IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Extending the space of software test monitoring: practical experience

Mykhailo Lasynskyi[1], Janusz Sosnowski[2], (Senior Member, IEEE)
[1]Warsaw University of Technology, Institute of Computer Science, Warsaw 00-665, Poland
[2]Warsaw University of Technology, Institute of Computer Science, Warsaw 00-665, Poland

Corresponding author: J. Sosnowski (e-mail: janusz.sosnowski@pw.edu.pl).

This work was supported in part by Warsaw University of Technologyt under FEIT Faculty Grant 2021.

ABSTRACT Software reliability depends on the performed tests. Bug detection and diagnosis are based on test outcome (oracle) analysis. Most practical test reports do not provide sufficient information for localizing and correcting bugs. We have found the need to extend the space of test result observation in data and time perspectives. This resulted in tracing supplementary test result features in event logs. They are explored with combined text mining and log parsing techniques. Another important point is correlating test life cycle with project development history journaled in issue tracking and software version control repositories. Dealing with the outlined problems, neglected in the literature, we have introduced original analysis schemes. They focus on assessing test coverage, reasons of low diagnosability, and test result profiles. Multidimensional investigation of test features and their management is supported with the developed test infrastructure. This assures a holistic insight into the test efficiency to identify test scheme deficiencies (e.g., functional inadequacy, aging, insufficient coverage) and possible improvements (test set updates). Our studies have been verified in relevance to a real commercial project (industrial case study) and confronted with the experience of testers engaged in other projects.

INDEX TERMS bug diagnostics, software defect repository, software testing, testing framework, test monitoring.

I. INTRODUCTION

Software testing is still a challenging problem. Many relevant studies are reported in the literature. They mostly focus on test algorithms, test models, test coverage and diagnosability [1]-[6]. An important and practical issue is tracing test progress and results. Referring to our practical experience we have identified the need of multidimensional test monitoring based on combined analysis of software repositories generated during program testing, development, and exploitation. We consider test reports, event and performance logs, software issue and version control repositories. In the literature these repositories were analyzed separately and targeted at specific aspects, e.g., software reliability, software production improvement [7] [8]. They were not correlated explicitly with test processes.

Most test supporting tools facilitate managing test execution and collecting test results, typically stored in relevant repositories. They can also provide some statistical analysis tools. Examples of such systems are outlined in [2] [9] [10] and references therein. Executed test scenarios verify application functions and are composed of test steps which integrate some sets of test cases. Tests are executed in a specified sequence and produce results saved in a test repository. The results describe test progress and termination aspects, e.g., execution state (passed, non-passed), relevant test scenario step, test case. In the case of non-passed tests, we can get information on suspicious program modules, and optionally screen shots, stack traces, etc.

Event and performance logs as well as other software repositories can enhance the analysis of test results. Here, derivation of general and application-oriented features characterizing tests is needed. This aspect is neglected in the literature. Hence, our research has been targeted at exploring test repository and its correlation with a wide spectrum of the other ones, to get deeper insight into test execution processes and their effectiveness. We have introduced special metrics

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

and analysis tools to assess test diagnosability. Unlike unidirectional studies of software testing in classical research (often targeted at a single specific problem – compare Section II) we propose a holistic approach integrating diverse observation perspectives.

Our practical experience with software testing, including commercial projects, showed deficiencies of data comprised in test reports which delay diagnostics and the problem resolution. Our previous studies of software repositories and event logs induced an idea of using them to improve test observability and diagnostic capabilities. The gained experience resulted in three interleaving research goals:

R1 Identifying limitations and possible enhancements of test oracles.

R2 Developing a framework for tracing test execution context in project repositories.

R3 Correlating test assessment and project development issues.

Software repositories and event logs are ignored in testing processes due to lacking knowledge on their links with these processes and not available supporting tools. This issue was neglected in the literature and our paper fills this gap. It is focused on four tasks:

- Creating a program framework which integrates diverse tools for test management, monitoring, and extracting correlated data from available project repositories.
- Evaluation and taxonomy of software repository contents associated with testing aspects. It uses statistical and text mining techniques involving semantical aspects and relevant profile metrics (e.g., word usage, event keywords and sources).
- Deriving dependencies and interactions between testing and project development processes in short and long-time perspectives (referred to issue and version control repositories).
- Revealing test observability deficiencies and possible enhancements/upgrades of test sets (test result statistics, test coverage, test life cycle, test execution and diagnostic profiles).

The key point in our studies is the introduced original analysis method supported with an efficient framework for test management and monitoring. It integrates some available tools with developed special software modules. Our study facilitates to acquire substantial artefacts of software repositories and on which aspects of data analytics to focus when improving development and testing processes. It has been verified using real data collected during development of a commercial project of transaction-oriented domain. It can be considered as an instructive reference for other projects.

The rest of the paper is structured as follows. Section II provides the background and related works. Section III presents test management schemes and tools followed by an outline of the developed test framework. Section IV is devoted the analysis of event logs generated during test sessions. Section V outlines test execution schemes and results. The problem of correlating test features with issue and commit reports is studied in Section VI. Discussion of results and threats to validity are presented in Section VII followed by final conclusions in Section VIII.

II. BACKGROUND AND RELATED WORKS

In software development and maintenance effective testing is a crucial problem with several practical aspects. In many papers testing is correlated with software reliability. Software reliability growth models (SRGMs) are proposed to predict bugs in software, needed testing time to achieve an assumed quality level (release time), optimization of resources and test costs ([7] [11] and references therein). On the other hand, diverse publications focus on specific test problems: test algorithms based on structural, functional, and mixed approaches, test efficiency, software quality, etc. Test support tools have been developed to assess test efficiency and software quality [1] [4] [8] [10]. Software testability relates to costs of testing, and it is challenging to identify factors impacting these costs (e.g., basing on historical data in the company).

A. TEST CHALLENGES

Tests can be derived manually or in an automatic way based on application requirements, architectural and functional models, source code, etc. [2] [12] [13]. For this purpose, we can also use artificial intelligence [14]. Sometimes, the correctness of the program outcomes is not easy to specify – the test oracle problem. This problem was studied in [15] for scientific software and a metamorphic testing was proposed, it specifies output changes related to input changes.

In many applications testing system performance is important. The problem of the performance regression caused by introduced code commits is discussed in [16]. Some metrics associated with code changes are proposed to facilitate finding tests that can manifest performance degradation (regression). Test capability to check system performance is explored in [17]. This study is based on selecting performance issues from repositories of two open-source projects and analyzing test effectiveness before and after the relevant code commit (targeted at performance improvement). This results in searching for tests demonstrating performance improvements related to performance issue fixes. In consequence we can optimize regression test suits.

Some publications deal with test optimization issues: prioritization, reduction, and selection. Test prioritization is focused on test execution ordering to detect the highest number of faults at the earliest [18] – [21]. Test reduction and selection is correlated with checking their results and usage during project development [21]. Program diagnosability [22] can also impact quality of developed tests. Improving test observability by monitoring internal and output program variables is discussed in [23].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

Ideal tests should detect and localize (diagnose) all bugs in the program. Considering bug manifestations (program behavior), software engineers distinguish Bohrbugs, Heisenbugs, Mandelbugs and Shrodingbugs. However, more practical is bug taxonomy related to diagnostic and repair aspects. Correlating issue tracking (e.g., Jira) and version control system reports (e.g., Git) we can classify bugs taking into account their impact on code. In [24] four bug types have been distinguished. Type 1 and Type 2 refer to bugs which are fixed by modifying a single or more than one location in the code, respectively. Type 3 and Type 4 refer to multiple bugs fixed in the same location or the same set of locations, respectively. This is related to the quality of the code, which can be assessed basing on historical data in the considered company. Usually, Type 3 and 4 are more difficult to diagnose as those of Type 1 and 2. Predicting such bug classes facilitates bug handling processes by appropriate issue allocation to experienced developers. Similarly, we can consider bug severity (e.g., blocker, critical, major, minor, undefined). We can further precise bug categories (compare Section V) to correlate them with sources and repair aspects.

In practice, we should also examine incorrect behavior resulting from environment and system (hardware and software) interactions with the program, this may change in time, e.g., in relevance to introduced updates, functional extensions, etc. This should be inspected also during the test design and evolution, e.g., within the specified range of application versions (the oldest, the newest). In practice, tests are not perfect and should be systematically and continuously assessed as well as improved. In this process a wide scope monitoring is especially helpful.

Commercial software projects provide rich repositories on their development and operations. They comprise information on code changes (commits due to bug corrections, added functionalities, performance improvement) and bug or other requested tasks (issues) reports, which constitute development and field (users) knowledge related to some historical project/company perspective [17] [25]. Issue tracking repositories comprise requests of developing new functionalities, code modifications, correcting bugs, merging codes, etc. Bug reports should comprise information helpful to locate and fix bugs [26]. It describes what is expected to happen and what happens, samples of relevant code, stack traces, optionally used test cases, etc. The problem of assessing and improving the quality of bug reports is discussed in [26]. We have also explored this in [25]. Historical analysis of software repositories is also helpful in optimal attributing bugs to programmers for fixing (bug triaging [27]).

Another source of valuable data is the recorded run-time information produced by logging statements included in the code. Code changes may require appropriate log modifications and updates within the project lifetime. This can be supported with some quality metrics and classification methods [28] [29]. Logs record events useful in detecting and diagnosing anomalies [9] [30]. For this purpose, log parsing algorithms and tools have been developed [31] [32]. Anomaly detection can correlate with derived log classes, issues (e.g., task ids), code modules and time windows. Here, we can produce event count matrix, event sequence groups relevant to system workflow, etc. [33].

Diverse anomaly detection algorithms have been proposed based on machine learning, log classification, invariant mining, etc. [23] [33]. Usually, they do not assure sufficient capability to gain insight into the anomalies. This can be improved by better correlation of logs with the application (software) specificity (compare Section III and IV). The problem of optimizing logging statements is studied in [28]. It is based on automatically computed topics of code snippets as candidates to include logging statements. Understanding and interpreting log lines can be supported by tracing information provided by other repositories: issue tracking systems and referencing to source code (log statements, modules, comments), event call graphs [34]. These issues were not considered in relevance to software tests, which we include in our approach.

B. TEST ASSESSMENT

Special metrics are used to assess test quality. Most popular include branch and path coverage, decision/condition coverage, program mutation coverage [1] [2] [35] [36], and program version coverage [37]. They may suggest program components not sufficiently stressed by the test suite. This can be enhanced with the operational profile coverage expressed by application driven features (compare Section IV and V). In [37] test coverage is also considered in relevance to program versions. Test diagnosability evaluation needs deeper analysis. In [38] a spectrum-based approach is proposed which optimizes test suit sets in relevance to fault localization capabilities. It is based on an activity matrix correlating software components (e.g., class, method, branch) with stressing them test cases. This is some indirect approximation of test diagnosability.

