Verify Embedded Systems Faster and more Efficiently with Artificial Intelligence

Alexander Schwarz[1], Björn Morgenthaler[2], Victor Vaquero Martinez[3], Miguel Garrido García[4], Manuel Duque-Antón[5]

[1] comlet Verteilte Systeme GmbH
alexander.schwarz@comlet.de
[2] comlet Verteilte Systeme GmbH
bjoern.morgenthaler@comlet.de
[3] comlet Verteilte Systeme GmbH
victorvaquero.etereot@gmail.com
[4] comlet Verteilte Systeme GmbH
miguel.garrido@comlet.de
[5] comlet Verteilte Systeme GmbH
manuel.duque-anton@comlet.de

Abstract. Embedded systems are the basis for many electronic devices. As a combination of hardware and software designed for specific purposes, Embedded Systems ensure the functionality of Connected Cars, Autonomous Driving, Smart Farming, Industrial Internet of Things and Smart Homes. The enormous competitive pressure forces manufacturers to significantly shorten their time to market and thus reduces the corresponding production cycles. This challenge is directed to the same extent to quality assurance. Due to the constantly growing number of (regression) tests, it is no longer practicable to perform all verifications in all development phases up to the finished product: each quality feature is planned and configured individually. But this approach is usually carried out manually with a lot of effort and is rather rarely adapted over time. On the other hand, software changes very quickly: new features are added, new dependencies arise or are resolved. Communication between individual components change. The probability that errors are found by tests (too) late is substantially increased with each change.

This paper presents an approach that successfully mitigates this challenge with the help of suitable Artificial Intelligence methods. To reduce lead time, a mechanism is developed that reduces the number of required tests. Based on the data from previous verifications, a (significantly) smaller subset of tests, which is sufficient to verify the correctness of the change, is selected. The remaining probability that tests necessary for negative verification of the software are not considered, is thereby accepted. Initial results, using data from several open-source projects as well as the use of a prototype machine learning pipeline, show promising results with respect to their predictive capabilities.

Keywords: Embedded Systems; Automated Testing; Artificial Intelligence; Reduced Complexity of Testing; Machine Learning; Continuous Testing; Industrial Internet of Things (IIoT); Agile Software Development

1 Introduction

As a technology partner and solution provider, comlet connects embedded systems of both global manufacturers (OEMs) and medium-sized companies into an overall system and offers intuitive usability by Artificial Intelligence (AI) while generating increased value. For example, predictive maintenance can be offered in the Industrial Internet of Things and new security risks can be detected in digital networking with the help of intelligent anomaly detection. To assure quality, comlet can also reduce the complexity of necessary tests with the help of AI while maintaining reliability.

The rise of monolithic databases, agile methodologies doing fast integration and multimillion source code line projects have created a practical problem in terms of computing power. What previous methodologies could do with branch manual testing and nightly runs is impossible or too time consuming to do for Continuous Integration (CI) teams.

2 Problem

Approaches have been established that take the introduced challenge into account: one of which being Continuous Testing (CT) as the integration of automated tests into a CI pipeline (see *Figure 1*), including the requirement of detecting errors as early as possible ("fail fast").

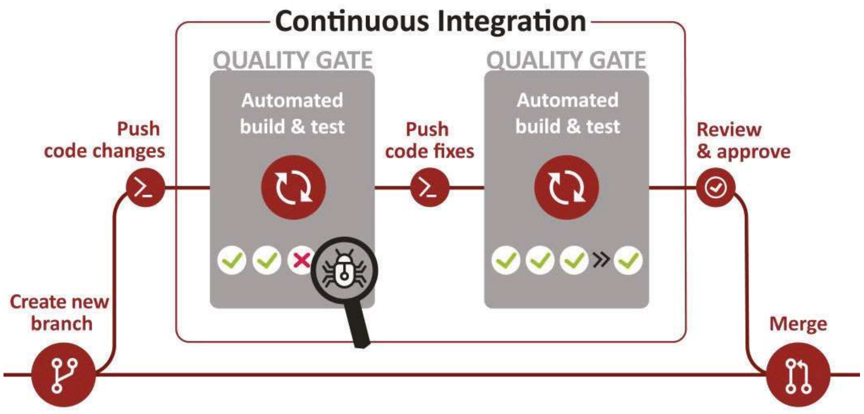


Fig. 1. Test automation as quality gate for a CI pipeline

In a CI pipeline, software is continuously developed in separate branches and eventually merged with the main development branch (trunk) once it has been completed. To assess the quality level before the individual branches are merged, various checks are performed at the so-called quality gates (QG).

