Navigating Information Overload Caused by Automated Testing – A Clustering Approach in Multi-Branch Development

Erman, Nicklas; Tufvesson, Vanja; Borg, Markus; Ardö, Anders; Runeson, Per

Published in:
2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings

DOI:
10.1109/ICST.2015.7102596

2015

Navigating Information Overload Caused by Automated Testing – A Clustering Approach in Multi-Branch Development

Nicklas Erman
Qlik
Lund, Sweden
nicklas.erman@qlik.com

Vanja Tufvesson
Accenture
Copenhagen, Denmark
vanja.tufvesson@accenture.com

Markus Borg, Anders Ardö and Per Runeson
Lund University, Sweden
markus.borg, per.runeson@cs.lth.se
anders.ardo@eit.lth.se

Abstract

Background. *Test automation is a widely used technique to increase the efficiency of software testing. However, executing more test cases increases the effort required to analyze test results. At Qlik, automated tests run nightly for up to 20 development branches, each containing thousands of test cases, resulting in information overload.* ***Aim.*** *We therefore develop a tool that supports the analysis of test results.* ***Method.*** *We create* NIOCAT*, a tool that clusters similar test case failures, to help the analyst identify underlying causes. To evaluate the tool, experiments on manually created subsets of failed test cases representing different use cases are conducted, and a focus group meeting is held with test analysts at Qlik.* ***Results.*** *The case study shows that* NIOCAT *creates accurate clusters, in line with analyses performed by human analysts. Further, the potential time-savings of our approach is confirmed by the participants in the focus group.* ***Conclusions.*** NIOCAT *provides a feasible complement to current automated testing practices at Qlik by reducing information overload.*

Index Terms

Software testing, test automation, test result analysis, clustering, case study.

1. Introduction

When trying to improve software test efficiency, test automation is often brought forward as a key solution [1], [2]. However, while automated testing (auto testing) provides the benefits of reducing manual testing, minimizing human error, and enabling a higher testing frequency [3, p. 466], new challenges are introduced. With higher testing frequency the volume of test results increases drastically. Consequently, there is a need for tools to navigate the potential information overload [4].

Qlik[1], a software company in the business intelligence domain, has adopted auto testing to save time, improve test coverage and to enable development of new features in parallel, while assuring a high quality product. At Qlik, automated tests (autotests) run every night on multiple source code branches using Bamboo (see Section 3.2). However, Bamboo simply groups test results based on the test case (TC) names, and it is both difficult and time consuming to manually analyze the large amount of test results.

To support the analysis of test results, Qlik developed PasCal, a tool that clusters TC failures based on the error message generated by the failed TCs (see Section 3.2). However, PasCal still uses a naïve clustering approach: exact matching of the error messages. Moreover, PasCal was not mainly developed to provide an overview of the auto testing, but to automatically generate new bug reports based on TC failures on the main development branch.

Although PasCal is an important first step toward improved analysis of results from autotests, there are still several open challenges. First, there is no efficient way to determine that specific TCs fail on multiple branches. Second, intermittent failures due to variations in the testing environment make the results unreliable, thus triggering re-execution of autotests. Third, concluding that multiple TCs fail because of the same root cause is difficult. All three challenges are amplified by the information overload caused by

1. www.qlik.com

the auto testing, and the test analysts request support.

To improve the overview of test results, we developed NIOCAT, a tool that clusters TC failures from auto testing in multi-branch environments. The clustering goes beyond exact string matching by calculating relative similarities of textual content using the vector space model [5]. Furthermore, we complement the TC name and error message by execution information, in line with previous work [6].

We evaluate the accuracy of NIOCAT in a case study, using three manually constructed scenarios representative for the work of test analysts at Qlik. In the study, we also explore different weighting of textual information and execution information using space-filling experimental design [7]. Also, we qualitatively evaluate NIOCAT through a focus group interview, following the case study methodology proposed by Runeson *et al.* [8].

In this paper, we briefly summarize related work in Section 2 and describe the case company in Section 3. Section 4 jointly presents NIOCAT and its evaluation in Section 4. We present the results in Section 5, and finally conclude the paper in Section 6.

2. Related work

Information overload affects software engineering, as large amounts of formal and informal information is continuously produced and modified. With the growth of information retrieval and tools, their application to the software engineering information overload context is a natural step. Recommendation systems have evolved as a concept to support the navigation through large spaces of software engineering information [9]. Borg *et al.* summarize two decades of work in their recent systematic review of information retrieval for trace recovery [10]. The history covers a lot of work, beginning with the seminal work on requirements engineering by Borillo *et al.* [11], through trace recovery work initiated by Anoniol *et al.* [12]. However, when it comes to application to the information overload in software testing, the list is much shorter. Only 2 out of 79 papers were about test-test or test-defect relations; 2 were about code-test relations and 10 about test-requirements relations [10].