The problem of fault localization was studied in many papers. In [39] some techniques of fault localization are evaluated using artificial faults generated by program mutations or manually. The basic techniques refer to program logging targeted at monitoring variable values and program state information, assertions, execution speed and memory usage profiling. More advanced techniques identify which parts of the program pass tests, which fail and suggest rankings of possible code areas comprising faults. They are described in [40] [41]: SBFL - spectrum base fault localization, MBFL - mutation-based fault localization. All these techniques are targeted at unit tests and correlate test results only with code coverage features. Their localization capabilities are still relatively low, so they did not attract practitioners. Fault localization is more complex in behavioral and wide scope of regression tests. The available software repositories comprise many artefacts which can be used to assess diagnosability features of the test suite and show appearing difficulties or

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

possible improvements, which we study in the paper. Tracing links between test cases and the source code of units under test is also useful in diagnostic processes [42]. The correlation between test suite metrics and the quality of automatically generated software repairs (repair success, repair time) is studied in [43].

We have aggregated research topics on test issues based on the presented literature review. This resulted in a taxonomy of considered problems that is summarized in the first column of Table 1, it includes also representative references considered in the paper. Moreover, it facilitates to position the contribution of our research. Published studies on software repositories and event logs were not combined with testing issues. Nevertheless, in the second column of Table 1, we list indicative publications focused on research areas which, in our opinion, can be combined with testing. It is worth commenting the contents of the recent survey paper on software testing [21]. It covers over 170 positions (including some other surveys). Despite so wide and comprehensive survey we did not notice there such topics as: test life cycle, test observability, test outcome analysis, event logs and other software repositories. Similarly, in the review of 21 testing tools [44] these issues were not mentioned. In the other survey [1], event logs and assertions are only mentioned as helpful, without deeper insight in their treatment and significance. This additionally confirms negligence of the pointed problems in correlation with testing, which we found as important practical issues needing investigation.

TABLE I
PROBLEM PROFILES OF PUBLICATION

Test issues	Software repository issues
Test design principles and tools [2, 5, 8, 12 - 14, 34-36, 44]	Logging schemes [9, 28, 30]
Testing project performance [16, 17]	Log analysis and parsing [9, 30 - 32]
Bug classification [17, 24-26, 28, 29]	Anomaly detection [9, 33, 34]
Test quality assessment [1 - 4, 6, 10, 11, 15, 21, 34-37]	Log statement optimization [9, 34]
Fault localization and program diagnosability [3, 22, 23, 38, 39]	Issue and version control tracking [7, 25, 30, 37]
Test optimization and prioritization [18 – 23, 42, 43]	Bug triaging [27]

The created test scenarios are upgraded during the development and maintenance of the program. Moreover, in a large extent they are used during regression testing in relevance to function upgrades, extensions, or corrections. Test quality should be evaluated considering detection and diagnosability aspects. Detectability can be assessed indirectly by module or code coverage, fault seeding (program mutation techniques) as well as by analyzing reported issues during exploitations (e.g., Jira repository). Test diagnosability features can be evaluated by tracing bug handling times including the number of exchanged comments between project actors and users up to the final problem resolution. Here, evaluation of the semantic contents of the software repository is helpful.

Depending upon the developed test schemes, the relevant test oracles are more or less detailed. In the simplest case we have pass/no pass notion with the relevant test scenario/step/test case or code lines. This can be enhanced with registered events during test execution which is neglected in the literature. Hence, our research has been targeted at exploring test repositories and their correlation with other ones. This resulted in 3 goals of our studies: i) finding complementary data enhancing test diagnosability, ii) assessing test efficiency and test schemes, iii) revealing possible improvements. The project performance can be tested directly by checking response times of requested services or indirectly by monitoring usage of system resources (e.g., processors, memory, transmission links), queue lengths, etc. For this purpose, we can use available performance monitoring programs (e.g., IBM Tivoli) including the available ones in the operating systems. Designing tests, we must consider program functional and performance features. Moreover, in many developed systems some auto diagnostic functions are included to detect/mitigate incorrect user activities, abnormal environment interactions, testing these features needs complex simulations. To support our research, we have developed an original and universal test framework outlined in Section III.

III. TEST ENVIRONMENT

We consider the problem of monitoring software testing in relation to a commercial system oriented at financial transaction processing outlined in Section A. This is a quite complex and representative enterprise system. It handles requests from many clients and provides financial services. The system is systematically updated and adapted to new functional requirements and performance features appearing in its life cycle. It has been developed, maintained, expended, and upgraded for many years. Hence, efficient testing and issue handling were challenging problems, that resulted in developing a flexible and comprehensible test framework (Section B). This framework provides rich data (Section C) that is useful to assess and improve testing processes.

A. TESTED SYSTEM FEATURES

The tested system is dedicated for processing financial transactions related to accepted client orders, generating invoices on selected positions, handling money transfers, performing audits of the whole system or selected transactions, etc. It is a distributed system involving many servers installed on several machines combined into a cluster assuring efficient data communication for clients and system components (including product data bases). Different configurations of services are possible, they adapt to several software environments and the hardware infrastructure. High dependability, coherence, and auditing capabilities of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

performed activities are assured. Within this system we distinguish three modules: MM1 – web application, MM2 – web services, MM3 – databases.

MM1 module is written in Java and operates in an enterprise application server. It assures the execution environment for applications based on JVM (Java Virtual Machine). The application server handles executing Java servlets and Java Server Pages (JSP). JSP pages use XML markers and scriplets (a piece of Java code embedded in HTML-like JSP code) to separate page contents from their formatting logic level (which generates the servlet contents). All formatting tags (HTML and XML) are appended directly to the generated HTML page. Container Java EE handles additional functions such as server load distribution. It includes dynamic (servlets, Java Server pages, Java classes) and static (HTM pages, images) resources.

MM2 relates to web services handled by Enterprise Service Bus which manages received messages and constitutes an intermediate programming for distributing jobs between connected components of the application. It assures product related tasks that include accepting, translating, and handling client requests to deal with messages between services, monitoring and controlling routing of messages, resolving communication problems, administering deployment and versioning of services. It also assures event, security, and exception handling, transforming, and mapping data, queuing, and sequencing messages, protocol conversion, etc.

MM3 involves two relational databases The first one is used for operational and configuration data; the second database is used as a tool for integration with another financial system. The system consists of 1100 000 lines of source code not including libraries and used enterprise solutions.

B. TEST FRAMEWORK

Testing complex systems involves diverse supporting tools, such popular as: Junit library (for unit testing), Cucumber library (for preparing high abstract level test cases – behaviour driven development), Selenium for testing internet applications (focused on web page elements). Junit is a library which facilitates activating and checking unit tests from a pointed catalogue or a class file. Selenim is a high-level interface library implemented for many browsers to manipulate web page elements. Selenium Grid is a cluster of servers used to arrange a distributed network and activating simultaneously multiple browsers on diverse computers. A central server sends requests to distributed nodes. It facilitates complex testing in the distributed environment. The Cucumber library supports describing test cases at a higher abstract level, we use automatic functional and acceptance tests based on Behavioural Driven Development concept. Test cases specify actions and expected behaviour of the tested program (TP) in a logical business-oriented language (Gherkin syntax), e.g., entering login, page, user id, user password and activity, expected confirmation of correct login (or a missed one). Cucumber uses Selenium library covering many browsers (e.g., Firefox, Chrome, Internet Explorer, Safari). It supports such actions as clicking on a page element, selecting an element, entering some text, reading its contents, scrolling the window, etc.

During test execution the tested system can be additionally monitored with Metricbeat which provides metrics and statistics on server resource usage, e.g., CPU, memory, network, discs, filesystem, services. Moreover, various types of logs (e.g., system, application, security, errors) are collected with the Filebeat program. Elasticsearch tool supports storing the collected data, searching and analysing their contents, this can be enhanced with Kibana providing visualization of results in diverse graphical forms.

The outlined tools generate independently a lot of data. We found the need of coordinating operation of these tools, consolidating the collected data and analysing them in a holistic way. This resulted in developing Cucumber-Monitoring-Plugin (CMP) which integrates the main testing system (Cucumber + Selenium) with the set of tools for monitoring virtual machines. This constitutes the main part of the developed test framework, as shown in fig.1. CMP bases on Java version 8, which facilitates integration with testing systems. It was built using Apache Maven v.3.6 and relevant Project Object Model concept, so it can be used for a wider scope of projects based on Cucumber technology. Integration with the main testing system (Cucumber + Selenium) is performed via specially defined and implemented interfaces to three modules. This scheme was verified with a real complex project developed according to typical software engineering rules used by many IT companies. It is quite universal and can be applied in a wide spectrum of projects.

The data flow in the test framework (FT) is as follows: the main testing system generates requests on pages (activities on pages), analyses responses of the system and page behaviour. This activity results from executed test case specification which may involve simulation of system users and invoked reaction of relevant services. Moreover, Elasticstack handles collected monitored data by Filebeat and Metricbeat. The developed CMP plugin correlates monitored data with executed test cases as illustrated in Figure 1.

CMP program includes the following features: i) analysis of test progress (initialization/completion times of executed test steps, storing relevant artefacts, test results, etc.), ii) extracting values of monitored performance parameters (provided by *Elasticsearch*) relevant to test case execution, iii) integration with *Kibana* module facilitating visualization of specified statistical features. The collected data in diverse repositories are processed (explored) with the specially developed analitic module (DAS – Data Analysis Scripts), it derives statistics of specified metrics and also comprises special scripts to detect nonstandard situations.

The developed test set is composed of functional test groups, each comprising some set of test scenarios involving predefined test steps (test cases). As compared with classical test approaches targeted at direct test control and oracle

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

checking/interpretation we significantly extend the observation space in data and time perspectives. This is assured by the developed CMP and DAS modules. Depending on the organization of data repositories we must extract data relevant to test processing and its environment, if needed perform data anonymization, identify/compensate time skews of reports in different repositories (based on context analysis), archiving data for long time behavioural analysis. Developing the analytic module (DAS) needed investigation of a wide scope of repositories and adaptation to the specificity of collected data. This process has been supported with the introduced original text mining schemes in DAS dealing with peculiarities of registered reports incompatible with classical document text mining approaches. This allowed us to identify report features providing valuable information in assessing test detectability and diagnosability. The proposed analysis absorbs our previous experience with monitoring system operation, software development and maintenance processes [7] [25] [30] [37].

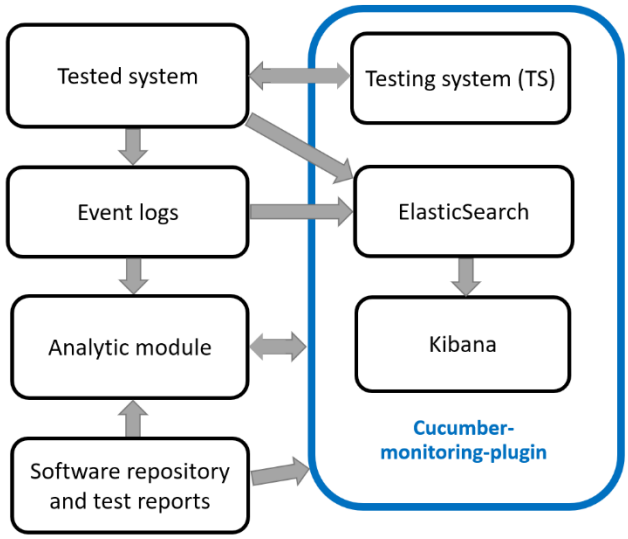

FIGURE 1. Block diagram of the test framework (TF).

