Regression testing techniques in a continuous integration environment - a comparison

Emanuel Eriksson
mat14ee1@student.lu.se

Keiwan Mosaddegh
ke2476mo-s@student.lu.se

Max Strandberg
ma2536st-s@student.lu.se

Erik Stålberg
er4047st-s@student.lu.se

***Abstract*—Regression testing is an important part of any software project, but can be both very costly and time consuming. This is especially true in a continuous integration (CI) development environment. This paper analyses and compares four new techniques for regression testing. In the CoDynaQ method the dispatch queue is continuously re-prioritized based on the remaining test cases' historic co-failure distributions with the ones executed. The RETECS method is a new method which uses reinforcement learning and neural networks to prioritize and select test cases. ROCKET is a test case prioritization approach where the test cases are prioritized by their historical failure data, and execution time. Finally, the Bloom filter method improves regression testing by using the Bloom filter data structure to filter out test that only fail once, and never again. The methods all show promise and each of them outperform their respective benchmarks. CoDynaQ also has the possibility to be combined with any of the others, or yet another method. We, however, find it unlikely that any of them will be widely applied anytime soon, as these types of academic results have generally proven themselves slow to propagate into industrial practice.**

I. Introduction

The purpose of this report is to perform research and deep dive into the field of regression testing in a continuous development environment. The report will examine a handful of different methods, such as bloom filters and ROCKET, and compare the results these methods produce in the context of continuous integration.

A. Continuous integration development environment

The context of this report is that of a continuous integration (CI) development environment, a method of software development that has generated a lot of attention in recent years. In general, the CI approach differs from traditional software development in that changes in software are integrated into the main code base as quickly as possible, to prevent large divergences from impacting merge stability. There are many advantages to using CI, but they are not within the scope of this paper. However, working within a CI environment is not without risk, as developers lose the ability to safely experiment in development branches separate from the main production branch. Continuous Integration, especially when combined with Continuous Deployment, may be a risky endeavor - any issues and bugs need to be located and addressed as soon as possible.

Continuously ensuring the stability and correctness of the main production branch is essentially impossible to do through manual means. Therefore large-scale automatic test suites are a prerequisite for CI to function as intended. As software evolves and grows over time, the need for comprehensive regression testing increases. According to Kerzazi and Khomh, as merge intervals and release cycles grow shorter or disappear entirely, testing activities are the greatest time bottleneck [2].

As a consequence of this, regression testing in continuous integration must be efficient and cost-effective. Simply executing all regression tests, or only tests directly impacted by a change in the code-base, have proven to be weak and unsustainable approaches. There exist a number of new

How does regression testing fit into a continuous workflow? why is it necessary, or what purpose does it serve?

B. Regression test selection

Since regression testing can be very costly and time consuming it is important to choose wisely which tests to run. This is especially true for CI environments where development cycles are very short. In other contexts regression tests may be initialized by the end of the work day and the results received in the morning. This would, however, be to disruptive to work efficiently in a CI environment. Therefore, each regression test suite has to be chosen carefully, so as to minimize the time consumed. This is called the *regression test selection* (RTS) process. This is partly done by removing tests with a low failure probability, but also by removing tests that overlap already selected test cases.

C. Test case prioritization

After the RTS process it is time to prioritize the tests. The purpose of test case prioritization (TCP) is to reveal failures as early as possible. This lets the tester proceed to fix the revealed bugs, or to stop the execution of the regression test suite and return to the drawing board.

D. Research Question

RQ1: *How do the regression testing techniques discussed in this report compare, in the context of a continuous integration development environment?*

II. Description

1) ROCKET: The authors Marijan, Gotlieb, and Sen present the test prioritization approach *ROCKET* in their paper [5]. The aim of ROCKET is to effectively reduce the time required to test, while increasing and keeping a high fault detection rate. In order to achieve these characteristics, the authors choose to prioritize test cases based on their previous amount of failed test executions. The historical failure data is then translated into a so called *failure weight*. If two test cases have the same failure weight, then their respective execution time is taken into account, where the test case with the shortest execution time is prioritized.