Each QG is adequately planned at each integration step and configured with respect to the trade-off between duration, the risk of not detecting errors and feedback (see *Figure 2*).

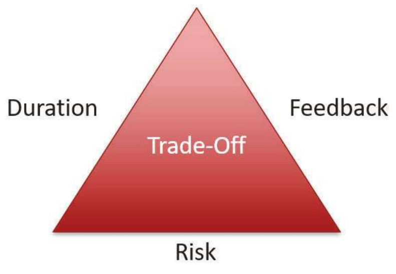


Fig. 2. Trade-Off while planning QGs

This process is usually carried out manually and must be adapted according to the changing requirements during development.

The developer also receives immediate feedback by the QG whether the changes contain errors and is obliged to eliminate these accordingly before integration.

As every step of development gets moved to the trunk, there is a higher need for quality control before and after the integration. The problem resides here, as projects with hundreds or thousands of commits and thousands or more of test suites cannot bear the computing power needed for running every test on every new code change – even with CT.

3 State of the art

To manage the previously mentioned problems, there have been multiple solutions: Google [1], for example, uses a simplification by accumulation. First, they use a bigger aggregation of tests named test targets. Secondly, they batch multiple code changes until they reach a specific threshold, e.g., 100 commits. At that moment the new code version, with the complete recollection of code changes, is run as a single unit on every test target (needed) that has not been run since the latest quality control.

This solution is based on a couple of heuristics and approximations but at its core it runs on something named change-based testing. The main property of this scheme is to only run the tests that somehow (the means change between methods) depend on the code changes, and as such are likely to fail, to reduce the resources needed. For this, useful features like historical failure data or code dependencies could be used. It can be done by any manual or automatic heuristic, but any non-automatic means bear a problem of scalability.

As of now there have been quite a few ad-hoc attempts at resolving the change-based testing task, and specifically, selecting an optimal subset of tests. Other companies commonly do it with specialized automatic solutions for some business or project - sometimes in conjunction with manual work [2] [3] [4]. There have also been some attempts at solving this problem [5] [6] by using simple heuristics and rules like recent failures, similarity between test and code; and specially code coverage. Some try to develop a set of diversified tests, with coverage measures or input analysis to cover the whole code change through search methods like greedy or local beam search [7]. There's also some more general development in risk prediction through specific features like historic failure data, similarity measures, code changes [2], etc.

To the author's knowledge there are only a couple of public papers, from Google [1] and Facebook [8], trying a similar approach of using machine learning to resolve the problem of selecting an optimal test subset but, as mentioned before, only for their own companies in an ad-hoc fashion. Another paper [9] used a reinforcement learning method only with historical failure input data.

The tool "Sherlock" by OMICRON [10] [11] tries to blend automatic and manual test selection by means of code-test dependency by dynamic analysis but without taking care of the remaining information.

To the author's knowledge up until today there is no homogenized approach that analyzes the general problem for every possible available input.

4 Solution

To solve the above stated problem more universally, the development of a pipeline capable of predicting a minimal test subset through training on historical test failure data is approached (see *Figure 3*).

Main objective then is to create a data model to input this learning algorithm and identify the minimum features required, the cost-effectiveness of the features, the most suitable algorithms and finally the actual predictive capability of the learning model.