C. SCOPE OF DATA ANALYSIS

The collected data during test executions create a quite complex repository which comprises diverse logs on executed test elements, results, generated events, etc. An important issue is analyzing their contents, derive short and long period statistics, find correlations between different logs, identify abnormalities, derive some metrics characterizing test efficiency, etc. Beyond data repositories directly related to testing processes we also have software repositories related to issue/task requests (e.g., Jira) and software version control (e.g., Git) with specification of performed code commits, configuration changes, etc. They describe development process progress and the range of code changes. This can be correlated with test repositories to get a better insight on the test cost and effectiveness during the software lifecycle. Test relevant data are interspersed with a bulk of other project data and their extraction needs tracing time and context dependencies.

Analyzing a wide scope of repositories allows us to reveal test deficiencies, imperfections of reports in repositories and formulate recommendations for their improvements. For example, reported bugs by users may be attributed to skipped or obsolete tests (needing refinement), the project usage profile can differ from the testing one, some needed functions can be missed. Introducing new functionalities or modifying some of older ones needs not only tests targeted at these changes but also regression tests checking the impact of these changes on the remaining not modified functionalities. Moreover, we can predict the cost of bug repairs basing on historical test results and involved commits (bug handling time, range of code modifications).

The above-listed problems and our practical experience with testing resulted in enhancing classical testing schemes with monitoring their execution at different observation perspectives (test oracles, generated events, performance metrics) and tracing historical reports in diverse software repositories (issue tracking and software version control). The developed test framework provides a multidimensional perception of test progress. Basing on the integrated extraction of characteristic features from a wide scope of relevant software repositories, we have introduced original quality metrics covering different dimensions. This results in a holistic view on test effectiveness and revealing possible enhancements. As opposed to other publication on testing our approach is characterized by two features: 1) assuring high attention to extract useful data details (fine-grained), 2) a wide-scope (spread over diverse data sources) exploration of testing and software development repositories. Correlating test results with other software repositories we take into account timestamps (including estimated time skews), report sources and semantical context (based on text mining), non-relevant reports are filtered out. Checking the impact of executed tests on issue handling processes (e.g., in Jira repository) and the range of code changes gives new additional criteria to evaluate test efficiency. The presented problems are discussed in Sections IV-VI.

IV. EVENT LOG ANALYSIS

Depending upon the system diverse logs are generated, e.g., system, security, application logs. They can also be partitioned into diverse groups covering relevant issues. During testing processes especially interesting are application logs, we experienced this in our practice. An important issue is to adapt log analysis processes to their specificity in the real project. Hence, we explored software repositories correlated with regression test scenarios executed during development of a complex transaction-oriented system. Typically, each testing session was performed in the night (about 6 hours) and generated about 5MBs of application logs (about 350 000 words, 19-32 thousand events). Analyzing event logs

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

correlated with executed tests we identified three research goals:

1) General text mining – focused on log dictionary analysis, word and n-gram (phrase) frequency profiles (Section IV A)
2) Structural/semantical log analysis – log parsing, application-oriented keywords, log feature profiles (Section IV B)
3) Log profile analysis in time - log feature distributions and comparative studies (Section IV C).

A. GENERAL TEXT MINING

In classical text mining we deal with documents based on natural language. The specificity of the text in log entries is quite different and needs cognitive studies of used word dictionary aspects. Exploring dictionary taxonomy (morphology) and profiles we focus on word syntactical and semantical issues. Typically, a word is considered as a sequence of characters bounded by space or other specified characters (e.g., dot, semicolon). At the morphology level we can distinguish words comprising only letters, numbers, mixture of alphanumeric and special characters. Next, it is reasonable to classify them in relevance to composition structure, e.g., single words and complex words, which constitute a concatenation of simple words using or not some linking characters (e.g. _, +, ., :). The complex words can refer to some predefined patterns (usually specified with regular expressions), e.g., file paths, Ip addresses, resource identifiers (e.g., URI). Basing on semantical properties we distinguish 4 classes of words: A - natural language words (compatible with thesaurus dictionaries), B - special IT words (e.g., thread, exception, commit), C - specific terms for the analyzed application (acronyms, names of components), D - code words (numerical, hexadecimal, typically used for timestamps, event ids, error codes). It is also worth mentioning that sometimes registered words have incorrect spelling.

The log dictionary used in the considered set of log files (L) can be characterized by the set of unique words D_L, its cardinality $|D_L|$ and profiles of word usage. We introduce three types of such profiles as sets of values in descending order (denoted with sharp brackets <….>):

- Frequency profile gives the number of occurrences of each word $w_i \ni D$ in the set of log files L:

$$FP(L) =< f(w_i|L): w_i \ni D_L > \quad (1)$$

- Relative usage profile, i.e., FP(L) expressed in percent

$$RP(L) =< \frac{f(w_i|L)}{|D_L|}: w_i \ni D_L > \quad (2)$$

- Aggregated usage profile AP(L) is defined in relevance to ordered sets of n value ranges $R_j = [r_{j-}, r_{j+}]$, $r_{j-}, r_{j+} \ni RP(L) = \{ r_1 \dots r_k\}$, $k=|RP(L)|$, $r_{j-} \le r_{j+}$, $j=1\dots n$, $r_{1+} = r_1$, $r_{n+} = r_{|RP(L)|}$, DL_j is the set of all unique words corresponding to R_j,(i.e. $\{w_i: r_i \ni R_j\}$ than the aggregated profile is:

$$AP(L) =< < R_j, u_j >: u_j = \sum_{i \ni D_{Lj}} \frac{f(w_i|L)}{|D_L|}, j = 1 \dots n > \quad (3)$$

For an illustration we give the derived aggregated word usage profile for the set of 25 daily log files (L_{25}):

AP(L_{25}) = < <[9.7%-10.5%], 0.5%>; <[1.5%-2.2%], 0.8%>; <[0.4%-0.8%],1.8%>; <[0.05%-0.4%], 4.9%>; <[0.01%-0.05%], 16%>; <[0.006%-0.01%], 75%>>

The aggregation ranges can be selected to reflect similar frequency values. In practice, this profile is neither uniform nor compatible with usage profiles of texts in natural language. Relatively small group of words shows high frequency (e.g., *for*-10.5%, *INFO*- 10.0%, *ExecuteThread* – 9.2%, *queue* – 9.2%, *ACTIVE* – 9.2%). They are not important in log classification and result from the logger configuration. Many application oriented technical words dominate, however their frequency is low. Some semantic grouping of words can be useful to derive interesting log features (compare Section IV B). Having analyzed identified words in generated application logs we found 28.5% of complex words of various structures (some examples are given later). Log entries comprised mostly English words, however 8% were Polish (which complicates the analysis). The text in related logs was based on dictionary of several thousand of unique words. We observed these features also in logs of other projects [30].

Due to the mixture of comprised information, raw log data usually looks a little bit strange, as in the following examples (transaction accept, and alert message):

2019-03-08 00:32:17,866 [[ACTIVE] ExecuteThread: '0' for queue: 'self.tuning.ClassName (self-tuning)'] INFO com.package.ClassName - transactionStatus – AC

2019-04-01 20:09:02,677 [[ACTIVE] ExecuteThread: '1' for queue: 'self.tuning.ClassName (self-tuning)'] INFO com.package.ClassName - Alert args: alertType: TransactionResponseTimeout, source: REST, Details null, Args: {transactionId = 00000000000XX0000000000, agentCode = 00000000}

Quite often complex words relate to C category and may comprise words from other categories (including A). Hence, partitioning these words may enhance text mining, e.g., "exception" component (expressing some general event) can be concatenated with other precising words. For an illustration we give an excerpt of complex words:

FileProcessingServiceCriticalException,
SourceDirectoryNotFoundException
Java.lang.NullpointerException,
java.lang.illegalArgumentEception,
org.springframework.web.client.HttpsServerErrorException

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

org.hibernate.util.JBDCExceptionReporter
NetworkAvailabilityErrorHandler

Many other examples are given in Section IV B. Complex words are created by simple concatenation of words, a string of words separated with non-alphanumeric characters (e.g., hyphen, underline, dot), letter case separated words (upper case for the first character of the word). In practice, complex words are defined by accepted conventions related to software architecture, program language (e.g., class and method names) or company. Hence, they can be easily analyzed with simple regular expressions.

Combining log parsing with text mining techniques gives a deeper insight in their meaning. Derived keywords can be correlated with fixed log parts which are followed by variables specified using other word categories, sometimes they are preceded with the colon character (:). Description parts of logs base mostly on natural language words, however those from other classes also appear sporadically. The text mining process is supported with a set of regular expressions targeted at specific character patterns, e.g., date, time, IP addresses, file paths. This process can be performed in some hierarchical way, e.g., we can split complex words into simple word components and perform their semantical classification. The next step is to use other text mining techniques, e.g., targeted at identifying keywords or characteristic n-grams. Here, we can use word frequency, context, tf-idf and other text mining metrics (compare Section IV B). Some complex words or sequences of words express values of specified parameters. The name of the parameter can be identified with parsing techniques that extract the fixed text part (appearing frequently). In classical parsing schemes the variable part is usually replaced by * character [30,32], below we give an excerpt of log messages:

(*) executed command from host (*) at (*)
Failed login attempt from host (*) at (*) by (*)

In our approach, we do not skip the parameters (neglected in other approaches), we also focus on analyzing their distribution, anomalies etc. They can contain important data with significant semantical values (e.g., error code, alarm type, event source). Searching for keywords can be preceded with introducing some set of predefined keywords based on developer knowledge, which can be further extended. As compared with classical log analysis based on parsing algorithms our approach assures deeper insight in their contents and facilitates interpretation, which is helpful in test diagnosis. Some optimization process is included after preliminary log profile studies related to derived dictionaries, frequencies of used terms, etc.

The basic text mining (word frequency analysis) allowed us to identify major keywords, for example: *transactionId, filePath, eventId, reason code* (code of transaction rejection), *eventtimestamp* (time of sending the message by the system), *remoteTimestamp* (time specified by the sender), *transactionStatus*. Some other keywords related to application server, e.g., INFO (level of log), *Executionthread*, queue. Usually, such keywords are followed by some variable texts (parameters) correlated with them. They can be single words or a set of words sometimes comprising special characters, embraced by brackets, etc. Beyond that log entries may comprise non structured text messages. For example, a descriptive part of an alert log entry comprised the following text (it also includes Polish word at the end – "*Blad skladni*"):

ValidationError, severity Major, details: filename:xxxxxxxxx full/file/path, Blad skladni: Attempt to access field 'parseException' with null value

The classical text mining analysis based on tf-idf parameters did not provide interesting words. Hence, we decided to analyze phrases (n-grams). Some 7- or 8-word phrases with low tf-idf related to interesting events signaling some critical issues, they comprise negative words (marked in bold) and relevant tf-idf of the n-gram.