The information retrieval research on test artifacts is mostly related to issue reports, starting with Runeson *et al.*'s duplicate detection [8], followed by replications [6], [13]. However, to our knowledge, there is no previous application of information retrieval techniques to the analysis of test results. Our approach is inspired by the ability to use information retrieval techniques to find duplicate bug reports, i.e. to cluster similar software failures. The work done by Wang *et al.* [6] and Lerch *et al.* [14] showed that execution data combined with natural language outperforms the natural language approach. We choose to focus only on failures found through auto testing, and consequently, the data have slightly different characteristics compared to the bug reports. The text is machine generated and not written by a human reporter. However, we investigate further if execution data, in this case an HTML snippet, can be incorporated into the analysis to improve the accuracy of the clustering of test results, compared to focusing on textual information alone.

3. Description of the Case

Qlik, a company with over 1,900 employees worldwide (July 2014), develops business intelligence[2] solutions for decision support within a wide range of industries, e.g. finance, life science, retail, and telecommunications. The main product, QlikView, has evolved since the company was founded in Lund, Sweden in 1993. More functionality and features have been added and the complexity has grown. The company's next major release is a new product called Qlik Sense, migrating the whole user interface (UI) to web technology. In this paper, Qlik Sense is the software under test (SUT).

3.1. Software Configuration Management

The development of Qlik Sense is divided into several feature teams. To allow teams to develop and maintain the same code base in parallel, a branching strategy is in place where each team has at least one development branch. When a team has successfully completed a development iteration, the new code is delivered to the main branch. Auto tests execute regularly for all active branches. A full system and integration test suite runs nightly for each branch, helping teams to detect regression in the software early. The development branches are kept up to date with the main branch by regularly performing forward merges.

3.2. Toolchain for Auto Testing

Qlik has developed an auto testing framework, Horsie, that drives the software according to TCs specified by scenarios written in structured natural language. Horsie executes the steps specified in the scenarios via

2. Business Intelligence is a set of methodologies and techniques to transform data into useful information for strategical business decisions [15].

the Qlik Sense API. Horsie thus provides an integration between test specification and test execution.

Qlik uses Bamboo[3], a solution for continuous integration by Atlassian, to manage execution of the autotests. Upon finishing a test suite, the test results are made available through Bamboo's web interface. For each branch, the TC failures are grouped based on the exact TC names. We refer to this approach to cluster TC failures as BAMBOO, and later use it for benchmarking against NIOCAT.

Two main artifacts are produced when executing autotests. First, log files from Qlik Sense contain execution information, and if the application crashes, a stack trace. Second, Horsie's log files provide information on what methods were invoked via the Qlik Sense API. If a TC fails, Horsie appends two further pieces of information: 1) an error message provided by the developer of the TC, and 2) an HTML dump of the element in the web UI Horsie interacted with when the TC failed. The HTML dump is a linked list with HTML elements (including all attributes) starting from the top level parent element, (i.e. <HTML>), followed by a list of children down to the specific element.

Another internal tool at Qlik is PasCal, a tool that automates bug reporting of autotest failures on the main branch. Figure 1 presents an overview of the process. If a TC fails, the corresponding Horsie log file contains an error message. PasCal checks if the error message has been encountered before by searching in the bug tracking system using exact string matching. If the error message has not been previously encountered, PasCal automatically submits a new bug report. If the error message has been reported before on the other hand, PasCal attaches the information from the recent TC failure to the already existing bug report. A bug report can thus be considered a group, or a cluster, of TC failures. We refer to this clustering approach as PASCAL, and later use it for benchmarking against NIOCAT.

3.3. Current Challenges in Test Analysis

Several challenges are associated with the analysis of autotest results at Qlik. Three challenges are considered particularly important by the test analysts:

Cross Referencing Failures ("Does this TC fail on multiple branches?"): Each development branch produces test results from auto testing. However, there is no easy way to map what TC failures occurred on which branches. As autotests run nightly for between ten and fifteen branches, the amount of autotest results

3. https://www.atlassian.com/software/bamboo

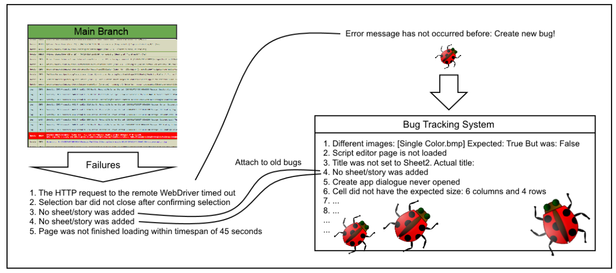


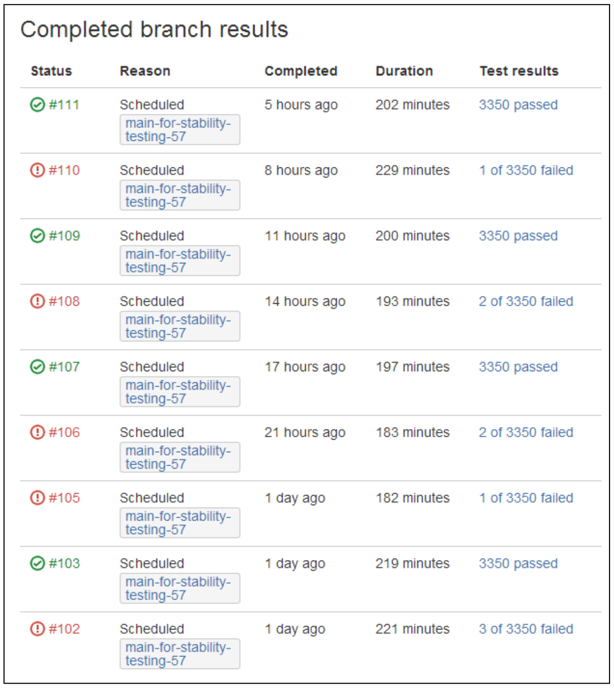
Figure 1. Overview of PasCal, a tool for automated bug reporting. If a new error message is encountered, PasCal creates a new bug report.