ROCKET requires 4 different input data: the set of the test cases to prioritize $S = \{S_1, S_2, ..., S_n\}$; their corresponding execution time; each test case's historical failure data; and the total execution time limit for the test suite T_{max}. Initially all test cases receive a priority of 0. A failure matrix *MF* is constructed from the data of the test cases' historical failures (equation 1).

$$MF[i,j] = \begin{cases} 1, & \text{if } S_j \text{ passed in } (current - i) \text{ execution} \\ -1, & \text{if } S_j \text{ failed in } (current - i) \text{ execution} \end{cases} \tag{1}$$

The impact of a failure that occurred *i* executions since the current execution is assigned an impact weight w_i, that reflects the level of impact of the failure. The value of the weight is hence based on how many executions ago a test case failed, and the evaluation looks as in equation 2.

$$w_i = \begin{cases} 0.7, & \text{if } i = 1 \\ 0.2, & \text{if } i = 2 \\ 0.1, & \text{if } i \geq 3 \end{cases} \tag{2}$$

At this point, the cumulative priority for a test case P_{S_i} can be calculated. This is done by taking the sum of each value from the failure matrix for the specific test case, multiplied by the corresponding impact value. As a result of this, every test case has a priority value, where a lower value corresponds to a higher priority.

With the historical failure data taken into account, an evaluation of the execution times remains. First; the test cases are put into different classes based on their calculated priority value. All test cases in the same class have the same priority value *t*, which is increased by 1 for the successive class. ROCKET then checks the execution time of each test case T_{e_i} and assigns a new priority value as shown in equation 3:

$$p_{S_i} = \begin{cases} t + 1, & T_{e_i} \geq T_{max} \\ p_{S_i} + \frac{T_{e_i}}{T_{max}}, & otherwise \end{cases} \tag{3}$$

As mentioned; lowest priority value means highest execution priority. If a test case is put in the first class (by having the highest total impact weight), and is assigned the lowest additional priority value (by having the shortest execution time), it (the test case) has the highest priority.

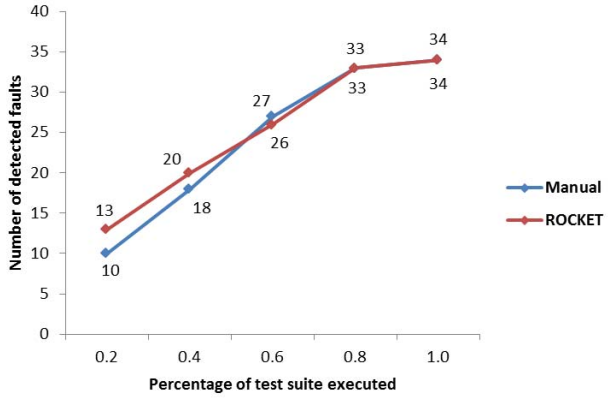

Fig. 1. Number of detected faults

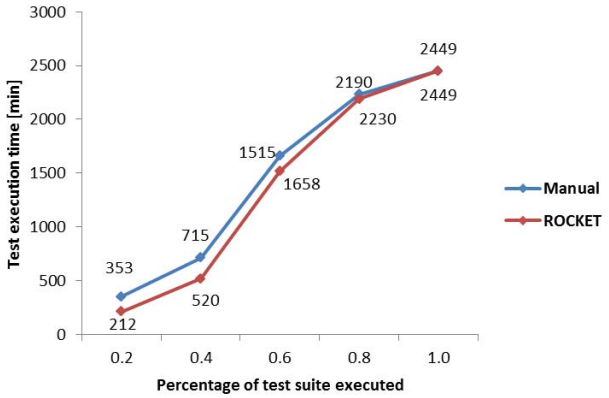

Fig. 2. Test execution time (min)

The context of which ROCKET is derived from, and tested in, is an industrial video conferencing software. The software system consists of 100 test cases, with an average of 30 minutes of execution time per test case. This means that a test session with no test case selection nor prioritization would require a minimum of 2 days. When measuring the performance of ROCKET, the authors compared their approach against manual prioritization by test engineers. The results of the comparison are presented in figure 1 and 2. The former (1) shows the amount of detected faults compared to the manually prioritized test cases. The latter (2) instead shows how the test execution times compare between the different approaches.

For the first tests 20% of the test suite were to be prioritized. The test was then repeated, increasing the size of the test suite every time, in 20% increments.

ROCKET-prioritized test cases initially outperformed manually prioritized test cases. 3 more faults were detected in 40% less time. The progression of figure 2 shows how the difference in test execution time eventually reaches 0. This is expected, as the complete test suite is executed in both cases, and the act of prioritization becomes inconsequential. However, ROCKET-prioritized test cases consistently execute in less time, and overall detect more faults. Additionally, Cost-

cognizant weighted Average Percentage of Faults Detected (APFDC) was used to measure the effectiveness of ROCKET, compared to manual prioritization and random ordering of test cases. APFDC rewards test case orders proportionally to their rate of units of fault severity detected per unit test cost. In the case of ROCKET, the cost is determined by the execution time of the test case. ROCKET performed the best against the other methods, and received a score of 17.09, compared to 15.81 and 13.85, received by manual and random, respectively.