***cannot** create claim incorrect related credit transfer (3.83)*
caused http response fiddler lookup failed when (2.31)
active recovered another instance this cause inconsistent (1.22)
***couldn** read message file neither from directory (1.15)*
***error** zasób zajęty zlecono uzyskanie nowait upłynął (1.15)*
***erro**r occurred during **error** handling give up (0.71)*
*details **blad** aplikacyjny podczas rejestracji reklamacji oraz reklamacja (0.41)*

Looking for such critical events we have specified a set of negative words: 25 English words (error, alert, fail, failed, abort, reject, rejected, invalid, incorrect, cannot, interrupt, interrupted, inaccessible, break, stop, not, phrases) and 25 polish words (including blad denoting error). Having selected log entries comprising these words we looked for 7-word phrases comprising them. We have identified 691 such negative phrases (with relatively high if-idf), they suggested some problems. Logs comprising *Alert* keyword provide valuable application specific diagnostic information, similarly as complex words comprising *Exception*, this is analyzed in Section IV B.

B. STRUCTURAL/SEMANTICAL ANALYSIS

The identified keywords correlate log entries with specific issues, e.g., exceptions, alerts, errors appearance, performed activities in the application (e.g., transactions). Some keywords are followed by a related variable value (digital or alphanumeric character strings). In classical log parsing schemes (e.g. [31] [32]) log templates are derived with fixed parts (equivalent to keywords) and variables specified with * character. In our approach keywords are correlated with application specificity. Usually, identified variables comprise valuable data (neglected in other approaches), worth deeper analysis, in particular, statistical distribution of reported

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

values. For an illustration in Table 2 we give distribution of some basic keyword's appearance in 25 log files from the executed regression testing in 25 subsequent days. We give minimal, average, and maximal number of specified keywords within daily log files. This can be confronted with the number of entries in a log file in the range 18000-24000. Hence, these keywords appear in 5-20% of log entries. Here, it is worth noting that most log events are single line with average 20 words. However, some events, e.g., related to exceptions may comprise from tenths to hundred lines (e.g., due to included stack trace).

TABLE II
STATISTICS OF BASIC KEYWORDS

	Trans Id	File Path	Reason Code	Even tId	Alert Type	Source
Min	1255	840	763	763	0	0
Av.	4603.0	3427.2	3015.2	3015.2	558.8	588.9
Max	5823	4593	4029	4029	2205	2205

TABLE III
DISTRIBUTION OF KEYWORD RELATED VALUES

	TransId	Reason Code	Eventid	Alert Type	Source
Range	473-2283	5-14	8-11	0-17	0-15
Min.	1	1-7	3-208	0-4	0-15
Max.	6-10	3-237	123-625	0-2048	0-2047

An illustrative distribution of 5 main keyword values is given in Table 3. For each considered keyword we give the range of unique values registered over all entries of the 25 files, followed by the number of minimal and maximal occurrence of the same value. The most interesting are the reason code, alert type, and source variables, especially those which appeared less and most frequently.

Another group of logs relates to errors with specified thread name, exception class within this thread, short description and optionally stack trace. Typically, such log entries comprised 65 lines (about 400 words separated by space or dot). In addition, an alert (type and source) is given for every error with the domain meaning.

In Java an exception can be caused by another exception, hence in the error report we have specification of the recent exception and a sequence of causing it higher level exceptions (up to the class of the primary one). For each of them the stack tracing is included. Some statistics of exceptions is given in Figure 2. X-axis shows the number of lines comprised within the entries and y-axis shows the number of their appearance. Moreover, the entry severity is denoted by appropriate color: i) yellow - exceptions with short description (0-10 lines) without stack trace; ii) blue – exceptions with localization in code, the name of relevant class (11-50 lines); iii) green – exceptions with a single primary exception (51-1000 lines); iv) red - exceptions with 2 or more primary exceptions (101 and more lines). The biggest is the first group with a single line (3052 cases – beyond the scale of the plot). In fact, it relates to standard not critical behavior, e.g.: errors related to validation, user actions, configuration checking. The second group refers to local errors (of lower significance), the third group specifies application problems previewed by developers. These problems relate to several layers of the application, e.g., exception informing about unsuccessful configuration changes caused by inaccessibility to the database. The last red group relates to unforeseen errors, the exception propagated for a longer time in the stack trace. This may result from errors not announced in application (not intercepted by the application and logged at the level of application server) or recursive method calling or looping. In the case of generating many exceptions and relevant stack traces, log entries can show only a limited number of them. If needed detailed analysis can be resumed in the debug mode.

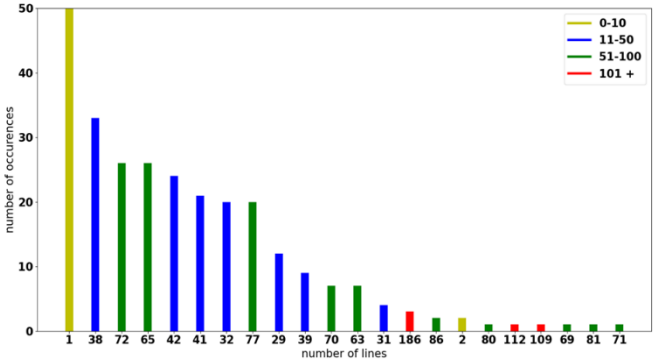

FIGURE 2. Exception size/severity profile diagram.

Logs with alerts, beyond basic components of event logs, comprise the following specifications (fields): *alertType*, source, details (optional textual description), *args* – arguments, *transactionId*, *agentCode* – identifier of the message sending unit. We distinguish 7 alert type groups listed in Table 4. They facilitate to identify reported problems. For the better problem localization, we can also use the alert source, here we have 6 source groups, listed in Table 5. Table 4 and 5 also specify (in brackets) the numbers of distinguished alert types and sources for each category, respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

TABLE IV
ALERT TYPES

Group type	Description (number of types)
AccountJob	Planned processes correlated with running operations on accounts (3)
Clearing	Alert correlated with account debit (3)
Provisioning	Alerts correlated with account credit or debit (4)
Scheduler	Alerts related to errors during planned processes (3): SchedulerCriticalError, SchedulerError, SchedulerInfo
Transaction	Errors related to money transfer process (7): TransactionAgentInactive, TransactionBadFormat, TransactionBadMessage, TransactionClaimCreated, TransactionDuplicated, TransactionResponseTimeout, TransactionTimeExceeded
Unavilabale	Alerts related to actor inaccessibility (5)
Others	Positive confirmations and some negative alerts (17), e.g. ClearingStatusUnknown, DuplicatedTransLimitExceeded, InconsistentTransaction, SecurityViolated, UnknownError,

TABLE VI
ALERT SOURCES

Source category	Description (number of sources)
Application error	Application errors during network operations, resulting from processing algorithms (8)
Confirmation	Validation errors related to message confirmation (2)
Scheduler	Errors of planned processes (7 scheduler functions)
Timeouts	Crossing waiting time limitations (4)
Transactions	Errors related to specified transaction processes (5)
Others	Related to diverse functions (17)

Distribution of reported alerts for all test sessions performed per one year period was as follows: *SchedulerCriticalError* (50.33%), *TransactionTimeExceeded* (22.92%), *TransactionResponseTimeout* (19.16%), *TransactionsClaimCreated* (4.28%), others (3.31%). However, this distribution for each session may differ, for example for one day test sessions (6 hours) it was: *SchedulerCriticalError* (70.04%), *SecurityViolated* (10.65%), *TransactionAgentInactive* (8.53%), *UnknownError* (5.49%) and others (4.29%). This is a consequence of code changes in the tested application or test changes. Hence, tracing fluctuations of report features can provide useful knowledge on test strategies.

C. LOG PROFILE ANALYSIS IN TIME

The log profile analysis (in time) can be targeted at tracing changes (or stability) of specified features. This can be focused on dictionary changes within specified categories (e.g., file paths, thread, method names). Here, we can consider long or short time perspectives, correlate them with application development life cycle, e.g., in relevance to test monitoring, issue and software version control repositories (Section V and VI).

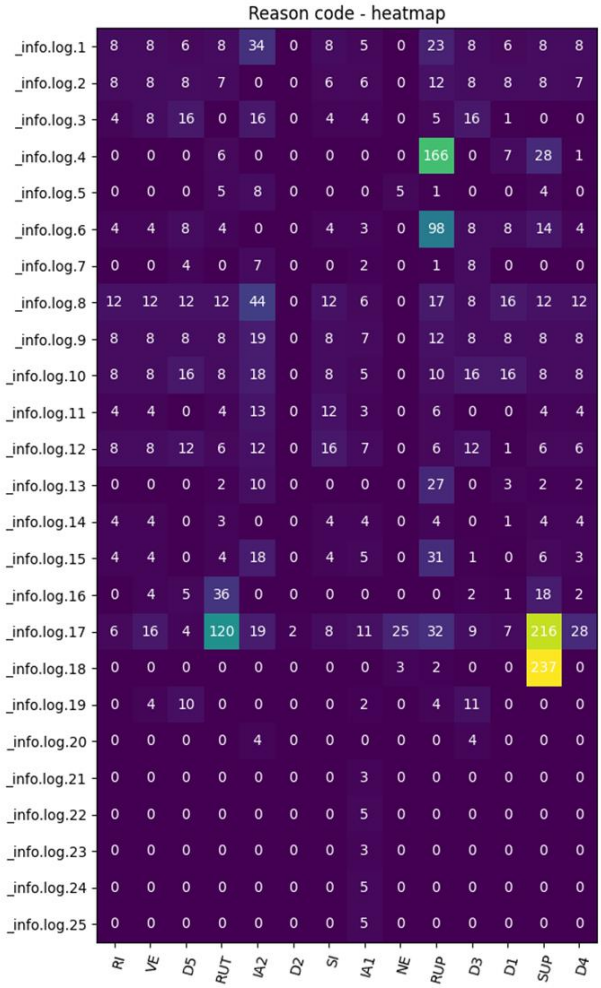

FIGURE 3. **Heatmap covering transaction rejection reason codes.**

In this section we focus on analyzing logs using cross section heat maps to trace differences in specified types of features. They form 2 dimensional matrices with x-axis (columns) specifying considered features (e.g., alert types, exceptions, transaction termination codes), y-axis shows subsequent log files (e.g., relevant to one test run). The matrix entry H(i,j) shows the number of identified features (corresponding to the j-th column) within the considered i-th log file. The entry background color can be correlated with this number, to facilitate revealing some regularities or irregularities. Subsequent values H(i,j) of the j-th column show distribution in time (test runs) of the j-th feature. We can trace cross section heat maps $H(i,j)_{TS}$ by filtering logs in relevance to test scenarios (TS). Disappearance or reduction of some features in time can be correlated with code corrections (new version, bug correction). On the other hand, the appearance or a significant increase of some features may result from new functionalities, imperfect bug corrections, system configuration changes, etc. The identified suspected

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

feature deviations can be explained/interpreted by correlating their appearance with other software logs.