Plan	Build	Completed	Tests
--Ver12.00-dev-ft-personal	#9	3 days ago	1 of 3545 failed
--Ver12.00-dev-ft-phoenix	#74	6 hours ago	3544 passed
--Ver12.00-dev-ft-phoenix-QLIK-2877	#5	N/A	No tests found
--Ver12.00-dev-ft-phoenix-QLIK-5237	#4	N/A	2301 passed
--Ver12.00-dev-ft-prometheus	#123	15 hours ago	3550 passed
--Ver12.00-dev-ft-publishing	#23	15 hours ago	4 of 3581 failed
--Ver12.00-dev-ft-ratatosk	#76	6 hours ago	1 of 3582 failed
--Ver12.00-dev-ft-ratatosk-bugs	#42	5 hours ago	11 of 3565 failed
--Ver12.00-dev-ft-ratatosk-codearea	#2	1 week ago	2 of 3314 failed
--Ver12.00-dev-ft-ratatosk-responsive-grid	#20	5 hours ago	1 of 3405 failed
--Ver12.00-dev-ft-responsive-client	#6	2 weeks ago	3304 passed
--Ver12.00-dev-ft-sdk	#53	5 hours ago	2 of 3576 failed
--Ver12.00-dev-ft-thejungle-refactor	#94	11 hours ago	3594 passed
--Ver12.00-dev-ft-theming	Never built		
--Ver12.00-dev-horsie-sdk	#46	11 hours ago	331 of 3720 failed
--Ver12.00-dev-main-for-stability-testing	#159	8 hours ago	3582 passed
--Ver12.00-dev-MergeFromVer11.20	#1	2 weeks ago	5 of 3198 failed
--Ver12.00-dev-QLIK-1476	#6	4 days ago	113 of 3402 failed

Figure 2. Overview presented by Bamboo displaying autotest results from 18 branches. Cross referencing TC failures requires manual navigation into individual branches.

make it practically impossible for a human analyst to get a proper overview of the current problems in the SUT. Figure 2 shows the overview of the test results for multiple branches, as it is presented in Bamboo. Several autotest suites fail, but three of the failing branches have only one TC failure, "Ver12.00-dev-ft-personal", "Ver12.00-dev-ft-ratatosk" and "Ver12.00-dev-ft-ratatosk-responsive-grid". Since Bamboo does not provide a way to cross reference TC failures between the branches, the test analyst must manually navigate into each branch to determine if the same TC has failed on all three branches.

Intermittent Failures ("Is this really a problem?"): Qlik refers to TCs that irregularly fail because of variations in the testing environment (e.g. timing is-



Completed branch results

Status	Reason	Completed	Duration	Test results
#111	Scheduled main-for-stability-testing-57	5 hours ago	202 minutes	3350 passed
#110	Scheduled main-for-stability-testing-57	8 hours ago	229 minutes	1 of 3350 failed
#109	Scheduled main-for-stability-testing-57	11 hours ago	200 minutes	3350 passed
#108	Scheduled main-for-stability-testing-57	14 hours ago	193 minutes	2 of 3350 failed
#107	Scheduled main-for-stability-testing-57	17 hours ago	197 minutes	3350 passed
#106	Scheduled main-for-stability-testing-57	21 hours ago	183 minutes	2 of 3350 failed
#105	Scheduled main-for-stability-testing-57	1 day ago	182 minutes	1 of 3350 failed
#103	Scheduled main-for-stability-testing-57	1 day ago	219 minutes	3350 passed
#102	Scheduled main-for-stability-testing-57	1 day ago	221 minutes	3 of 3350 failed

Figure 3. Overview presented by Bamboo showing results from nine consecutive autotest executions on the same branch. Neither the SUT nor the TCs changed between the test runs, indicating an intermittent failure.

sues caused by unbalanced load of test servers) as "intermittent failures". Figure 3, also a screen shot from Bamboo, shows that consecutive execution of autotests for the branch "main for stability testing" yields different results. Note that neither the SUT nor the TCs have changed between the different runs, but still the test results vary. To determine whether a TC failure is due to a "real" problem in the SUT, or to an intermittent failure, typically the autotests are executed several times. If an overview of all branches with a particular TC failure was available, the time spent re-executing the autotests could be saved.