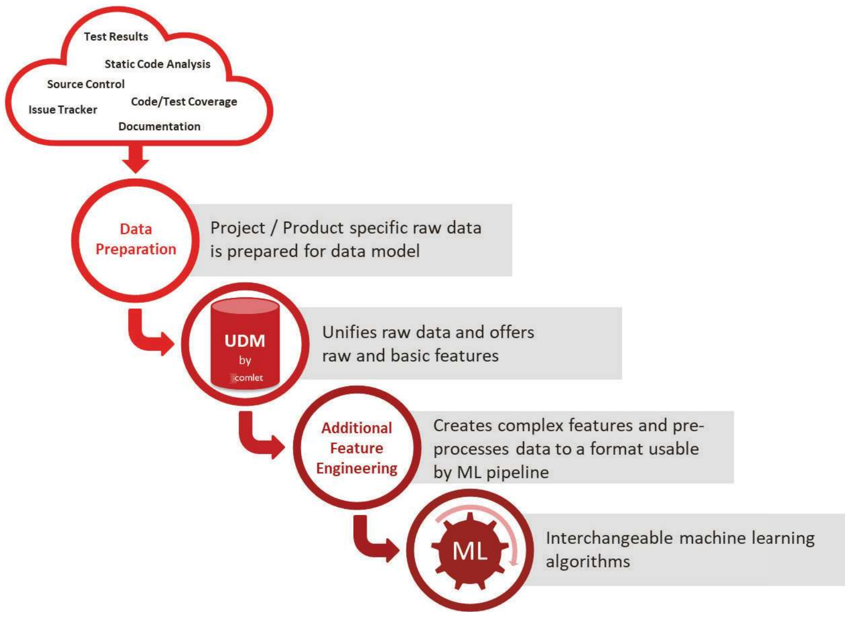

Test Results
Static Code Analysis
Source Control
Code/Test Coverage
Issue Tracker
Documentation
Data Preparation
Project / Product specific raw data is prepared for data model
UDM by comlet
Unifies raw data and offers raw and basic features
Additional Feature Engineering
Creates complex features and pre-processes data to a format usable by ML pipeline
ML
Interchangeable machine learning algorithms

Fig. 3. Overview of proposed pipeline

The pipeline transfers all available data into a common, unified and abstract data model, the Unified Data Model (UDM). This data can come from various, very different sources, such as issue or bug trackers, source control management, documentation, (static) code analysis and automation servers.

Prior to transferring the data from all possible sources into the UDM, preprocessing and data preparation is required. The raw data presents itself in very diverse formats, for example all sort of databases such as SQL or No-SQL or file formats like JSON or CSV. Therefore, those (raw) data points are converted into a generalized data structure which is easy to work with. While this structure is created, it is also assured that the different values like dates or time are converted to common formats as well. Eventually, all data points can be classified into the following five categories:

- *Info Point*: Abstraction of any documentation about architecture, tasks, bugs, issues, pull requests

- *Code Block*: Abstraction of any possible aggregation of code as it is in the current moment in time. There are multiple possible types, but some would be modules, files, classes, functions, etc.
- *Code Change*: A basic data point that records every historic modification to the code base and serves as a version reference.
- *Test*: One of the basic data points comprised of the most updated information about the tests that are currently in use.
- *Execution*: Record of every time a *Test* has been run, keeping historical data about the successes or failures of every *Test* and *Code Change* to which every *Execution* is related to.

These five categories define the basis for the Unified Data Model which serves as an interface between source specific raw data and the unified feature engineering and machine learning algorithms, resp. models. Thus, it completely separates feature engineering and training of machine learning models from the specific (raw) data. The UDM itself provides raw and basic features out of the box that do not require further processing. Of course, a more thorough data analysis including data preprocessing and complex feature engineering based on the unified data is also performed to optimize the machine learning results.

Based on the data provided by the UDM, an attempt is now being made to reduce the time required to test software together with the time a developer waits for feedback whether the changes are causing test failures. This results in two different goals:

- *Subset selection*: Given a set of *Tests* and *Code Changes*, output the subset that finds all the failures. The preferred method to achieve this is to analyze every test individually and determine a value between 0 and 1 that reflects the probability of failing. With these probability values for all tests, it is possible to create an optimal subset.
- *Test prioritization*: Given a set of *Tests* and *Code Changes*, output the optimal order of the set. This optimality can be measured in several different ways, such as mean time or maximum wait time. The objective is to give feedback to the developer as soon as possible. It is also possible to use the same setup as in the subset selection algorithm simply by choosing those tests with a greater risk of failure first.

The specific approach as an initial step towards the general solution is to identify a minimal subset of tests that are very likely to fail, i.e. to fulfill the subset selection goal. No further prioritization of these tests will be done for the time being. For this purpose, data is collected from two large open-source projects, Pytorch [12] and Chromium [13]. This data is then merged into the UDM.

Since the UDM can manage different projects, it also ensures that non-existing features and missing values in the raw data are filled with meaningful values. This is necessary so that all further steps such as feature engineering or machine learning can be performed independently of the source project.

