INSTITUTO SUPERIOR TÉCNICO

PROJECTO MEFT

Keeping Master Green with Machine Learning

Author:
João Lousada

Supervisors:
Prof. Doutor Rui Dilão
Eng. Miguel Ribeiro

Research work performed for the Master in Engineering Physics

at

Research Group Name
Department of Physics

January 3, 2020

1 Introduction

Nowadays, it has become crucial to have an efficient and reliable way to keep track of software changes, specially in large and fast-paced companies [1]. In many of them, source-code *"repositories"* are the preferred choice: offering the agility of having access to all historical code versions developed so far, keeping track of the changes that were made. Due to the sheer amount of changes that are made daily, in these kinds of environments, the probability that one of them creates a conflict is significant, leading to a breakage. This is an undesirable situation, since it may lead to development over a defective *"master"* (mainline hereafter) [1], so it is critical to rapidly detect and patch what caused the breakage.

A mainline is *"green"*, if all build steps (i.e. compilation, unit tests, User Interface tests) are successfully executed for every change point in history, otherwise it is called *"red"*, and keeping it that way is a key factor to achieve maximum performance [1]. However finding a scalable solution, given a constraint of time and computer resources, is a challenging task in order to find a balance between correctness and speed, so there is a high demand for automated techniques to help developers keep the master green.[2]. The strategy is to collect data from a large company and use heuristics to exploit common patterns that can be learned by machine learning algorithms and then, based on that knowledge, estimate where a breakage, most likely, occurred.

There are many ways to configure the architecture of such systems and to manage testing. Therefore, several approaches are compared below. For now, let us start by defining some fundamental concepts needed for the analysis.

1.1 Version Control Systems

In software engineering, version control systems are a mean of keeping track of incremental versions of files and documents. Allowing the user to arbitrarily explore and recall the past changes that lead to that specific version. Usually in these kind of systems, changes are uniquely identified, either by a code number or letter, an associated timestamp and the author.[3]

1.1.1 Repository & Working Copy

The central piece of these systems is the **Repository**. - which is the data structure where all the current and historical files are stored and possibly, remotely, accessible to others (*clients*).[3]

When a client *writes* information, it becomes accessible to others and when one *reads*, obtains information from the repository. The major difference from a server is the capability of remembering every version of the files. With this formulation, a client has the possibility to request any file from any previous version of the system.

While developing, having multiple versions of a project is not very useful, given that most compilers only know how to interpret one code version with a specific file type. So the bridge between the repository and the user is the **working copy**. - a local copy of a particular version, containing files or directories, on which the user is free to work on and later on commit the changes to the repository.[3]

1.1.2 Branches

A repository may be composed of many branches or just one, corresponding to different modification paths. Each one is organized in a time-wise fashion, where one can imagine a time axis, starting at the beginning of a project, with points scattered along it, representing when changes occurred and

[1] a master represents the main history of code versions - i.e. the main branch (described below)

were uploaded from the working copy to the repository - this procedure is called *committing* or *to commit*. Branches can share common ground and diverge at some point generating its own history and later they can be merged back or substitute one another. By default, when a repository is created, it only has one branch, which is called *master branch*. Later, at a given point, the set of files contained in version control might be split into two copies that can be developed at different speeds or with different approaches, independent of each other, giving rise to two branches. Like it is depicted in the following example [4]:

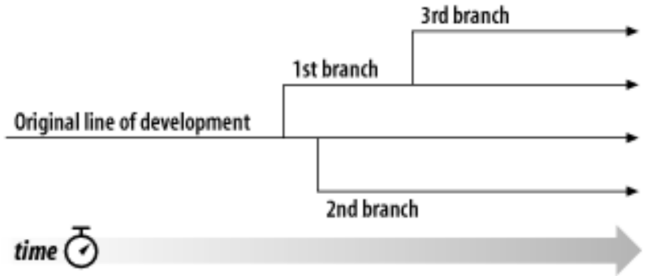

FIGURE 1: Branches of development [4]