Root Cause Analysis ("Do these TCs fail for the same reason?"): The same TCs can fail in different ways, i.e. the same TC may fail in different ways in different branches. For example, a six-step TC could fail at any step, but still the same TC name would be presented by Bamboo. To identify differences between the two TC failures, additional information about the failure, such as the error message, has to be taken into account. Similarly, two different TCs might fail in a step that both TCs have in common, e.g. an intial setup step. These problems should not be treated as two different issues, as the common trigger is the setup

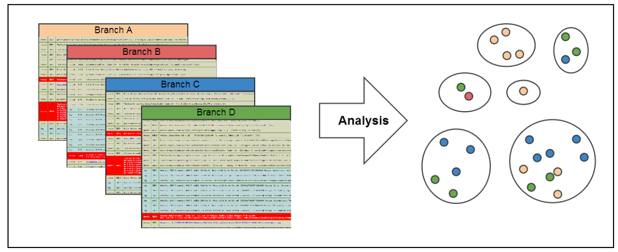


Figure 4. NIOCAT generates a clustering of TC failures based on user selected autotest results.

phase. Again, a naïve comparison using only the TC name would not identify this common root cause. A clustering of all TCs that fail during the same step, i.e. share a common root cause, would support a timely resolution of the issue.

4. Study Design and Solution Approach

As the development and evaluation of NIOCAT were tightly connected, this section contains a joint presentation. First, based on the background and challenges described in Section 3, we state two research questions. Then, the rest of this section presents the details of NIOCAT, the corresponding evaluation, and the major threats to validity.

RQ1 How can clustering of test case failures help test analysts at Qlik navigate the information overload caused by automated testing?

RQ2 Can execution data be used in addition to textual information to improve the clustering of test case failures?

4.1. Solution Approach – NIOCAT

Our approach to support the test analysts at Qlik is to introduce a tool for high-level analysis of autotest results. We name the tool NIOCAT – Navigating Information Overload Caused by Automated Testing. The output from NIOCAT is a clustering of TC failures from a user selected set of autotest results. NIOCAT aims to group similar TCs failures, i.e. each cluster should represent a unique issue in the SUT, containing one or several TC failures.

Figure 4 illustrates how NIOCAT processes autotest results from multiple branches to generate clusters of TC failures. The small circles represent TC failures, whereas larger circles depict TC failures that have been grouped together. To support interactive navigation of the NIOCAT output, we use QlikView (see Section 3) to present the results.

Clustering TC failures provides the test analysts a starting point for further investigation. Test analysts can use NIOCAT in different ways, what use case is supported depends on the analyst's choice of input autotest results. The use case for a development team leader might be to analyze data from test runs within the last seven days, for the team's branch only. A configuration manager on the other hand, might look for a bigger picture and a typical use case could be to analyze the results from the latest test run for each development branch.

4.1.1. Representing a TC Failure. NIOCAT represents a TC failure by three components. Two components consist of natural language text: 1) *The test case name* (TC Name), and 2) *The test case error message* (Message). NIOCAT does not perform any pre-processing of the textual content in TC Name and Message. The third component contains execution information. As the UI of Qlik Sense (the SUT) is based on web technology, Horsie executes TCs using a web browser. Thus, the underlying *HTML* of the elements that Horsie interacts partly reveals the execution state of the Qlik Sense application. NIOCAT extracts the HTML of the element that Horsie interacted with when the TC failed, including all attributes and parent nodes (as described in Section 3.2). The HTML is filtered to contain only element and attribute names as well as attribute values.

4.1.2. Clustering Algorithm. NIOCAT implements a clustering algorithm based on cosine similarity in the vector space model [5]. We define the similarity between a TC failure and a cluster as the average similarity between the new TC failure and all TC failures already in the cluster. The similarity measure between two individual TC failures is calculated as a weighted average of the cosine similarities of the three different components representing a TC failure. By configuring the component weighting (i.e. putting emphasis on either TC Name, Message, or HTML), NIOCAT can be tailored to for the specific use case at hand. Finally, we we defined a threshold for how similar two TC failures should be to appear in the same cluster. A detailed description of the clustering algorithm follows:

1) Let $B = \{b_1, b_2, ..., b_n\}$ be the set of development branches, and $b_iR = \{b_ir_1, b_ir_2, ..., b_ir_l\}$ be their respective test results, where n is the number of branches and l is the number of test runs for branch b_i.
2) Represent each TC failure as a document d that belongs to (only) one cluster c of documents that represents a unique problem with the software product. Let D be the set of all documents representing TC failures for all branches, and let C be the set of all clusters.
3) For each document $d_i \in D$ do
   a) Represent the d_i as three vectors $\vec{d_{i1}}, \vec{d_{i2}}$ and $\vec{d_{i3}}$. Each vector is built using the terms in that component (i.e. TC NAME, Message, and HTML respectively).
   b) Retrieve the clusters $c_j \in C$ that have been created so far. Let the documents $D_j = d_j^1, d_j^2, ...d_j^k$ be all the documents belonging to cluster c_j.
   c) For each pair (d_i, c_j), compute a similarity score $sim(d_i, c_j)$ between the document and the cluster. The score is based on the average similarity score between the document d_i and the documents D_j within the cluster c_j, such that
   $$sim(d_i, c_j) = \frac{\sum_{t=1}^{k} docSim(d_i, d_j^t)}{k} \quad (1)$$
   where
   $$docSim(d_i, d_j^t) = \frac{\sum_{l=1}^{3} w_l \cdot cosSim(d_{il}, d_{jl}^t)}{\sum_{l=1}^{3} w_l}. \quad (2)$$
   The document to document similarity score is based on a weighted average similarity score $cosSim(a, b)$ for each document component and w_l are the weights for the components, respectively. The component similarity $cosSim(d_{il}, d_{jl}^t)$ is computed as the cosine similarity
   $$cosSim(d_{il}, d_{jl}^t) = \frac{\vec{d_{il}} \cdot \vec{d_{jl}^t}}{\|\vec{d_{il}}\| \times \|\vec{d_{jl}^t}\|} \quad (3)$$
   d) Retrieve the cluster c_{max} with the highest value of $sim(d_i, c_j)$. If $sim(d_i, c_j)$ is greater than a predefined threshold T, add d_i to the cluster c_{max}.