The UDM and the general approach of selecting a set of tests that are likely to fail are validated with a simple decision tree (DCT) algorithm. The DCT was chosen because it is easy to understand and a look at the internal structure is possible at any time. This gives more insight into the underlying data and features.

5 Results

The first step is defining performance metrics to compare the quality of different experiments objectively. After that, multiple experiments are conducted with different sets of features. From these experiments the first ones use only raw features which are directly available from the data and the next experiments use more complex engineered features.

5.1 Performance / Quality metrics

Predicting the outcome of a test is mainly a classification problem. Therefore, the data is labeled with two classes: the successful tests are labeled with "success" and the failing tests with "failed". Because the goal is to predict if a test fails, the failing tests are considered as positives. The following performance metrics are used to evaluate the quality of the trained models and the overall approach.

Precision is a measure for false positives. This means that a successful test is predicted as failed test. A low precision leads to a subset of tests with many tests that are successful.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

On the other hand, there is the *Recall* which is a measure for the false negatives. False negatives are failing tests that were predicted as successful, so a low recall will lead to the problem that many failing tests are not predicted as those.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

The overall A*ccuracy* is simply a measure for how many tests were correctly predicted.

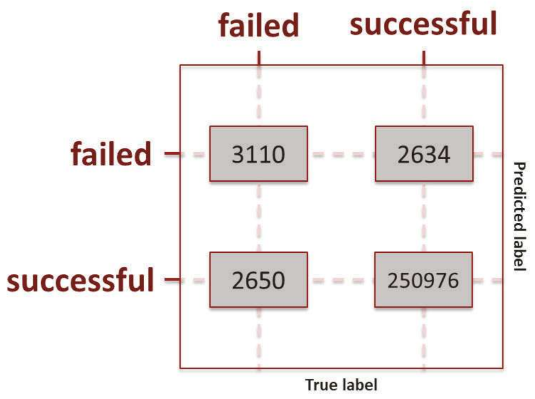
$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

The focus lies on improving the recall as much as possible without decreasing the overall accuracy and precision too much. This is due to the higher-level goal in predicting failing tests as early as possible. With a high recall fewer failing tests will be wrongly predicted.

5.2 Raw features

For the first prototype, it was decided not to work with individual test cases, but on the next higher hierarchical level: test suites. Test suites combine several tests that belong together thematically. This makes it much easier to deal with the huge amount of data in the first step. Another advantage is that it is easier to create a measure of complexity. In the case of test suites, complexity is the number of associated test cases. If individual test cases were considered, such a complexity measure could be, for example, the number of lines of code or even the number of function calls. This information is not directly available in the open-source projects considered and is very complicated to generate from the existing data. Another limitation chosen for the first prototype is to set the focus on whole commits and not on single files that changed. A commit can consist of changes in multiple files.

On this basis, initial experiments were conducted with a small subset of the collected data to better assess the underlying data. *Figure 4* shows the results of a decision tree when all directly usable features from the raw data are included. These are for example added, removed, changed lines of code, number of changed files, number of comments in an issue, number of reviews, number of tests inside a test suite, etc.



Metric	Value
Precision	0,5414
Recall	0,5399
Accuracy	0,9796

Fig. 4. Confusion matrix and metrics using only raw features

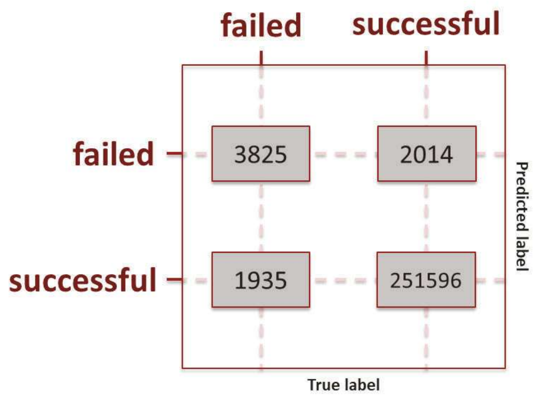
These results are from a 10-fold cross validation. The overall accuracy of the decision tree is close to 100%, as expected, because there is a strong imbalance in the data: there is much more data from successful tests than from failing ones. But despite this imbalance, already more than 50% of the failed tests are correctly identified as such, which is reflected by a recall of 53.99%. In principle, this shows that it is possible to predict the probability that a test will fail.