Figure 3 shows a heatmap related to transaction rejection reason code. For the most log files, low values of each code (0-10) dominate. Only one log file (log-17) comprised all reason codes (in total 420 entries): RUP, RUT – receiver unavailable permanently, temporarily; RI - receiver inactive; IA1, IA2 inappropriate amount; D1, D2, D3, D4, D5 – diverse duplicates; SI – sender inactive; SUP – sender unavailable permanently, temporarily; VE validation error; NE network error. The highest number of registered reason codes related to SUP (216, 237) and RUT (120) codes. In a similar way we analyzed other keywords. *Transactionid* assumed the following values: SRAC – send request of account charging; RCAC – received confirmation of account charging; TMT - timeout; SR – sent rejections; RR1, RR2 - received rejection type1, 2; RPA – returned positive answer; SA – sent authorization; SRCA – sent request for credit acknowledge; RA – receiver authorization; RCA – received credit acknowledgement. We observed relatively uniform (200-400) distribution for 7 values (SRAC, RPA, SA, SRCA, RCAC, RA, RCA), 3 with very low values (TMT, SR, RR2) and one (RR1) with the medium average value (50), however large dispersion (3-470).

The heat map for alert types revealed bigger number of some critical types: *unavailable sender* (187 and 146 for log11 and 12, respectively), *SenderCriticalError* (533-2045 for logs 13-15), *AccountJobinactive* (555-1405 for logs 11 and 10), *transactioBad Message* (218-251 for logs 1-3), *securityviolation* (200-604 for logs 18, 11, 12), *transactionResponseTimeout* (100-604 for logs 13). Five logs did not comprise alerts, for 4 logs alerts constituted a fraction of percent, 3 logs with about 2-5%, the remaining ones below 2%. The heat map for identified alert sources (20) revealed most alerts related to *Scheduler* (500-2007 for log 13-15 and 10-11), other 6 sources related to several timeout generators and bad message validation with significant values (50-250) for about 30% of logs. The heatmaps for registered exceptions (25 types) showed for most logs low values (0-10) per exception, and 10-50 exceptions per log file in total. However, two exceptions *FileProcessingFileServer criticalerror* and *SourceDirectorynotFoundError* showed very big values (533-2045 for logs 13-15). They correlated with Source Scheduler (552-2047) and alert type *SchedulerErrorCritica*l (533-2045) for the same logs. For these logs, the percentage of entries comprising exceptions (in fact double ones) was 2-5%. We can analyze other correlations, e.g., logs with abnormal values with test reports (Section V).

V. TEST EXECUTION MONITORING

Tests can be structured at three levels: test step, test scenario (a set of subsequent test steps related to a test case), test block (group) – set of test scenarios on a specified functionality. We define 5 states of test steps: passed (positive execution), failed (negative test result), skipped (not executed), undefined (e.g., not implemented in the test), pending (test step under execution). The test scenario is passed while all relevant test steps were passed or failed if any test step failed. We define the test run as the complete set of test blocks covering all software functionalities. Typically, it includes 500-1000 test cases, each comprising 10-20 test steps. Reports on the test execution are stored in test result files. In the considered project regression test runs were executed in nights by about 6 hours.

A. BASIC TEST STATISTICS

Table 6 shows distribution of test results of one day run covering 31 test groups generated by the test management module. For each test group we give the number of relevant test steps (followed by the number of failed and skipped ones), the number of test scenarios (with specified failed ones) and the test execution time (minutes). For correctly executed test groups we give only ranges of relevant values. The total execution time was 6 hours 11 minutes. The aggregated statistics of the whole test session (numbers of test steps, test scenarios and relevant distribution in percent) are given in bottom table rows. The summarized test report is analyzed, in consequence of this analysis not passed tests are verified to check the reason of negative result. Sometimes this is caused by an inappropriate or incorrect test (e.g., wrong assertion), test or its environment configuration flow (e.g., inaccurate usage of the testing framework), etc. Negative results need transferring them for further processing, e.g., labelling issue id in Jira, etc. Test result presentation assures hierarchical access to detailed data via CMP module (Section III).

TABLE VI
EXCERPT OF A SYNTHETIC TEST REPORT

Test Group	Test steps			Test scenarios		Test time [min]
	Passed	Failed	Skipped	Passed	Failed	
G1	432	18	192	14	18	72
G2	815	17	90	42	17	80
G3	175	6	23	25	6	85
G4	180	6	10	20	6	05
G5	82	5	8	9	5	11
G6	180	4	3	23	4	10
G7	41	3	3	4	3	07
G8	409	2	0	136	2	11
G9	30	2	2	4	2	05
G10	878	1	12	60	1	44
G11	35	1	0	2	1	05
G12-31	1-649	0	0	1-144	0	0.1-17
Total	6181	65	343	821	65	371
Distrib.	93.8%	1.0%	5.2%	92.7%	7.3%	

Test session profiles can differ in time. This is illustrated in Figure 4 for subsequent daily regression session tests covering about 600 days (long term statistics). The percentage of the negative tests ranged from 5 to 42%. The x-axis specifies subsequent test runs, y-axis shows the percentage of the negative tests (average value aggregated over consecutive 30 test runs). Higher values were correlated with bigger code changes (new functionalities, major code corrections). Such

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

analysis can be performed for a shorter time perspective and a lower aggregation window (e.g., 1 day). We can also focus on other test features, e.g., the first use (execution) of the test and its expiration date (due to irrelevant code, new functionalities, updates), confirmed skipped faults reported by users.

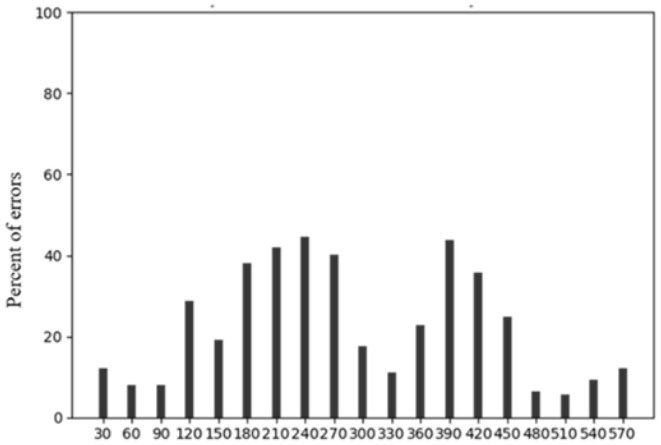

FIGURE 4. Distribution of negative test results (aggregation per 30 days).

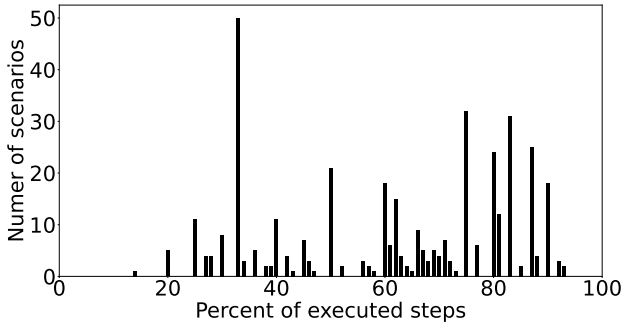

FIGURE 5. Distribution of correctly completed test steps (in percent) over test scenarios executed in 5 subsequent days.

Studying test executions, we should refer to the application lifecycle due to the development of subsequent versions, code modifications or corrections, introduced new functionalities, etc. Hence, we trace negative test results in time (as in Figure 4). Figure 5 presents distribution of faulty test steps related to negative test scenarios over 5 days. The y-axis shows the number of faulty test scenarios, the x-axis shows the percentage of correctly executed test steps within these scenarios. In the case of stabilizing software versions, the bars are moved towards higher percentage values and lower number of test scenarios. Significant values related to newly introduced functionalities. They were identified by correlating commits with added functionalities and the number of failed scenarios. Analysing such profiles, we can perform deeper exploration related to higher values to derive the reasons of these anomalies.

B. EXPLORING TEST REPORTS

Tracing test executions we have introduced some metrics related to three aspects: test coverage, test productivity and result outcome. The test coverage can be correlated with application features, e.g., the number of executed transactions (in total or per transaction class), code coverage. For an illustration, we give statistics of the performed transactions in the regression test suites executed each day in 6-hour sessions. This statistics for subsequent 6 hours over 10 days was as follows (h: a-b; where h denotes hour, a and b denote transaction ranges):

TC (6/10) = {1: 8-12; 2: 8-12; 3: 180-210; 4: 150-162; 5: 410-500; 6: 38-42}

This is some measure of application stresses. The quality of the test-suite (TS) can be assessed by statement or branch coverage, test suite size, application stressing, mutation score, etc. In [43] mutation score (the number of killed mutants) has been extended by TS capability ratio, i.e., the ratio of the number of tests in TS that kill at least one mutant over the total number of tests in TS. This was applied in controlling the reliability of repairs (related to involved regressions). Test productivity we define as the percentage of passed tests, executed test steps, test scenarios, test execution time, etc. In the case of failed test elements (e.g., steps) we can admit their repetition for a specified number of times. False tests may relate to activated faults (errors). In general, we distinguish the following error classes:

1) Application logic errors – incorrect algorithm, not predicted handling of an erroneous situation, not specified behavior.
2) Implementation errors:
   - User interface fault – incorrect graphical display of data, inconsistency between expected and real interaction behavior,
   - Calculation faults – wrong program coding, inaccurate calculations,
   - Thread management errors – synchronization problems.
3) Environment configuration errors – lacking or incorrect environment or authentication variables, incorrect project profile build, incorrect IP or URL addresses.
4) Test data configuration errors – e.g., lacking account of senders or receivers.
5) Test errors:
   - test scenario logic error – incorrect test steps, wrong sequence of executions, inconsistency with application specification,
   - test scenario implementation fault – test code error, test library error, technology instability (e.g., related to time dependencies),
   - test configuration errors – incorrect test data, or environmental variables.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

Analyzing test execution, we should correlate it with the application lifecycle taking into account appearance of subsequent versions, error corrections, modifications, updates, etc., which impact test runs.

The distribution of detected errors in the project lifecycle may differ, as well as the required effort to fix them. Typically, the distribution of the error classes 1-5 was 10%, 30%, 20%, 35% and 5%, respectively. Most of application logic errors (class 1) appeared in the initial phase of the project and for test scenarios with big branch coverage (related to diverse configurations and releases). The implementation faults (class 2) are rarely announced during exploitation, they are detected with sophisticated test scenarios. These scenarios cover diverse aspects of the tested system, e.g., uncommon configuration and operation conditions which even rarely appear in production. The environment configuration errors (class 3) are strongly coupled with introducing new releases including applications redeployments, running migration tools, choosing right environment profiles (e.g., resulting from some deficiencies in documentation), etc. Test data configuration errors (class 4) dominated and increased with higher complexity of system functions. This resulted from test scenarios becoming more complex and involving more modules, parameters, and test data (difficult to grasp in short time). Quite often the test data is elaborated in relevance to the tested code which can be incorrect. Test errors (class 5) have a quite short lifecycle as most of them are eliminated during the test development, moreover they are quite easy to fix. The presented error taxonomy and the relevant investigation triggered fruitful discussion within the project team meetings and provided useful hints for possible improvements.