4.2. Evaluation Approach – Case Study

We conduct a two phase industrial case study at Qlik. In the first phase, we quantitatively evaluate the accuracy of NIOCAT using three reference clusterings manually created by a test analyst. Using this gold standard, we systematically tuned the parameters for the component weighting in the clustering algorithm. In the second phase, we asked other test analysts at Qlik for their views in a focus group interview.

	RefClust A	RefClust B	RefClust C
Date of first auto test	2014-03-27	2014-03-23	2014-03-28
Sample period	1 night	1 night	≈1 week
#Branches	2	10	1
#Auto test runs	2	10	9
#Test cases	6,696	33,160	26,464
#Test case failures	25	11	61
#Clusters	4	9	13

Table 1. Characteristics of the three reference clusterings: A="compare with main", B="overview all development", and C="analyze single branch".

4.2.1. Reference Clusterings. There is no single archetypical NIOCAT use case, thus we created three reference clusterings (cf. Table 1 RefClust A-C) representing important use cases with different characteristics. A test analyst at Qlik created the reference clusterings, originating from recent auto testing, by manually examining TC failures from three selected use cases. The work involved investigating screen shots, reading log files and interpreting error messages, in line with the test analyst's everyday work tasks.

RefClust A represents a use case of *comparing autotest results from two different branches*; main and development. The same suite of autotests were executed for the two branches during the same night, resulting in 7 and 18 TC failures, respectively. The test analyst identified that the TC failures fall into four distinct problem areas. One of the problems caused ten TC failures across the two branches, while another problem caused only one failure in one branch. Two of the problems caused seven failures each.

RefClust B contains autotest results from one nightly run of auto testing for all development branches, thus representing *an overview use case*. Although more than $30,000$ TCs were executed, only 11 of them failed, reflecting a more mature state of the SUT. In contrast to RefClust A, most TC failures for RefClust B originate from unique problems (9 out of 11).

RefClust C represents a use case of *analyzing a single branch*, containing autotest results from nine consecutive test runs over one week. All autotest results, including 61 TC failures, originate from auto testing of a single development branch. The test analyst classified the TC failures into 13 clusters of various size. The three largest clusters contain $18, 11$ and 8 TC failures, respectively, while the remaining 10 clusters contain fewer than six TC failures.

4.2.2. Evaluation Measures. To evaluate the accuracy of NIOCAT, we compare different clusterings generated by NIOCAT with corresponding reference clusterings (i.e. gold standards) for three selected use cases. As recommended in previous clustering research, we use Adjusted Rand Index (ARI) as a measure for evaluating different partitions of the same data set [16], [17]. ARI between a reference clustering and clusters generated by NIOCAT gives the percentage of correct decisions, calculated pairwise as described in Appendix A. We use ARI to enable benchmarking [18] of both different NIOCAT configurations and the two baselines BAMBOO and PASCAL (described in Section 3.2).

4.2.3. Component Weight Tuning. Four parameters configure the NIOCAT clustering algorithm, the weight of each component in the similarity calculation (TC Name, Message, and HTML), and the similarity threshold (T). To identify a feasible parameter setting for NIOCAT, we systematically evaluated different settings using a uniform space-filling experimental design [7].

We calculated the ARI for the NIOCAT clusterings of RefClust A-C with weights ranging from 0.0 to 1.0, with increments of 0.05. As the weighting of the three components sums up to 1, we evaluated almost $5,000$ different combinations. Furthermore, we use a decremental approach to explore T, i.e. we iteratively reduce the similarity threshold for clusters. Our decremental approach to identify a feasible clustering similarity is similar to the incremental approach proposed by De Lucia *et al.* for trace recovery using information retrieval [19].

The outcome from the systematic parameter tuning is reported in triangle graphs (cf. Figures 5–7), in which each dot represents a parameter setting. A dot in the center indicates a setting with weights equally distributed between the different components, and a dot in the bottom-left represents a setting with emphasis put on the similarity of the error message, etc.

4.2.4. Evaluation Based on Qualitative Feedback. As stated by Runeson et al. [20] much of the knowledge that is of interest for a case study researcher is possessed by the people working in the case. Thus, as a complement to the evaluation based on ARI for the reference clusterings, we conducted a focus group interview to receive qualitative feedback of our work. A focus group is basically a session where data is collected by interviewing several people at the same time [20].

Three people from the R&D department at Qlik participated in the focus group. Two of them work with configuration management and process automation, and their daily work involves analysis of results from auto testing. The third participant works with development of the autotest framework and is also working with analysis of autotest results on a regular basis. Details of the focus group meeting can be found in Appendix B.