2) RETECS: RETECS is a new method used in continuous development environments to perform regression test selection and regression test prioritization. It performs these tasks through reinforcement learning and neural networks, and performs analysis on the test case failure history, computation time, and last previous execution[7]. The goal of RETECS is to reduce the time it takes for the developers to receive failed test feedback after committing new code.

In the article, Spieker et al. state that compared to other prioritization algorithms, RETECS is able to adapt to situations where test cases are added or removed. It is also capable of adapting to new test prioritization rules. The cost of running the prioritization is negligble as RETECS doesn't do any costly computations during prioritization.

RETECS itself is as stated by Spieker et al. an application of reinforcement learning as an *online-learning* and *model-free* method for the ATCS problem. Model-free meaning that it has no initial knowledge or concept of the environment, and online-learning means that the method is constantly learning, even during run-time. The reinforcement learning works by letting an agent interact with its environment, and select an action based on attributes such as learned policies or random exploration. The agent will then receive feedback in form of a reward, which will tell the agent if the selected action did well or not. From this feedback the agent will develop and adapt learning policies regarding behaviour and action choices

Spieker et al. write in their article that conventionally rewards should be negative or positive to deterr respectively promote behaviour, and based on common metrics used in the TCP and TCS. This does however require knowledge about the whole system, something which is impossible to obtain in a CI environment. Therefore RETECS only supplies its agents with positive or zero feedback.

To evaluate RETECS Spieker et al. used the it and three other methods. The first other method is called *Random*, it acts as baseline and is a random test case prioritization method. The second method is called *Sorting*, and is a test case prioritization method which sorts the test cases where recently failed cases have higher priority. The third and final method is called *Weighting*, which Spieker et al. defines as a naive version of RETECS as it analyses the same data and does so with weighted summation with equal weights. These four methods were used on three data sets from the industry, paint control, IOF/ROL and GSDTSR. For the paint control data set, the paper concludes that within 60 CI cycles, RETECS is on par, or better than other methods. Similar results are seen on the other two data sets, but with a longer adaption phase and smaller performance difference on IOF/ROL, and a comparable performance on GSDTSR.

3) CoDynaQ: In their article [8], Zhu et al. present a novel approach for TCP in CI environments called *CoDynaQ*. More specifically, the article proposes three variants of this method called *CoDynaQSingle*, *CoDynaQDouble* and *CoDynaQFlexi*. Their method is based on two ideas. The first is to make use of the co-failure distributions between test cases. The second is to re-prioritize test cases already in the dispatch queue. Every test case is assigned a priority score s. This score is updated according to equations 4 and 5, where t_1 is the test case just executed and t_2 is the test case whose score is being updated.

$$s_{2,new} = s_{2,old} + (P\{t_2 = fail | t_1 = fail\} - 0.5) \quad (4)$$

$$s_{2,new} = s_{2,old} + (P\{t_2 = fail | t_1 = pass\} - 0.5) \quad (5)$$

If t_1 and t_2 have never been run together before their co-failure probability is unknown, and may be set to anything between 0 and 1. In their article, however, Zhu et al. choose to set it at 0.5, for them to maintain their original score. After each test case execution the scores of the remaining tests cases in the dispatch queue are updated and then re-prioritized based on their respective new scores.

The three method variants vary with respect to their test queues. The simplest variant CoDynaQSingle only has a single queue to which test case requests are added. After a test case has been executed the priority scores of all remaining test cases are updated and the queue is reordered. This, however, leads to a problem called *starvation*. Starvation is when a test case is continuously pushed back in the queue and whose result is thus further and further delayed. If the tester suspects a test case will fail it may assigned it an original high priority, but if the test case has low co-failure rates it may be substantially delayed, even indefinitely, if new tests are continuously requested.

This problem of starvation is addressed by dividing the single queue into a *dispatch queue* and a *waiting queue*. The waiting queue is a FIFO queue (first in, first out) to which new requests are added. The dispatch queue on the other hand is continuously re-prioritized as described earlier. In the CoDynaQDouble variant the dispatch queue is filled to capacity when empty. By contrast, in the textitCoDynaQFlexi the dispatch queue is refilled, not when empty, but when the remaining number of test cases sinks below a certain threshold.

