xlix] Casado, R., Tuya, J., Younas, M.: Testing long-lived web services transactions using a risk-based approach. In: 10th international conference on quality software. pp. 337–340. IEEE (2010)

[l] Felderer, M., Haisjackl, C., Breu, R., Motz, J.: Integrating manual and automatic risk assessment for risk-based testing, pp. 159–180. Software quality. Process automation in software, development (2012)

[li] Amland, S.: Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. J. Syst. Softw. 53(3), 287–295 (2000)

[lii] Bach, J.: Heuristic risk-based testing. Softw. Test. Qual. Eng. Mag. 11, 99 (1999)

[liii] Redmill, F.: Exploring risk-based testing and its implications. Softw. Test. Verif. Reliab. 14(1), 3–15 (2004)

[liv] Rosenberg, L., Stapko, R., Gallo, A.: Risk-based object oriented testing. Proceedings of 13th international software/internet quality week-QW 2 (2000)

[lv] van Veenendaal, E.: Practical risk-based testing—The PRISMA Approach. UTN Publishers (2012)

[lvi] ISO: ISO/IEC/IEEE 29119 Software Testing. http://www.softwaretestingstandard.org/ (2013). Accessed 16 June 2020.

[lvii] Felderer, M., & Schieferdecker, I. (2014). A taxonomy of risk-based testing. International Journal on Software Tools for Technology Transfer, Vol. 16, pages 559–568 (2014)

[lviii] Erdogan, G., Li, Y., Runde, R. K., Seehusen, F., & Stolen, K. (2014). Approaches for the combined use of risk analysis and testing: a systematic literature review. International Journal on Software Tools for Technology Transfer, 16(5), 627-642.

[lix] Concas, G., Marchesi, M., Murgia, A., Tonelli, R., & Turnu, I. (2011). On the distribution of bugs in the eclipse system. IEEE Transactions on Software Engineering, 37(6), 872-877.

[lx] Niedermayr, R., Röhm, T., & Wagner, S. (2019). Too trivial to test? An inverse view on defect prediction to identify methods with low fault risk. PeerJ Computer Science, 5, e187.

[lxi] T. A. Budd, D. Angluin: Two notions of correctness and their relation to testing. Acta Informatica 18(1), pp. 31–45, 1982.

[lxii] E. J. Weyuker: Assessing test data adequacy through program inference. ACM Transactions on Programming Languages and Systems (TOPLAS) 5(4), pp. 641–655, 1983.

[lxiii] King RD, Whelan KE, Jones FM, Reiser PG, Bryant CH, Muggleton SH, Kell DB, Oliver SG: Functional genomic hypothesis generation and experimentation by a robot scientist. Nature 427(6971): pp. 247–252, 2004.

[lxiv] J. Henkel, A. Diwan: Discovering algebraic specifications from java classes. In: European Conference on Object-Oriented Programming, Springer, pp. 431–456, 2003.

[lxv] P. Papadopoulos, N. Walkinshaw: Black-box test generation from inferred models. In: Proceedings of the Fourth International Work- shop on Realizing Artificial Intelligence Synergies in Software Engineering, IEEE Press, pp. 19–24, 2015

[lxvi] L. C. Briand, Y. Labiche, Z. Bawar, N. T. Spido: Using machine learning to refine category-partition test specifications and test suites. Information and Software Technology 51(11): pp. 1551–1564, 2009.

[lxvii] A. Bennaceur, K. Meinke: Machine Learning for Software Analysis: Models, Methods, and Applications, pp 3-49 in: Machine Learning for Dynamic Software Analysis, Lecture Notes in Computer Science 11026, Springer 2018.

[lxviii] H. Khosrowjerdi, K. Meinke, A. Rasmusson: Learning-Based Testing for Safety Critical Automotive Applications, pp 197-211 in: Lecture Notes in Computer Science, 10437, Springer, 2017.

[lxix] R. Groz, K. Li, A. Petrenko: Integration testing of communicating systems with unknown components, Annales des Telecommunications, 70, (3-4), pp. 107-125, 2015.