5.3 Additional and more complex features

Since it is not sufficient to predict only half of the failed tests, the recall must be increased even further. This is achieved by encoding the unused information in the raw data so that it can be used to train machine learning models and by engineering entirely new features from the existing data.

An example of such raw data that can still be usefully encoded is the build target, i.e. the underlying architecture or the operating system. In the raw data this is only present as a simple string containing all this information. A feature that was created completely new is the number of historical failures of a test; more precisely: how often a test failed in the last n runs.

With these new features, a recall of 66.4% is already achieved, as can be seen in *Figure 5*. Compared to the previous results, this value is already an improvement of 22.98%.



Metric	Value
Precision	0,6550
Recall	0,6640
Accuracy	0,9847

Fig. 5. Confusion matrix and metrics with additional features

It is expected that with further features the recall can be increased significantly without degrading the accuracy or precision. In the raw data, there is still a lot of information available, from which suitable features are yet to be generated through appropriate encoding and feature engineering.

6 Conclusion

The results from the experiments show that the introduced Unified Data Model works. The data from different software projects can be unified with it, which was examined at the example of two open-source projects. This makes it possible to carry out further data processing and feature engineering independently of the specific project. Also, the used machine learning algorithm can be exchanged now without adjustments. This makes it much easier to compare the performance of different algorithms and to select the best suitable one. The results also show that, despite the strong imbalance in the data in favor of successful tests, a recall of over 50% can be achieved even with a relatively small set of features. That is, over half of the tests that fail are correctly predicted. This recall could even be improved significantly with the first additionally engineered features.

The next steps are now to validate the Unified Data Model using a third open-source project as well as the first results of the machine learning model with significantly larger data sets. Also, the recall must be improved by further feature engineering and appropriate encoding of the still unused information in the raw data. Furthermore, it should also be investigated what results can be achieved if the granularity is changed away from test suites back to individual test cases. Likewise, it could improve the recall again clearly, if the focus lies no longer on the individual commits, but directly on the changed files. This would also give the opportunity to use some sort of code/test dependency if available as feature. It should also be possible to learn this dependency from historical test executions if it shouldn't be available.

References

1. Atif Memon, Zebao Gao, et al. "Taming Google-Scale Continuous Testing", IEEE/ACM 39th International Conference on Software Engineering, 2017

2. Quinten David Soetens et al., "Change-Based Test Selection in the Presence of Developer Tests", 17th European Conference on Software Maintenance and Reengineering, 2013
3. Elmar Juergens et al., "Regression Test Selection of Manual System Tests in Practice", 15th European Conference on Software Maintenance and Reengineering, 2011
4. Everton Note Narciso et al., "Test Case Selection: A Systematic Literature Review", International Journal of Software Engineering and Knowledge Engineering, Vol. 24 No. 04 pp. 653-676, 2014
5. Armin Najafi et al., "Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report", IEEE/ACM 41st International Conference on Software Engineering, 2019
6. S. Yoo, M. Harman, "Regression Testing Minimisation, Selection and Prioritisation: A Survey", Software Testing, Verification & Reliability, ACM Digital Library, 2012
7. Bo Jiang et al., "Input-based adaptive randomized test case prioritization: A local beam search approach", Journal of Systems and Software, 105: 91-106, 2015
8. Mateusz Machalica et al., "Predictive Test Selection", IEEE/ACM 41st International Conference on Software Engineering, 2019
9. Helge Spieker et al., "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration", ISSTA 2017: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 12-22, 2017
10. Rudolf Ramler et al., "Tool Support for Change-Based Regression Testing: An Industry Experience Report", International Conference on Software Quality, Springer Press, pp. 133-152, 2016
11. Christian Salomon et al., "Sherlock: A tool prototype for change-based regression testing", ASQT 2013 - Selected Topics to the User Conference on Software Quality, Test and Innovation 2013, OCG, Vol. 303 pp. 33-36 2014
12. Adam Paszke et al., "Pytorch: An Imperative Style, High-Performance Deep Learning Library", Advances in Neural Information Processing Systems 32, pp. 8024-8035, 2019
13. The Chromium Authors, "Chromium" [Online] https://www.chromium.org/Home [Accessed 24.08.2021]