4.3. Threats to Validity

In this work, the primary aim is to support the case company, thus *construct* and *internal validity* are more important than *external validity* and *reliability*. However, for the generalization of the results, the latter two are relevant as well.

The *construct* under study is the analysts' navigation of output from autotests. We have collected both the objective measures of ARI and the subjective opinions from the focus group meeting, together forming triangulated input data, that improves the construct validity. The *internal* validity is primarily concerned with causal relations, and the only one in this case is whether the presence of the tool causes the observed effects. The "Hawtorne effect" can of course never be excluded, but considered having a minimal impact in this case, as the tool is the major treatment.

The *external validity* of the results depend on the similarity of the development context to others. The case with several parallel branches is not unique to Qlik, and neither is the information overload created from test automation. The *reliability* of the study is also relatively good, as the treatment is a tool, which parameters are openly explored and evaluated; thus the same results would be achieved by another set of researchers.

5. Results and Discussion

In this section, results for the two different evaluation techniques are presented. We present the highest achieved ARI for each reference clustering and the corresponding parameter setting. The section ends with the results from the the focus group interview.

5.1. Clustering Accuracy of NIOCAT

Using the best parameter settings, NIOCAT achieved an ARI of 0.59 for RefClust A ("compare with main"). This accuracy of the clustering was obtained using 22 different parameters settings as shown in Figure 5, corresponding to T ranging from 0.85 to 0.55. As seen in the figure, the highest ARI was achieved by settings weighting TC Name and Message higher than HTML. Also, the best settings for RefClust A shifts from up-weighting Message to TC Name as T decreases.

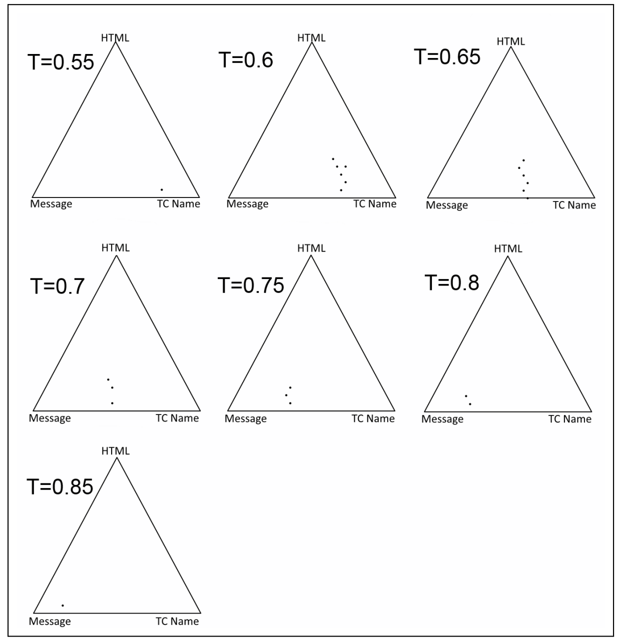


Figure 5. Optimal NIOCAT settings for RefClust A wrt. ARI. Textual information is the most important.

For RefClust B ("overview all development"), NIOCAT achieves an ARI of 1, corresponding to clustering identical to the manually created reference clustering. Figure 6 depicts the parameter settings (almost 400) that yield the optimal clustering, with T ranging from 0.95 to 0.6. At high levels of T, i.e. a strict clustering threshold, up-weighting HTML and Message is favorable. As T decreases however, TC Name continuously gains importance.

NIOCAT achieves an ARI of 0.96 for RefClust C ("identify intermittent failures") using four different parameter settings. Figure 7 shows that a balanced weighting of the components in the similarity calculation obtains the best results.

The results show that the optimal NIOCAT settings vary across the three use cases. However, we observe that when T is high, up-weighting HTML and Message is favorable. This appears reasonable, as HTML and Message consist of more machine generated content than TC Name. Thus, when relying on TC Name for clustering, T should be relaxed to capture variations in the natural language. Nevertheless, it is evident that NIOCAT must provide an intuitive way of changing the setting, preferably with instant graphical feedback on the effect of the clusters.

Table 2 summarizes the accuracy of NIOCAT on RefClust A-C. The table also shows the accuracy of the baseline approaches to clustering: BAMBOO and PASCAL. As described in Section 3.2, both BAMBOO

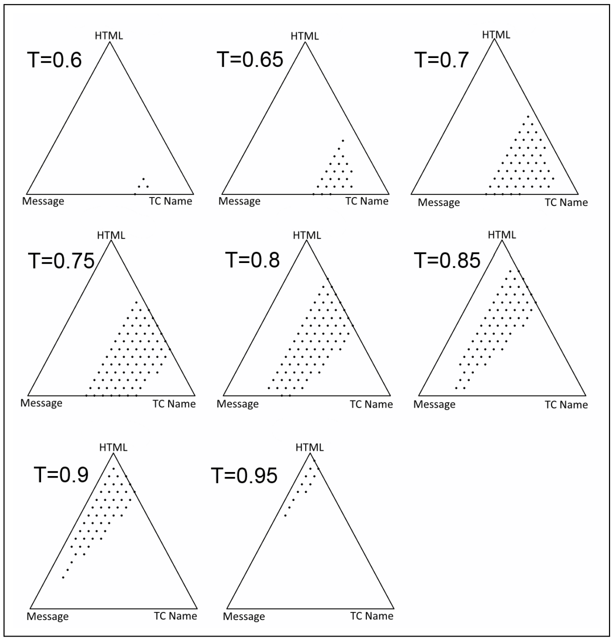


Figure 6. Optimal NIOCAT settings for RefClust B wrt. ARI. Best weighting depends on T.