Sometimes negative tests relate to timeout crossing, due to waiting for some expected events (e.g., waiting for generation of an invoice by a system component). Timeouts can trigger repetitions of the test, changing timeout limit or modifying test parameters. More difficult is handling errors resulting from framework incompatibilities with used libraries or technology changes in the tested application.

The developed test framework (TF - Section III) provides the capability of more detailed test result statistics in relevance to test steps, scenarios and runs correlated with specified functionalities. It is illustrated in Table 6 ordered according to the number of negative tests. It assures hierarchical access to test reports from upper to lower levels (test steps). We can generate cross-sectional statistics, e.g., distribution in time of negative test results for specified test runs, scenarios. We can create heat map with y-axis showing test identifier (e.g., test case) and x axis alert type or exception, the entries can specify the number of negative tests (of specified types). Basing on this we can provide a retrospective view on the test efficiency (detectability and diagnosability), performed updates, modifications or deletions of test cases in the used test sets. This can be done in relevance to the tested system lifecycle (revisions, etc.).

The test result repository comprises valuable data which can be explored and analyzed with relevant metrics targeted at diverse aspects, e.g.:

- Tracing historical changes of test sets (at different levels, groups, test cases and steps) due to application life cycle development and maintenance phase.
- Test modifications resulting from detected faults in tests,
- Error detection score per tests in time.
- Correlation of code correction triggered by a test A with failed tests after correction, this may provide information to reduce the needed tests in regression testing after corrections of specified application areas.
- Test modification analysis in software life cycle, may provide the range of changes, e.g., on the level of steps, test cases or test oracle specification.
- Identification of newly introduced tests due to errors reported by users and not detected by the existing set of tests.
- Correlating executed tests with the relevant issue handling time, the number of exchanged comments, time needed for analysis or resolution, etc. It allows to assess diagnosability features. Here, we can add suggestions on how the diagnosis could be improved, what kind of information was lacking as helpful (feedback from issue analyzers and code correctors).
- Impact of application updates on the test management. For example, identifying the impact of test A results on other tests (failed correlation steps, cases, groups).

It is interesting to trace the impact of introduced code changes on the test outcome, e.g., what is the manifestation of tests targeted at the introduced code changes, was there any impact on tests for other functionalities. This allows us to identify functional/code interrelationship. Having identified such dependencies, it is easier to identify regression subtests targeted at specified functionalities. Historical data of performed test facilitates this process, this needs referring to issue and commit repositories (Section VI).

Testing processes are correlated with the project development, and we can observe their mutual impact in time. Hence, it is reasonable to monitor the distribution of detected error classes (Section V) and correlate them with introduced new functionalities, used sets of test scenarios and actors involved in these processes. Staff fluctuations may also have a significant impact. Test suit sets are improved, corrected or adapted to the progress of the project live cycle. Hence, tracing test relevant software repositories (issue and code version control) is also useful to assess the efficiency of the test processes. The developed test framework (TF) is helpful in this process. We can derive test suit fluctuations for selected time periods T_p in relevance to four classes of test suits:

- TS_S - set of stable test scenarios, i.e., such which were used constantly within period T_p.
- TS_M - set of modified test scenarios, i.e., changed at some point within T_p (e.g., adaptation to functionality or

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

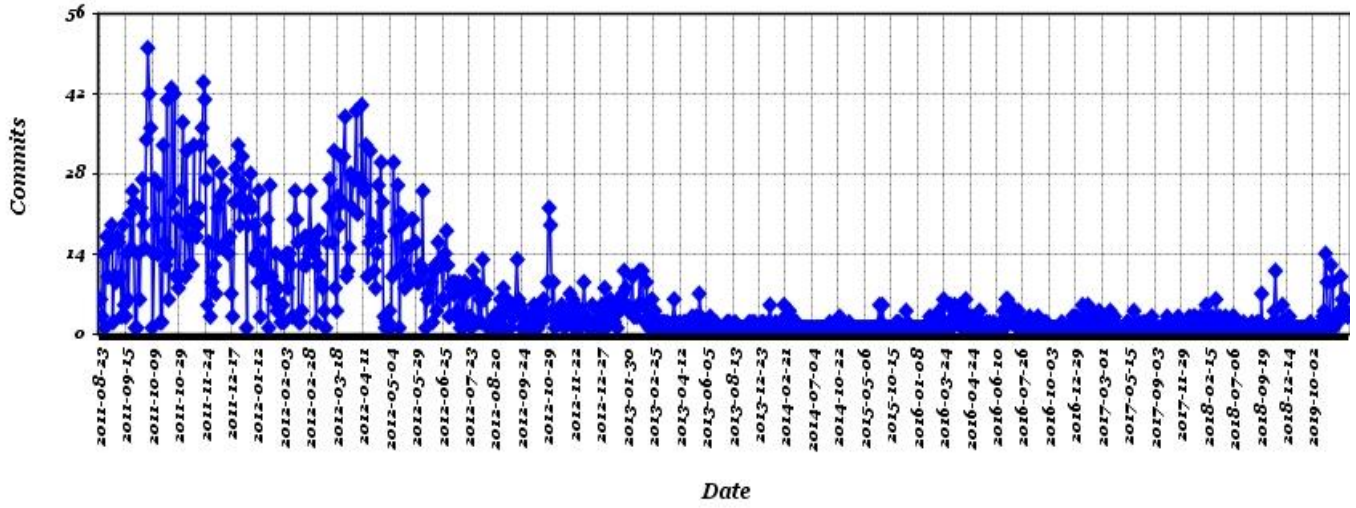

FIGURE 6. **Time distribution of performed commits (average values per day).**

configuration changes of the tested project, improved functionality description, remarks provided by project end users).

- TS_R - set of removed test scenarios, i.e., used up to some point within Tp and then abandoned (e.g., due to low efficiency, significant code or configuration changes, lack of maintenance due to staff fluctuation).
- TS_A - set of added test scenarios, i.e., appearing after some point within Tp (e.g., covering added new functionality, replacing previously used test scenarios).

Statistics of these classes can be derived using the developed framework by referring to issue and software version control entries of tests. Their interpretation needs referring to software repositories of the tested project. Typically, for Tp = 1 month, the distribution of test classes TS_S, TS_M, TS_R and TS_A was: 75-85%, 15-20%, 0-2% and 0-4%, respectively.

VI. CODE AND TEST DEPENDANCIES

In the software life cycle the code is changed due to releases of subsequent versions, fixing detected errors, functional modifications/extensions, performance improvements, etc. This has a significant impact on test processes. Hence, it is reasonable to analyze software version control (SVC) repositories in accordance with issue tracking system (Jira). We have quite rich experience with such analysis presented in [7,25] in relation to open source and commercial projects. This resulted in the original concept of the problem handling graph (PHG) and software development monitoring, which is also useful in tracing links with testing processes.

A. EXPLORING REPOSITORY DATA

Analyzing software lifecycle repositories and the distribution of performed commits including the scope of the relevant code changes (modification, deletion, addition), we focus on four aspects triggered by the negative test results:

1) Tracing handling issue reports triggered by negative tests.
2) Identifying reported bugs (e.g., by users) which were skipped by tests and filling this gap by improving or adding supplementary test scenarios.
3) Tracing used test sets in relevance to software development progress, deployed releases and user reports (profiles of test sets TS_S, TS_M, TS_R and TS_A).
4) Developers autocorrections or code refactoring.

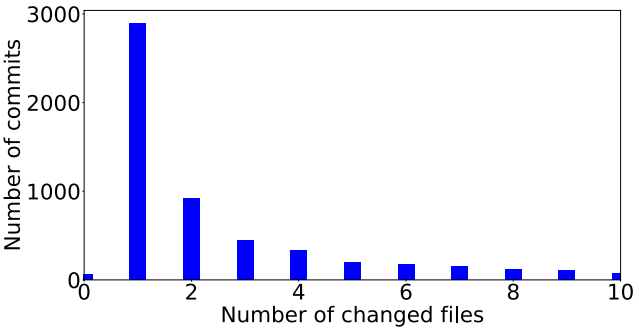

FIGURE 7. **Distribution of file changes in commits**

Figure 6 presents distribution of registered commits within the scope of 10-year project lifecycle. The first 2 years relate to development phase (75% of created files), subsequent years correspond to maintenance and improvement period (25% of files). In the initial phase a significant increase of newly introduced tests was observed, while in the stable phase this was about 10-20% per year. The registered commits resulted in changes, deletions, and addition of new files in the range: 1-4, 2-10, and 0.5-2, respectively. The distribution of commits involving specified number of changed files is given in Figure 7. Commits with single file changes dominate, this confirms efficient problem handling processes (involving testing). Each file is changed on average 3.83 times. However, some of the issues trigger high numbers of changes (exceeding 10), typically, this occurs in relation to new functionalities.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

Sometimes a simple modification appears in many files, and it is classified as a multiple file change, e.g., caused by a new name of a file updated in many places (hence maximum 296 file changes were reported).

Each commit is triggered by a registered issue in Jira repository. Figure 8 shows the distribution of the number of performed commits correlated with a single issue. The x-axis specifies subsequent issue ids, the y-axis shows the average number of relevant commits over aggregated subsequent 100 issue chunks. For each bar we give also the minimal and maximal values observed within the chunk (min/max). The first and second quartile for the first, second and 5th chunks were: (1,6), (1,3) and (1,4), respectively. For the remaining ones it was (1,2). The analyzed issues triggered 1-58 commits; on average a single issue is correlated with 2.33 commits. A single issue usually triggers a single commit, but it may also result in a sequence of commits which at first glance seems to be strange. This results from the fact of involving a group of issue resolving programmers (big problems) as well as from partial resolution of the problem. Generating a partial commit is useful to not block other project contributors for longer time, they can proceed their work. A single issue resulted in maximum 58 commits (relevant to introducing a significant range of new functionalities).