[lxx] K. Meinke: Learning-Based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study, Lecture Notes in Computer Science, 10497, pp 135-151, Springer, 2017.

[lxxi] A. Bainczyk, A. Schieweck, B. Steffen, F. Howar: Model-Based Testing Without Models: The TodoMVC Case Study, Lecture Notes in Computer Science, 10500, pp. 125-144, Springer, 2017
[lxxii] U. Kanewala, J. M Bieman, A. Ben-Hur: Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. In: Software testing, verification and reliability 26.3, pp. 245–269, 2016.
[lxxiii] X. Xie et al.: Testing and validating machine learning classifiers by metamorphic testing. In: Journal of Systems and Software 84.4, pp. 544–558, 2011
[lxxiv] Y. Tian et al.: DeepTest: Automated testing of deep-neural- network-driven autonomous cars. In: arXiv preprint arXiv:1708.08559, 2017.
[lxxv] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.
[lxxvi] 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, 2019, pp. 1187–1189.
[lxxvii] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "Poster: Performance testing driven by reinforcement learning," in Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2020.
[lxxviii] M. Helali Moghadam, "Machine Learning-Assisted Performance Assurance" (Licentiate dissertation). Mälardalen University, Västerås. 2020 Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-47501
[lxxix] Koo, C. Saumya, M. Kulkarni, and S. Bagchi, "Pyse: Automatic worst-case test generation by reinforcement learning," in2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). IEEE, 2019, pp. 136–147.
[lxxx] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in2012 34thInternational Conference on Software Engineering (ICSE). IEEE, 2012, pp. 156–166.
[lxxxi] T. Ahmad, A. Ashraf, D. Truscan, and I. Porres, "Exploratory performance testing using reinforcement learning," in2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2019, pp. 156–163.
[lxxxii] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "Machine learning to guide performance testing: An autonomous test framework," in2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2019, pp. 164–167.
[lxxxiii] B. Busjaeger, T. Xie, Learning for test prioritization: An industrial case study, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 2016, ACM, FSE 2016, Seattle, WA, USA, pp. 975–980.
[lxxxiv] BESZÉDES, Árpád, et al. Code coverage-based regression test selection and prioritization in WebKit. En 2012 28th IEEE international conference on software maintenance (ICSM). IEEE, 2012. p. 46-55.
[lxxxv] MULKAHAINEN, Markus. Test case selection and prioritization in continuous integration environment. 2019.
[lxxxvi] WU, Zhaolin, et al. A Time Window based Reinforcement Learning Reward for Test Case Prioritization in Continuous Integration. En Proceedings of the 11th Asia-Pacific Symposium on Internetware. 2019. p. 1-6.
[lxxxvii] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. ISSTA 2017: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, July 2017 Pages 12–22 https://doi.org/10.1145/3092703.3092709
[lxxxviii] Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering (TSE), 38(6):1276–1304.
[lxxxix] D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: A benchmark and an extensive comparison. Empirical Software Engineering, 17(4-5):531–577.
[xc] Catal, C. (2011). Software fault prediction: A literature review and current trends. Expert Systems with Applications, 38(4):4626–4636.
[xci] Menzies, T., Greenwald, J., and Frank, A. (2007). Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering (TSE), 33(1):2–13.
[xcii] Weyuker, E. J. and Ostrand, T. J. (2008). What can fault prediction do for you? In Proc. 2nd International 683 Conference on Tests and Proofs (TAP'08), pages 1–10. Springer.
[xciii] Rainer Niedermayr, Tobias Röhm, Stefan Wagner. Too Trivial To Test? An Inverse View on Defect Prediction to Identify Methods with Low Fault Risk. PeerJ Computer Science 5:e187, 2019