The performance of these methods were tested on a set of internal test data from Google and another from the Chrome project. The Google data set contained 11,457 change requests, resulting in 847,057 test case executions. The Chrome data set contained 235,917 change requests, resulting in 4,487,008 test case executions. As a baseline for comparison, the methods were compared with a simple FIFO-prioritization (i.e. no prioritization). Furthermore the methods were compared to a method Zhu et al. call *GOOGLETCP*, but which in the original article [1] is called *SelectPRETests*. There, on regular

time intervals, the test cases in the waiting queue are given a priority of 1 if $t_f < W_f$ or $t_e > W_e$, where t_f and t_e are, respectively the, times since last fail and last execution, and W_f and W_e are corresponding thresholds. The waiting queue is then prioritized based on these scores and added to the back of the dispatch queue.

These methods were then compared to the baseline FIFO model on the following metrics on both data sets.

- Median relative time gain until first failure detected
- Median relative time gain until all failures detected
- Median proportion of delayed failures, relative to FIFO

The results are shown in table I, where the best performances are shown in bold.

TABLE I
CODYNAQ PERFORMANCE RESULTS.

Chrome	CoDynaQSingle	CoDynaQDouble	CoDynaFlexi	SelectPRETests
First failure	**11.33%**	0.84%	5.01%	3.34%
All failures	**61.84%**	0.05%	0.08%	0.05%
Delayed failures	19.11%	17.34%	**0.51%**	31.78%
Google				
First failure	**31.01%**	0.04%	0.18%	4.36%
All failures	**39.60%**	0.09%	0.20%	4.61%
Delayed failures	31.14%	27.26%	**6.20%**	8.22%

Here we clearly see that CoDynaQSingle outperforms the other methods with respect to the first failure and all failure metrics. CoDynaQFlexi outperforms the others with respect to the delayed failures metric, however the starvation problem is also captured by the all failures metric, so it is not necessarily best with respect to actual starvation.

4) Bloom filter method: In a paper published in APSEC, authors Kwon and Ko present and discuss a new method for regression testing in continuous integration environments [3]. The authors argue that the testing technique, here called the Bloom filter method, is a combination of both logical and technical improvements on standards regression testing procedures. The method incorporates changes to how both RTS and TCP is performed, and originates from the prevalence of *one-hit-wonders* amongst failed test suites. The implementation presented is largely build around Bloom filters, a hash-based data structure with many characteristics that prove advantageous to the method's design. In an experimental comparison with industry baseline techniques, the Bloom filter method improves on the average number of failures detected by a factor of 2.23, and reduces the time taken to detect a failure by up to 42.2 hours. The experiment was performed on a sanitized data set of tests from Google.

Kwon and Ko motivate the use of Bloom filters by arguing that the technology is well suited for precisely the type of fast operations their method relies on. Essentially, Bloom filters are a data structure built around *k* hash functions connected to each element in an *m*-bit array. As each hash function is probabilistically unlikely to clash with another, *k* hash functions represent *k* element positions in the array. The purpose of the array is to mark if an item has been added to the data structure, and allows one to quickly check for the presence of an item within the filter. If well designed, a Bloom filter has a negligible probability of both false negatives, as well as false positives. With a time complexity of $O(1)$ for a content check operation, Bloom filters are incredibly time-efficient. Kwon and Ko also analyze if the added overhead of using them as part of regression testing has a negative impact on testing times, and show that they do not.

The Bloom filter method is designed around the existence of test suites that fail once, but do not fail again. These test suites, also called *one-hit-wonders*, make up 44% of failed test suites in the pre-submit phase of testing, and 33% of failed tests in the post-submit phase. The concept of one-hit-wonders is not unique to testing, as they have been observed in many other areas of computer science. Kwon and Ko note that the idea for the Bloom filter method originates from the over-representation of one-hit-wonders amongst cached objects in a Content Delivery Network . According to Maggs and Sitaraman, approximately three-quarters of objects accessed within a CDN are requested only once, which making caching them extremely inefficient [4]. The Bloom filter method attempts to use this observation to improve both test suite selection and prioritization.

In practice, the Bloom filter method attempts to improve on existing window-based selection and prioritization techniques. Both RTS and TCP rely on a Bloom filter combined with a *failure cache*. When a test suite fails, the method checks if it exists in the Bloom filter. If it does not, it is added to the filter, but not to the failure cache. This effectively marks it as a one-hit-wonder. If the Bloom filter already contains the test suite, it is instead added to the failure cache. Kwon and Ko argue that in the context of regression testing, test suites are analogous to test cases, and that their method could be applied on either scale.