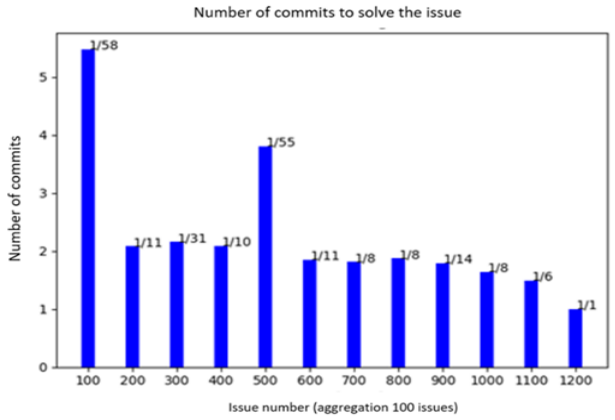

FIGURE 8. **Distribution of performed commits in relevance to reported issues.**

SVC repository gives some view on the cost of performed code changes. Many of them are the consequence of negative tests which triggered appropriate issues. We can trace handling these issues in Jira repository. Typically, they are handled according to the following scheme defined by the sequence of issue state processing (PHG graph):

New → Workaround → open → in analysis → accepted → in_progress → in review → resolved → closed

An issue in *Workaround* or in *in-analysis* states can be considered as irrelevant and transferred to *reject* state (not shown branch). The *accepted* state corresponds to positive detection and diagnosis of the problem. The time needed to attain this state in a large extent depends upon the test diagnosis quality. Nevertheless, it can also depend upon available resources or the issue handling organization. Some indirect metrics of diagnosis capability are the number and the size of exchanged comments included in the considered issue. The issue handling process can also depend on external actors correlated with the project. For example, staying in *in-progress* state is conditioned by human reactions. They refer to activities (including comment exchanges) of project users, vendors (external provider of some software modules) or technical support. More complex problem handling paths (with several branches and loops) we have observed in other projects [7,25]

B. CORRELATING REPOSITORY DATA

Correlating issues and commits with test reports needs some effort, due to often neglected references to negative tests in repository entry descriptions. In such case the correlation can base on context analysis supported with text mining. Assessing report quality in this aspect we calculate the ratio of entries with direct test references, those with needed deeper analysis and ambiguous ones. In the analyzed project only 5-20% reported issues in Jira comprised explicit references to relevant test scenarios. Another view on this problem is checking the size of the entry descriptions. The commit description provides some information on its reason, this can also facilitate correlation with relevant issue or test reports, however many programmers neglect this. For an illustration we give derived statistics of commit descriptions. For 6000 commits (23000 file changes within 5 years) we found on average 9.7 words (minimum 0 and maximum 206 words) in the description fields. However, we observed an increasing trend in time (average values 5-12 words) and interquartile ranges for consecutive 500 commit groups stretched from 13 to 37 words. Commits related to new functionalities showed longer description fields.

Typically, a logging statement in the code contains static and dynamic part. The static part is a fixed character string and variable part is determined during the code execution of the run time. Associating development knowledge to test results and generated logs is helpful in diagnostic processes. Interactions between developers and users provide additional information in repositories. The report of an issue comprises its description, resolution and development discussion about it (comments), history of processing stages. An issue which is related to event logs may be helpful in its interpretation (rationale of log), moreover it can be correlated with log line in the source code which generated the log entry. A code commit describes the changes of the code and other corresponding (associated) information, a logging statement may provide the meaning of the code line, etc. Hence, we can trace the following associations: *log line-log statement (file, method location, code comment) – code commit – issue report*,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

in addition we can trace emails from users on the registered logs.

Log repositories form a rich source of information (knowledge) on testing, neglected in the literature. Systematic processing of a wide scope repositories leads to improvement of testing and diagnosis. Dealing with diverse types of repositories, we trace their interactions and information associations. Code corrections or new functions may trigger inclusion or modification of relevant log statements in the code, this results in log profile changes or updates during the testing processes. In addition, we can investigate user reports and correlate them with possible test gaps.

Correlating issue tracking and version control repositories with testing processes needs appropriate filtering. Test reports should comprise the specification of the tested software version (including the merged commits); hence we can trace issues to correlate them with issue requests related to new functionalities, corrections resulting from previous negative tests, corrections triggered by project users, etc. In the case of confirmed missed errors, it is important to identify the reasons of their skipping by regression tests and produce requests for test set updates. Tracing the handling process (depicted by PHG graph) of recorded issues triggered by the test outcome, we can check the exchange of messages, comments which can reflect the precision of bug specification and localization. We can check the time between the issue registration and problem allocation as well as the identified decision looping. This can provide some indirect metric of test diagnosability. However, they can be obfuscated by project actors' allocation policies, their capabilities or practical competence and workload load. We can derive positive correlations of project actors with test subsets (successful and fast handling). In the case of projects with recommendation repositories filled by users [37] we can correlate them with project functionalities and relevant test suites to check why there were not detected during development/maintenance phase, and specify, if needed, test drawbacks.

VII. DISCUSSION

The developed test framework (TF – Section III) allowed us to extend the observation perspective of test results over a wide scope of software repositories. It provides the capability of extracting detailed and important features of test execution. This is useful to assess test diagnostic efficiency and reveal imperfections in the context of the project development and maintenance by correlating test reports with issue and commit repositories. Insufficient accuracy and negligence in reporting (automatic or manual) creates problems in extracting appropriate data, hence discovering some time and semantic contexts is helpful. In relevance to this we showed the usefulness of deriving structural and semantic features of dictionaries used in repositories, keywords, and profiles (in time and value - heatmaps) of associated variables or other statistics. Log dictionary and keywords can be investigated (in long time perspective) to determine their impact on diagnostic capabilities and point out possible improvements. (Section IV).

The presented approach significantly expands the observation space of assessment processes related to regression, acceptance, and integration tests (Section V). This exceeds the capability of common approaches of unit testing based on finding links between test cases and related production classes ([42] and references therein). Diagnostic capabilities can be enhanced by correlating test reports with other software repositories (e.g., issue tracking and version control). Test set optimization and upgrades need tracing test impact in these repositories in time perspective (Section VI). Test diagnosability and efficiency can be evaluated by correlating detected problems (by tests) with relevant handling paths (PHG graph) and software fix ranges. Identified long or complex handling paths (e.g., involving state looping, comment exchanges [25]) we can drill down test outcome deficiencies (test reports and generated event logs) to reveal possible improvements.

We can increase not only test diagnosability, but the supplementary advantage of our approach is also identifying and controlling the consistency of tests with project/environment changes or updates during the life cycle. For this we explore test set profiles (TS_S, TS_M, TS_R and TS_A), distribution of test results (at test scenario and test step levels, e.g., Table 4) and detected defect classes, skipped errors, etc. The presented monitoring process covers project and test lifecycles (including their mutual interactions) taking into account relevant issue and version control repositories. Tracing detected and skipped defect classes for test runs (application logic, implementation, configuration, test flaws) facilitates to identify and counteract suspicious trends (Section V B).

Another issue is qualification of test importance, test strength and enhanced test outcome taxonomy. Depending upon the test targets we can attribute test importance correlated with potential bug severity levels (critical, major, minor, cosmetic, etc.). More sophisticated tests can provide multilevel outcomes instead of binary results (passed, non-passed steps). Hence, the outcome of non-passed tests can be further précised showing detailed deficiency specification which can be categorized according to bug criticality. This can be combined with tracing the progress of handling registered negative test results (in issue repository) in correlation with the attributed issue state and relevant impact on code commits (version control repository). Here, we can distinguish issues disqualified due to non-possible reactivating the fault, rejected due to negligent importance, postponed because of the previewed future function replacement/modification. On the other hand, we have tests triggering many bugs which usually are considered as the most effective and attributed higher priority. Monitoring issue handling we should link them with relevant tests and trace the resolution progress, e.g., rejection decisions, time needed for diagnosis, number of correlated comment exchanges, scope of triggered commits (number of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3136138, IEEE Access

identified bugs, number of changed files or code lines). This provides data for fine-grained test outcome categorization (compare specifications in Section V), which can be aggregated in a longer time perspective to identify behavioral trends. Many postponed, rejected minor or negligible faults can cumulate in time resulting in complex problems or hide detection of other important problems. In another e-commerce project, developed in SCRUM technology with weekly iteration periods (sprints), this problem appeared with a delay of 5-10 months. Hence, it is useful to control the range of piling not resolved issues and perform periodical revisions.

The presented methodology of analyzing test processes and their interaction with project development is based on holistic investigation of wide scope and detailed features derived from diverse software repositories. This involves tracing correlations between reports in repositories. In practice, this process can be simplified and more effective by improving report descriptions. The proposed statistics and diverse metrics are helpful to monitor report quality and provide feedback to actors involved in project development/testing (Sections V and VI).

The presented studies based on our experience with monitoring several software projects, however the developed test analysis methodology was referred to a single commercial project systematically upgraded and maintained for many years. It assured access to rich software repositories. Hence, we include some comments on validity threats of our investigation.

External validity. The presented investigation was performed on data collected by one industrial partner which accepted installation of our plugins into the used test infrastructure (comprising *cucumber*, etc.). The derived test features and relevant software repositories related to a specific software project developed with assumed technology in the company. Nevertheless, the gained experience and assessment methodology can be considered as sufficiently representative for many medium size companies and transaction-oriented applications. Moreover, the presented metrics, analysis methodology can be generalized for a wider scope of projects and development technologies. Retrospective view on testing from software repositories perspectives can show lacking data in repositories, drawbacks of testing processes which can impact development improvements.

We have confronted our results with the experience of testers in two other companies targeted at complex e-commerce and disc controller software. Tests were managed by proprietary software with similar capability to Cucumber. Regression tests (test cases with specified steps) were executed in weekly and several day periods, and they involved many hours in the night. Test reports were synthetic at functional level, sometimes supported with screen shots. Event and other software repositories were not correlated directly with the test reports. This lacking feature is available in our test framework (TF) including the analytical component. It has been assessed by interviewed project testers as promising for investigating test properties in a wider data context and longer time perspective observation. They found it helpful in tracing such problems as faulty and obsolete tests (causing losses in testing time), sources of skipping faults reported by product customers using their own tests. Another issue relates to faults of purchased software modules from other providers, which usually are considered as reliable within a long period of time. However, they showed some problems with delay, e.g., in the context of added or improved functionalities, changed environment. In the considered projects it was observed that a relatively small percent of tests (~20%) reveal most faults (60 - 80%), which can be considered in test optimization, e.g. appropriate execution ordering (prioritization). The presented problems can be effectively studied using our test framework by correlating data from diverse repositories.

Internal validity. The research goal was verified on the available data sets and using the developed plugins to the available test support tools and software repositories. Unfortunately, some data were screened due to their sensitivity touching the developers and users (customers). Nevertheless, this was not significant restriction in our studies, which have been also positively accepted by the industrial partner.

Construct validity. The used test management infrastructure is quite universal. The developed plugin components integrate collection, correlation, and analysis of monitored data. They can be adapted to other test/monitor platforms and software development technologies. Depending upon the organization of repositories we face the problem of getting access to relevant repositories (sometimes managed by different groups in the company or outsourced), selecting test relevant data, crossing data confidentiality restrictions (anonymization preprocessing).

VIII. CONCLUSION

We have analyzed a wide scope of collected data during testing a complex industrial project. This study proved the usefulness of software repositories in assessing and improving software test processes. Test report interpretation is enhanced by correlating registered entries with application logs, which needed adaptation of log parsing to their specificity including structural and semantic features of used words and phrases (n-grams). This is supported with a hierarchical text mining analysis targeted at used word classes (dictionary), keywords and associated variable parts. The presented analysis in short and long-time perspectives refers to issue and version control repositories. It provides an insight on test set consistency with the application and points outs needed adjustments during the system lifecycle. It is based on extensive investigation of a wide scope of test and development features complemented with specified statistical metrics and profiles.

The introduced metrics and statistics can positively affect test quality progress in time, which is important in projects with long live perspective (systematically introduced updates

and new versions of the tested project and system environment). The gained experience revealed the necessary improvements in quality and the range of registered reports (automatically or manually). Our future research will be directed towards deeper empirical studies, in particular tracing the impact of tests on created issues and relevant handling times. This will result in periodical revisions of testing processes and data repositories considering context changes. Another issue is extending our approach on performance tests taking into account supplementary resource usage and timing monitors.