[xciv] G. J. Myers. The Art of Software Testing. John Wiley & Sons, Inc., New York, 1979.
[xcv] (2020, Jun.) [Online]. Available: https://newrelic.com/products/application-monitoring
[xcvi] (2020, Jun.) [Online]. Available: https://www.stackstate.com/product/root-cause-analysis/
[xcvii] (2020, Jun.) [Online]. Available: https://www.dynatrace.com/platform/root-cause-analysis/
[xcviii] Holger Cleve, Andreas Zeller. Finding Failure Causes through Automated Testing. Proc. Fourth International Workshop on Automated Debugging, Munich, Germany, 28-30 August 2000.
[xcix] Alessandro Orso, Shrinivas Joshi, Martin Burger, Andreas Zeller. Isolating relevant component interactions with JINSI. WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis, May 2006 Pages 3–10 https://doi.org/10.1145/1138912.1138915
[c] (2020, Jun.) [Online]. Available: https://applitools.com/root-cause-analysis/
[ci] (2020, Jun.) [Online]. Available: https://www.functionize.com/visual-testing/
[cii] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in Proc. Int. Conf. Softw. Eng., 2006, pp. 361–370.
[ciii] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible debugging software,” Judge Bus. School, Univ. Cambridge, Cambridge, U.K., Tech. Rep., 2013.
[civ] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” in Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation, 2005, pp. 15–26.
[cv] (2014, Feb.) [Online]. Available: http://www.mozilla.org/security/bug-bounty.html
[cvi] (2014, Feb.) [Online]. Available: http://blog.chromium.org/2010/01/encouraging-more-chromium-security.html
[cvii] (2014, Feb.) [Online]. Available: http://msdn.microsoft.com/enus/library/dn425036.aspx
[cviii] C. World. (2014, Feb.) [Online]. Available: http://www.computerworld.com/s/article/9179538/Google_calls_raises_Mozilla_s_bug_bounty_for_Chrome_flaws
[cix] (2014, Feb.) [Online]. Available: http://msdn.microsoft.com/enus/library/dn425036.aspx
[cx] (2014, Feb.) [Online]. Available: http://www.daemonology.net/blog/2011-08-26-1265-dollars-of-tarsnap-bugs.html
[cxi] V. Dallmeier, A. Zeller, and B. Meyer, “Generating fixes from object behavior anomalies,” in Proc. IEEE/ACM Int. Conf. Automated Softw. Eng., 2009, pp. 550–554.
[cxii] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, “Automatically patching errors in deployed software,” in Proc. ACM Symp. Operating Syst. Principles, Big Sky, MT, USA, Oct. 12–14, 2009, pp. 87–102.
[cxiii] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in Proc. Int. Conf. Softw. Eng., 2009, pp. 364–367.
[cxiv] V. Debroy and W. E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” in Proc. Int. Conf. Softw. Testing, Verification, Validation, 2010, pp. 65–74.
[cxv] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” in Proc. Int. Symp. Softw. Testing Anal., 2010, pp. 61–72.
[cxvi] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze, “Automatic recovery from runtime failures,” in Proc. Int. Conf. Softw. Eng., 2013, pp. 782–791.
[cxvii] A. Carzaniga, A. Gorla, N. Perino, and M. Pezze, “Automatic workarounds for web applications,” in Proc. Int. Symp. Found. Softw. Eng., 2010, pp. 237–246.
[cxviii] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, “Automated atomicity-violation fixing,” in Proc. 32nd ACM SIGPLAN Conf. Program. Language Des. Implementation, 2011, pp. 389–400.
[cxix] W. Weimer, Z. P. Fry, and S. Forrest, “Leveraging program equivalence for adaptive program repair: Models and first results,” in Proc. IEEE/ACM 28th Int. Conf. Automated Softw. Eng., Nov. 2013, pp. 356–366.
[cxx] Z. Coker and M. Hafiz, “Program transformations to fix C integers,” in Proc. Int. Conf. Softw. Eng., 2013, pp. 792–801.
[cxxi] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in Proc. Int. Conf. Softw. Eng., 2013, pp. 802–811.
[cxxii] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “SemFix: Program repair via semantic analysis,” in Proc. Int. Conf. Softw. Eng., 2013, pp. 772–781.
[cxxiii] Y. Qi, X. Mao, and Y. Lei, “Efficient automated program repair through fault-recorded testing prioritization,” in Proc. Int. Conf. Softw. Maintenance, Eindhoven, The Netherlands, Sep. 2013, pp. 180–189.