For RTS, a subset of test suites are selected as candidates for execution, based on criteria such as Last Failure Time (LFT), Last Execution Time (LET), or if they are entirely new. From these selected test suites, a subset is created, containing only new test suites and ones from the failure cache. The authors argue that the exclusion of one-hit-wonders does not affect the overall effectiveness of regression testing, if it is done in the pre-submit phase of development.

As for TCP, the Bloom filter method operates on similar principles as for selection. Essentially, test suites that failed more than once are assigned a *high* priority, and one-hit-wonders are given the *lowest*. In practice, test suites from the failure cache are given a priority of 0, as they have failed more than once. Test suites that fulfill the regular window-based criteria are assigned a 1, and suites that fulfill none of the above are assigned a 2. Test suites are then executed in ascending order.

Kwon and Ko document the results of their experiment, which show that the Bloom filter method is much faster and more precise than random TCP and RTS. The same is true when compared with baseline industry methods, improving the average number of failures detected by 2.23 and reducing execution time by between 4.9 to 42.2 hours.

III. Analysis

The experiments conducted in [5] were, as mentioned, done in the context of a video conferencing software, where the test executions required a relatively significant amount of time. There is naturally a need to ensure that the tests that are to be executed are likely to find faults, as each execution requires time and resources that are desired to avoid wasting.

An interesting addition to ROCKET would be how dynamic and real-time selection and prioritization could be used to improve the performance of the approach. By achieving a sufficient real time test coverage estimation of the executing tests, the ROCKET approach could stop executing test cases as soon as it considers a sufficient test coverage to be reached. The approach of dynamic (re)selection and prioritization adopted by CoDynaQ could for example be something to attempt incorporating into ROCKET.

Generally; the logic behind the ROCKET approach is fairly intuitive. There is a problem of time consuming test case executions, and there is a need to be frugal with the available time. Therefore it's important to ensure that the executing test cases have a weighted history of failing, and that their execution times are as short possible.

One could consider ROCKET to be a fairly promising approach, but it's mainly a matter of context. Tests taking more than 30 minutes to execute is not the first thing that comes to mind when thinking about continuous integration. Then again, the fact that this is in the context of continuous integration is what makes an appropriate test case selection and prioritization so important in the first place.

If the ROCKET approach is just as effective in the context of test cases with significantly shorter test execution times is unclear. Perhaps the improvements that ROCKET delivers are not as viable when shifting from talking about hour-long improvements for the VCS context, to a couple of minutes in another context.

In regards of prioritization and selection ROCKET might find a greater value in the prioritization. ROCKET is derived from the context of time consuming, 30 minute (on average) long test case executions, where each executed test case has a significant impact in terms of time and resources. Therefore, by putting the test cases in an order where the first test case is very likely to fail, one receives (relatively) quick feedback if that is the case. On the other hand, if no test case selection is present, a significant amount of redundant test cases are prone to be executed.

With regards to the ROCKET method we have a proposed improvement. Instead of storing all previous results for every test case and multiplying it with a fixed set of weights w_i one could recursively update each priority score as

$$w_i = \begin{cases} 0, & i = 0 \\ \alpha\lambda_{i-1} + (1-\alpha)w_{i-1} & i \geq 1 \end{cases} \tag{6}$$

where λ_{i-1} is 1 if the last execution passed -1 if it failed and α weight for the last execution result. This approach retains the method's original property that recent results are more important than previous. Instead of having to store all previous test results, one only has to store a each test's priority score, which is updated after each test run.

RETECS aims to improve the process of test case selection and prioritization over time as the network gains knowledge and policies regarding choice-making for the best possible results. The evaluation performed by Spieker et al show that RETECS is at least on par, and often better than the other methods, Random, Sorting, and Weighted. This difference in performance would most likely improve over time as RETECS learns which order and selection of tests will yield the fastest feedback to the developers. As the evaluation of RETECS was performed on static data sets, further testing in a live dynamic environment should be performed to assert the actual advantages and drawbacks of the method. The limits of RETECS are hard to identify as the neural network's parameters can be modified and adapted to the project at hand. But a point could be made that the method needs to be applied to a varied set of projects to assert what and where the limits are. Regardless of the limits, the evaluation shows that RETECS is more proficient at learning and improving the selection and prioritization if the test suite has a higher percentage of failed tests.