REFERENCES

[1] V. Garousi, M. Felderer, F. N. Kilicaslan , "A survey on software testability", *Information and Software Technology* 108, 2019, pp. 35-64, DOI: · 10.1016/j.infsof.2018.12.003

[2] P. C. Jorgensen, *Software testing, a craftsman's approach,* Taylor and Francis Group, 2007.

[3] A. Perez, R. Abreu, A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches", in *Proc. IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, 1558-1225, DOI: 10.1109/ICSE.2017.66 .

[4] V. Antinyan, J. Derehag, A. Sandberg, M. Staron, "Mythical unit test coverage", *IEEE Software* 35(3):73-79, 2018, DOI: · 10.1109/MS.2017.3281318.

[5] C. Smidts, Ch. Mutha, M. Rodriguez, M. J. Gerber, "Software testing with an operational profile: OP definition", *ACM Computing Surveys*, February 2014, 46(3), DOI: · 10.1145/2518106.

[6] G. Tebes, D. Peppino, P. Becker, G. Matturro, M. Solari, "Analyzing and documenting the systematic review results of software testing ontologies", *Information and Software Technology*, Volume 123, July 2020, 106298.

[7] J. Sosnowski, B. Dobrzyński, P. Janczarek, "Analysing problem handling schemes in software projects", *Information and Software Technology* Volume 91, November 2017, pp.56-71, https://doi.org/10.1016/j.infsof.2017.06.006.

[8] S. Martinez-Fernandez, A. M. Vollmer, A. Jedlitschka, X. Franch, L. Lopez, P. Ram, et al., "Continuously assessing and improving software quality with software analytics tools: A case study", *IEEE Access*, vol. 7, pp. 68219-68239, 2019.

[9] C. Jeanderson, A. Maur´ıcio i A. Deursen, „Contemporary software monitoring: A systematic literature review," CoRR abs/1912.05878, 2019.

[10] S. Almugrin, W. Albattah, A. Melton, "Using indirect coupling metrics to predict package maintainability and testability", *Journal of Systems and Software*, Volume 121, November 2016, pp. 298-310.

[11] K. Gao, „Simulated software testing process and its optimization considering heterogeneous debuggers and release time", *IEEE Access* vol. 9, 3849-59, 2021.

[12] R. Ibrahim, A. Aminuddin, B. Amin, S. Jamel, J. A. Wahab, "A software testing tool for generation of test cases automatically", *International Journal of Engineering Trends and Technology* 68(7):8-12, July 2020, DOI: 10.14445/22315381/IJETT-V68I7P202S.

[13] D. Graham, M. Fewster, *Experience of test automation, case studies of test automation*, Pearson Education, Inc. (2012), ISBN: 9780321754066.

[14] M. Nosrati, H. Haghighi, M. V. Asl, "Test data generation using genetic programming", *Information and Software Technology*, Volume 130, February 2021, 106446.

[15] P. Saha, U. Kanewala, "Fault detection effectiveness of source test case generation strategies for metamorphic testing", in *Proc. of MET '18: Proceedings of the 3rd International Workshop on Metamorphic Testing*, May 2018, pp. 2–9, https://doi.org/10.1145/3193977.3193982.

[16] J. Chen, W. Shang, E. Shihab, "PerfJIT: Test-level just-in-time prediction for performance regression introducing commits", *Journal of Latex Class Files, IEEE Transactions on Software Engineering*, vol. 14, no. 8, August 2019, DOI 10.1109/TSE.2020.3023955.

[17] Z. Ding, J. Chen, W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet?" In. *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering*, June 2020, pp. 1435–1446, https://doi.org/10.1145/3377811.3380351.

[18] G. Kumar, P. K. Bhatia, "Software testing optimization through test suite reduction using fuzzy clustering". *CSI Trans. ICT* 1, 3, 2013, pp. 253–260. DOI:10.1007/s40012-013-0023-3.

[19] B. Miranda and A. Bertolino, "Scope-aided test prioritization, selection and minimization for software reuse", *J. Syst. Softw.* 131, 2017, pp. 528–549. DOI:10.1016/j.jss.2016.06.058.

[20] M. Reider, S. Magnus, and J. Krause, "Feature-based testing by using model synthesis, test generation and parameterizable test prioritization", in *Proc. of the IEEE 11th International Conference on Software Testing, Verification and Validation Workshops*, 2018, pp. 130–137. DOI:10.1109/ICSTW.2018.00041.

[21] Bluemke, I., "Software testing effort estimation and related problems: A systematic literature review, *ACM Computing Surveys*, Volume 54, Issue 3, June 2021, Article No.: 53, pp. 1–38, https://doi.org/10.1145/3442694.

[22] A. Perez, R. Abreu, A. van Deursen, "A theoretical and empirical analysis of program spectra diagnosability," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 412-431, 1 Feb. 2021, doi: 10.1109/TSE.2019.2895640.

[23] Y. Meng, G. Gay and M. Whalen, "Ensuring the observability of structural test obligations," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 748-772, 1 July 2020, doi: 10.1109/TSE.2018.2869146.

[24] M. Nayrolles, A. Hamou-Lhadj, "Towards a classification of bugs to facilitate software maintainability tasks", in *Proc. of the 1st International Workshop on Software Qualities and their Dependencies*, May 2018, pp. 25–32 https://doi.org/10.1145/3194095.3194101.

[25] J. Polaczek, J. Sosnowski, „Exploring the software repositories of embedded systems: An industrial experience", *Information and Software Technology*, vol. 131, March 2021, 106489, DOI:10.1016/j.infsof.2020.106489.

[26] R. Karim, A. Ihara, X. Yang, H. Iida, K. Marsumoto, "Understanding key features of high-impact bug report", in *Proc. of 8th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pp. 53-58, DOI:10.1109/IWESEP.2017.17Corpus ID: 6372261Tokyo, 2017.

[27] A. Sarkar, P. C. Rigby and B. Bartalos, "Improving bug triaging with high confidence predictions at Ericsson, in *Proc. of IEEE International*

Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 81-91, doi: 10.1109/ICSME.2019.00018.

[28] H. Li, C. Tse-Hsun, S. Weiyi, A. Hassan, „Studying software logging using topic models," *Empirical Software Engineering*, 2018, 23, pp. 2655–2694(2018), https://doi.org/10.1007/s10664-018-9595-8.

[29] Z. Li, Z. Jiang, X. Chen, K. Cao, Q. Gu, "Laprob: A label propagation-based software bug localization method", *Information and Software Technology*, Vol. 130, February 2021, 106410 , https://doi.org/10.1016/j.infsof.2020.106410.

[30] M. Kubacki i J. Sosnowski, „Holistic processing and exploring event logs," in Romanovsky A., Troubitsyna E. (eds) *Software Engineering for Resilient Systems. SERENE 2017*. Lecture Notes in Computer Science, 10479, 183-199, 2017.

[31] J. Zhou, S. Hey, j. Liuz, P. Hex, Q. Xiek, Z. Zhengz, M. R. Lyu , "Tools and benchmarks for automated log parsing", in *Proc. of the 41st International Conference on Software Engineering: Software Engineering in Practice*, May 2019, pp. 121–130, https://doi.org/10.1109/ICSE-SEIP.2019.00021.

[32] P. He, J. Zhu, S. He, J. Li, M. R. Lyu, "Towards automated log parsing for large scale log data analysis", *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, November/December 2018, pp. 931-944.

[33] B. Zhang, H. Zhang, P. Moscato, P. Zhang, "Anomaly detection via mining numerical workflow relations from logs", June 2020, TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.12570926.v2.

[34] W. Shang, M. Nagappan, A. E. Hassan, Z. M. Jiang: "Understanding log lines using development knowledge", in *Proc. 30th IEEE International Conference on Software Maintenance and Evolution*, Victoria, BC, Canada, September 29 - October 3, 2014. IEEE Computer Society 2014, ISBN 978-0-7695-5303-0.

[35] A. Roman, M. Mnich, "Test-driven development with mutation testing – an experimental study", *Software Quality Journal* 29, 1–38 (2021). https://doi.org/10.1007/s11219-020-09534-x.

[36] J. Yan, H. Zhou, X. Deng, P. Wang, R. Yan, J. Zhang, "Efficient testing of GUI applications by event sequence reduction", *Science of Computer Programming*, Volume 201, 1 January 2021, 102522 https://doi.org/10.1016/j.scico.2020.102522.

[37] M. Jeleński, J. Sosnowski, "Mobile application testing and assessment" in: *Advances in Intelligent Systems and Computing*, vol 1173. Springer, 2020, pp. 283-292, https://doi.org/10.1007/978-3-030-48256-5_28.

[38] C. M. Rosenberg, L. Moonen, „Spectrum-based log diagnosis," in *Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, **ISBN:** 978-1-4503-7580-1, October 2020, Article No.: 18, pp. 1–12, https://doi.org/10.1145/3382494.3410684.

[39] S. Pearson et al., "Evaluating and improving fault localization," in *Proc. Of IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, 2017, pp. 609-620, doi: 10.1109/ICSE.2017.62.

[40] Z. Cui, M. Jia, X. Chen, L. Zheng and X. Liu, "Improving software fault localization by combining spectrum and mutation," *IEEE Access*, vol. 8, pp. 172296-172307, 2020, doi: 10.1109/ACCESS.2020.3025460.

[41] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, L. Zhang, „An empirical study of fault localization families and their combinations", *IEEE Transactions on Software Engineering*, vol. 47, issue 2, February 1 2021, pp.332-347 https://doi.org/10.1109/TSE.2019.2892102.

[42] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, A. Roychoudhury. "A correlation study between automated program repair and test-suite metrics", *Empirical Soft. Eng.* 2018 23:2948-2979, DOI https;//doi.org/10.1007/s10664-017-9552-y.

[43] N. Ajawabrah, T. Gergely; S. Misra; L. Fernandez-Sanz, „Automated recovery and visualization of test to code traceability links, an evaluation", *IEEE Access* vol. 9, 2021 pp. 40111 – 40123, DOI: 10.1109/ACCESS.2021.3063158.

[44] D. Ateşoğulları, A. Mishra, "Automation testing tools: a comparative view", *International Journal on Information Technologies & Security*, no. 4, vol. 12, 2020 pp. 63 – 76.

MYKHAILO LASYNSKYI received the B.Eng. degree in 2017 from the National Aviation University of Ukraine in Kyiv and M.Sc. degree in computer science from Warsaw University of Technology in Poland. Currently he cooperates with the University and is working as a software engineer in a commercial company developing transactional systems for the financial sector. His research and professional interests include software engineering, project development and testing.

JANUSZ SOSNOWSKI (Senior Member, IEEE) received M.Sc. and Ph.D. degrees in computer science from Warsaw University of Technology in Poland. He is currently a Professor with the Institute of Computer Science of this University. He has published more than 150 research articles in international refereed conference proceedings or journals and several books. He has been on many program committees of international conferences and reviewer for diverse scientific publications worldwide. His research interests include dependable computing, system testing, diagnosis, performance, and fault handling issues. He has managed and participated in many scientific and industrial projects.

.