[cxxiv] M. Monperrus, “A critical review of ‘Automatic patch generation learned from human-written patches’: Essay on the problem statement and the evaluation of automatic software repair,” in Proc. Int. Conf. Softw. Eng., 2014, pp. 234–242.
[cxxv] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra, “Data-guided repair of selection statements,” in Proc. 36th Int. Conf. Softw. Eng., 2014, pp. 243–253.
[cxxvi] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, “Minthint: Automated synthesis of repair hints,” in Proc. 36th Int. Conf. Softw. Eng., 2014, pp. 266–276.
[cxxvii] M. Harman, “The current state and future of search based software engineering,” in Proc. Int. Conf. Softw. Eng., 2007, pp. 342–357.
[cxxviii] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” in Proc. Int. Conf. Softw. Eng., 2012, pp. 3–13.
[cxxix] Z. P. Fry, B. Landau, and W. Weimer, “A human study of patch maintainability,” in Proc. Int. Symp. Softw. Testing. Anal., 2012, pp. 177–187.
[cxxx] C. Le Goues *et al*., "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," in *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236-1256, 1 Dec. 2015, doi: 10.1109/TSE.2015.2454513.
[cxxxi] (2020, Jun.) [Online]. Available: https://github.com/rjust/defects4j
[cxxxii] (2020, Jun.) [Online]. Available: https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData
[cxxxiii] (2020, Jun.) [Online]. Available: https://www.functionize.com/blog/functionize-root-cause-analysis-self-heal
[cxxxiv] Ori Bendet, 6 ways to rightsize your tests with analytics, https://techbeacon.com/app-dev-testing/6-ways-rightsize-your-tests-analytics
[cxxxv] B. Snyder and B. Curtis, "Using Analytics to Guide Improvement during an Agile–DevOps Transformation," in IEEE Software, vol. 35, no. 1, pp. 78-83, January/February 2018, doi: 10.1109/MS.2017.4541032.
[cxxxvi] A. J. Albrecht, "Measuring Application Development Productivity," Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, October 14–17, IBM Corporation (1979), pp. 83–92.
[cxxxvii] ISO/IEC JTC 1/SC 7 Software and systems engineering (2007-02-01). "ISO/IEC 14143". International Standards Organization. Retrieved 2019-02-26), Mark-II (ISO/IEC 20968:2002 Software engineering – MI II Function Point Analysis – Counting Practices Manual.
[cxxxviii] ISO/IEC 20926:2009 Software and systems engineering – Software measurement – IFPUG functional size measurement method.
[cxxxix] ISO/IEC 24570:2018 Software engineering – Nesma functional size measurement method version 2.3 – Definitions and counting guidelines for the application of Function Point Analysis.
[cxl] Capers Jones (October 2009) "Software Engineering Best Practices": pages 318–320.
[cxli] (2020, Jun.) [Online]. Available: Microsoft Power Bi Software, https://powerbi.microsoft.com/en-us/
[cxlii] Dan Radigan, Story points and estimation, https://www.atlassian.com/agile/project-management/estimation
[cxliii] Walton, G. H., Poore, J. H., & Trammell, C. J. (1995). Statistical testing of software based on a usage model. Software: Practice and Experience, 25(1), 97-108.
[cxliv] Kelly, D. P., & Oshana, R. S. (2000). Improving software quality using statistical testing techniques. Information and Software Technology, 42(12), 801-807.
[cxlv] Hao, J., and Mendes, E. (2006, July). Usage-based statistical testing of web applications. In Proceedings of the 6th international conference on Web engineering (pp. 17-24).
[cxlvi] Sprenkle, S., Cobb, C., & Pollock, L. (2012, April). Leveraging user-privilege classification to customize usage-based statistical models of web applications. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (pp. 161-170). IEEE.
[cxlvii] Soltana, G., Sabetzadeh, M., & Briand, L. C. (2017, October). Synthetic data generation for statistical testing. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 872-882). IEEE.