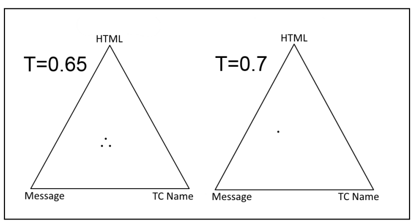


Figure 7. Optimal NIOCAT settings for RefClust C wrt. ARI. A balanced setting is the best.

and PASCAL rely on exact string matching. It is evident that the naïve grouping of TC failures offered by BAMBOO is much less accurate than NIOCAT, as its ARI for all reference clusterings is close to zero. PASCAL outperforms BAMBOO, and for both RefClust B and C at least half of the pairwise relations between TC failures (i.e. in the same cluster, or in different clusters) are correct. However, NIOCAT creates more accurate clusters than PASCAL, and the achieved ARI is higher for all three reference clusterings.

5.2. Feedback from the Focus Group

The answers to all the questions regarding the usefulness of NIOCAT were positive. All participants expressed that the output from NIOCAT provides an improved overview of the current development status, as compared to the current approach.

Regarding what conclusions could be drawn by

	BAMBOO	PASCAL	NIOCAT
RefClust A	0	0.15	0.59
RefClust B	0	0.65	1
RefClust C	0.2	0.5	0.96

Table 2. ARI for RefClust A-C using the three clustering approaches. PASCAL and BAMBOO both rely on exact string matching, using Message and TC Name respectively.

exploring the output in Qlik Sense, the participants confirmed that they were able to cross-reference failures and problems across branches, a valuable feature in decision making related to the test analysis.

A specific characteristic that the participants observed was the wide spread of problems through the SUT, meaning that, given a specific problem, an analyst can quickly find how many branches that are affected. Global frequency for either a specific TC or for a particular problem was mentioned as a further benefit of NIOCAT, i.e. how often a problem is occurring or how often a specific TC fails across all branches. A participant highlighted that it was valuable to see how many TC failures in total that a problem has caused.

One of the participants is responsible for deciding whether a development team is allowed to deliver its code changes to the main branch or not. Using NIOCAT, s/he could quickly determine which problems were isolated to one specific development branch. If the problem only occurs on one branch, that team is obviously responsible for the failure and thus may not deliver its changes to the main branch.

The participants discovered several new NIOCAT use cases during the focus group. The overview provided by NIOCAT enabled the participants to see what problems were the most common across all branches. The participants quickly figured out that a measurement of priority thus could be established, which was not previously possible. This is a use case we had not previously thought of.

Another comment from the group was that the teams, using the information provided by NIOCAT, can quickly determine if a TC failure is occurring on other branches. This could help them determine if they should invest more resources in investigating the TC failure or if it originates from another team. The third use case that was new to us, was suggested as a long term perspective of the tool. A participant pointed out the possibility to identify problem areas guided by NIOCAT. The test developers could then extend their test suites around the areas where many problems occur.

Regarding the potential usage of NIOCAT, two of the three participants explicitly stated that they would use NIOCAT in their daily work if it was available to them. The third participant estimated that his potential usage would be on a daily to weekly basis. To further benefit from the output of NIOCAT the focus group would like to see direct links to even more information about the TC failures, e.g. the original log files and screenshots generated by Horsie.

During the focus group meeting, the participants requested a full analysis across all development branches with data from a week back from the time of the meeting. During a short break we performed a NIOCAT analysis of the requested data and presented the output to the group. The group members were fascinated by what could be accomplished within a few minutes and the results caused an intense discussion. Based on the output, they were eager to take action and discuss problems with development teams. One of the participants stated "an overview like this does not currently exist". Another participant expressed immediate need and eagerness to start using the tool. Other quotes from the group members were "the following few weeks until the tool is put into production will be painful since we know how much the tool could help us" and "imagine all the noise and administration you would get rid of using this tool".

6. Conclusion

The overall aim of this study is to help test analysts at Qlik to overcome the information overload caused by auto testing in multi-branch development. To help the results analyst navigate the enormous information volumes produced by autotests, we developed NIOCAT, a tool that analyses test results across different development branches.

We conclude from the analysis of three reference clusterings and a focus group interview that (RQ1), NIOCAT provides an overview that currently does not exist, and that the participants are eager to start using the tool in their daily work. Further, the clusters created by NIOCAT allows a user to quickly discover information such as on what branches a problem is occurring and how many test runs failed because of a certain problem.

Exploring combinations of parameter settings (RQ2) we conclude that regardless of size and character of the input data, NIOCAT outperforms the two baseline approaches by a large margin in regards to partitioning TC failures into accurate clusters of problems. Thus, considering a combination of execution data (i.e. HTML) and textual information improved the accuracy of the clustering compared to clustering based on textual information alone.