The *QoDynaQSingle* method shows a lot of promise. It significantly reduces the time until both the first failure and all failures are detected. In their article [8], Zhu et al. claim their work to be novel with regards to dynamic re-prioritization after each test run. This sets the QoDynaQ methods apart from the other discussed techniques. It is also what makes it highly interesting. This makes it highly compatible with other prioritization techniques. Test cases may be given an initial prioritization score by another algorithm, which is then re-evaluated based on their co-failure distribution.

The Bloom filter method attempts to improve regression testing through both logical and technical improvements. The method does show some promise, even if the idea of filtering out one-hit-wonders is relatively simple when compared to the ones discussed in this report. However, Kwon and Ko do concede that the occurrence of one-hit-wonders is not necessarily universal to all data sets, or all parts of a continuous development process. The simplicity of the method is to its advantage though - the Bloom filter method can easily be combined with others presented in this report.

The use of the Bloom filter data stricture is certainly clever, as they are perfectly suited for the fast operations required for the method to function. However, there may very well be other data structures that work just as well - the strength of the method comes more from the logical improvement.

There are some similarities between the RETECS and ROCKET methods. It is very possible that the Weighted method used by Spieker et al during evaluation of RETECS is a variation of ROCKET as they both use weighted sums. As Spieker et al concluded that Weighted is a naive version of RETECS, and that RETECS had better results than the compared methods, it would mean that ROCKET is a naive and inferior version of RETECS.

In the context of regression testing, it is important to discuss whether TCP or RTS is of greater importance. This can be difficult, as comparing the method results directly is essentially impossible. The methods in this report optimize around different variables and operate on different data sets. Whether to focus on TCP or RTS becomes a question of context - in some scenarios, earlier fault detection is of more value than better fault coverage. Factors like average test execution time also affect whether to focus on TCP or RTS. Sometimes, in the case of some methods like ROCKET and RETECS, the distinction between prioritization and selection starts to disappear.

As QoDynaQ focuses on TCP, certain advantages gained from effective selection may be lost. Potentially, some methods could be combined, but this implies that companies would be willing to try out new testing techniques in the first place. According to Minhas et al., new methods from academia do not propagate well into industry [6]. This observation probably greatly reduces the possibility of method combinations.

IV. Conclusion

How do the regression testing techniques discussed in this report compare, in the context of a continuous integration development environment?

In conclusion, all of the methods discussed could potentially improve regression testing in a continuous integration environment, since they all outperform their respective benchmark comparison methods. All methods deal with the problem of TCP and all but CoDynaQ also deal with RTS. CoDynaQ however has the benefit that it could be combined with any of the others, to further increase efficiency. The Bloom filter method also shows potential, also because it can be combined with other regression testing techniques.

In regards of RETECS, the performed evaluation shows that it is a promising method. It is however too early to draw any conclusions whether or not it is superior to the other methods presented in this paper. To make such a statement further testing needs to be done, preferably in a live industrial setting. Testing all four methods on the same data sets would also be a viable method to see which method performs the best test case prioritization and selection.

As as final point, we suspect these new methods, even if proven effective in experimental environments, are unlikely to achieve wide use within industry. As a consequence of this, combinations of the techniques are also probably never going to be implemented in a real-world scenario.

V. Contribution statement

All sections of this document have been co-written by the stated authors, with the exception of section II, the origin of which are shown below.

II-1	*ROCKET*	Keiwan Mosaddegh
II-2	RETECS	Max Strandberg
II-3	CoDynaQ	Erik Stålberg
II-4	Bloom filter method	Emanuel Eriksson

References

[1] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[2] N. Kerzazi and F. Khomh. Factors impacting rapid releases: an industrial case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 61. ACM, 2014.

[3] J.-H. Kwon and I.-Y. Ko. Cost-effective regression testing using bloom filters in continuous integration development environments. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 160–168. IEEE, 2017.

[4] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[5] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*, page 540–543, Sep 2013.

[6] N. M. Minhas, K. Petersen, N. B. Ali, and K. Wnuk. Regression testing goals - view of practitioners and researchers. *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW), Software Engineering Conference Workshops (APSECW), 2017 24th Asia-Pacific, APSECW*, pages 25 – 31, 2017.

[7] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *ISSTA 2017 Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 12–22. ACM, 2017.

[8] Y. Zhu, E. Shihab, and P. C. Rigby. Test re-prioritization in continuous testing environments. pages 69 – 79, 2018.