Although there is room for further improvements and enhancements, e.g. preprocessing the textual data representing a TC failure, the feedback was exclusively positive and the life of NIOCAT at Qlik will continue with deployment and real world evaluation.

Epilogue

Six months after development, NIOCAT is now deployed in the Qlik production environment, integrated with the rest of the testing toolchain.

Appendix A. Adjusted Rand Index

The rand index, RI, is calculated using the equation

$$RI = \frac{tp + tn}{tp + tn + fp + fn} \tag{4}$$

where $tp, fp, tn,$ and fn are the number of pairs, classified as true/false positives and true/false negatives, respectively. Thus, the rand index is the fraction of correctly classified pairs of data points among all pairs of data points [21].

The rand index is intuitive but has several known drawbacks, e.g. it is highly dependent on the number of clusters. The Adjusted Rand Index (ARI) was proposed to overcome these issues [16]. ARI can be calculated based on the variables from equation 4 for RI [17]. ARI can thus be computed with the following equation

$$ARI = \frac{ab - c}{a^2 - c}, \tag{5}$$

where a, b and c are defined as:

$$a = tp + fn + fp + tn, \tag{6}$$

$$b = tp + tn, \tag{7}$$

$$c = (tp + fn)(tp + fp) + (fp + tn)(fn + tn). \tag{8}$$

Appendix B. Focus Group Procedures

We conducted the focus group in a number of phases, as suggested by Runeson et al. [20].
i) We explained the concepts of a focus group to the participants, followed by a brief description of the purpose with our work.
ii) We explained how NIOCAT works and demonstrated an example. After the demonstration the participants got to navigate and try NIOCAT themselves.

iii) Next, the interview was conducted, based on five questions:

1) Do you have a clearer overview of the test results now than you had before?

2) Looking at the result you can see and navigate through in QlikView, can you draw any conclusions?

3) Would NIOCAT be of use for you in your daily work? If yes, how? If no, what is needed for you to use it in your daily work?

4) Is there anything else that you would like NIOCAT to present, or anything you would like to change?

5) Do you have any other comments?

iv) We summarized our major findings to confirm that the participants' opinions and ideas from the focus group had been properly understood.

Acknowledgments

The authors would like to thank Lars Andersson at Qlik for initiating the study. Part of this work was funded by the Industrial Excellence Center EASE[4] – Embedded Applications Software Engineering.

References

[1] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[2] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Proc. of the 11th International Conference on Product-Focused Software Process Improvement*, 2010, pp. 3–16.

[3] I. Burnstein, *Practical Software Testing*. Springer, 2003.

[4] M. J. Eppler and J. Mengis, "The concept of information overload: A review of literature from organization science, accounting, marketing, mis, and related disciplines." *Inf. Soc.*, vol. 20, no. 5, pp. 325–344, 2004.

[5] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[6] X. Wang, L. Z. 0023, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information." in *Proc. of the 30th International Conference on Software Engineering*, 2008, pp. 461–470.

[7] L. Pronzato and W. Müller, "Design of computer experiments: Space filling and beyond," *Statistics and Computing*, vol. 22, no. 3, pp. 681–701, 2012.

4. ease.cs.lth.se

[8] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing." in *International Conference on Software Engineering*, 2007, pp. 499–510.

[9] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer, 2014.

[10] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: A systematic map of information retrieval approaches to software traceability," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1565–1616, 2014.

[11] M. Borillo, A. Borillo, N. Castell, D. Latour, Y. Toussaint, and M. Felisa Verdejo, "Applying linguistic engineering to spatial software engineering: The traceability problem," in *Proc. of the 10th European Conference on Artificial Intelligence*, 1992, pp. 593–595.

[12] G. Antoniol, C. Casazza, and A. Cimitile, "Traceability recovery by modeling programmer behavior," in *Proc. of the 7th Working Conference on Reverse Engineering*, 2000, pp. 240–247.

[13] M. Borg, P. Runeson, J. Johansson, and M. Mäntylä, "A replicated study on duplicate detection: Using Apache Lucene to search among Android defects," in *Proc. of the 8th International Symposium on Empirical Software Engineering and Measurements*, 2014.

[14] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report." in *Proc. of the 15th European Conference on Software Maintenance and Reengineering*, 2013, pp. 69–78.

[15] B. Evelson and N. Norman, "Topic overview: Business intelligence," *Forrester Research*, 2008.

[16] L. Hubert and P. Arabie, "Comparing partitions," *Journal of Classification*, vol. 2, no. 1, pp. 193–218, 1985.

[17] J. M. Santos and M. Embrechts, "On the use of the adjusted rand index as a metric for evaluating supervised classification." in *Proc. of the 19th International Conference on Artificial Neural Networks*, vol. 5769, 2009, pp. 175–184.

[18] A. Said, D. Tikk, and P. Cremonesi, "Benchmarking," in *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer, 2014, pp. 275–300.

[19] A. De Lucia, R. Oliveto, and P. Sgueglia, "Incremental approach and user feedbacks: A silver bullet for traceability recovery," in *Proc. of the 22nd International Conference on Software Maintenance*, 2006, pp. 299–309.

[20] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering*. Wiley Blackwell, 2012.

[21] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.