This way, it is easy to keep track of the progress made so far, even in multiple directions, but most importantly it is mistake tolerant. For example, a developer starts writing code in a particular way and later on realises that there are mistakes, or her point of view of the problem was not the most appropriate. There are two solutions, scrap the progress and start with a different strategy, or try to reverse all the changes manually to end up at the start, which is dull, error-prone and counterproductive. With version control it is simple to "load" any version and go back to any past point. In some cases, from the developers and teams point of view, not having to worry about "messing up" and having the ability to make changes without a great deal of compromise, provides the right safety net to tackle problems from different angles and adapt to changing environments, without wasting resources. On the other hand, some projects cannot afford a "messing up" scenario and other precautions need to be taken into account.

1.2 Mono-repositories vs. Multi-repositories

Some projects are very large so we can have two strategies to store all the code. Google generates N lines of code daily and their strategy is to ...

In order to manage and store amount of new code produced daily, there are two possible ways of organizing repositories: one single giant repository that encompasses every project or assigning one repository for each one. Google's strategy is to store billions of lines of code in one single giant repository, rather than having multiple repositories. [5] A mono-repository is defined such that it encompasses the following properties:

- **Centralization** - one code base for every project.
- **Visibility** - Code accessible and searchable by everyone.
- **Synchronization** - Changes are committed to the mainline.
- **Completeness** - Any project in the repository can only be built from dependencies that are also part of it.
- **Standardization** - developers communalize the set of tools and methodologies to interact with code.

This gives, the vast number of developers working at Google, the ability to access a centralized version of the code base - *"one single source of truth"*, foment code sharing and recycling, increase in development velocity, intertwine dependencies between projects and platforms, encourage cross-functionality in teams, broad boundaries regarding code ownership, enhance code visibility and many more. However, giant repositories may imply less autonomy and more compliance to the tools used and the dependencies between projects. Also developers claim to be overwhelmed by its size and complexity.

A multi-repository is one where the code is divided by projects. With the advantages that engineers can have the possibility of choosing, with more flexibility, the tools/methodologies with which they feel more at ease; the inter dependencies are reduced, providing more code stability, possibly accelerating development. However, the major con arises from the lack of necessity of synchronization, causing version problems and inconsistencies, for example two distinct projects that rely on the same library.

Although electing a preferred candidate is still a matter of debate, mono-repositories are linked to less flexibility and autonomy when compared to multi-repositories, but there is the gain of consistency, quality and also the culture of the company is reinforced, by unifying the tools used to link the source code to the engineer.[6]

1.3 Continuous Integration and Testing

Continuous Integration (CI) is a popular software development technique that allows developers to easily check that their code can build successfully and pass tests across various system environments.[3] However, it is not straightforward to develop and configure this methodology, there are numerous approaches and several ways to solve the same problem, which are explored below. The key aspect is to navigate through the options of how to commit and how to test, compare them and acknowledge what features can be leveraged at the expense of others, making it possible to find a scalable solution.

In testing, associated with each repository, there is a battery of tests that makes sure it is safe to integrate a change in the system, but what if we are in the case of applying the same tests twice or more ? What if a test is not adequate to the type of change a developer did? What are the boundaries of test quality? If a test always passes, is it a good test ? Should one apply tests in an orderly way, prioritizing some tests by time or relevance criteria ?

Lets take a step back and define what testing is. Testing is a verification method to assess the quality of a given software. Although, this sentence is quite valid, it is vague, in the sense that it does not define what quality software actually is. In some contexts, quality might just refer to simply code compiling with no syntax errors or "mistakes". When tests are applied, the outcome obtained is of the form PASS/FAIL, with the purpose of verifying functionality or detecting errors, receiving quick and easy to interpret feedback. However, the connections between testing and quality are thin: testing is very much like sticking pins into a doll, to cover its whole surface a lot of tests are needed. For example, running a battery of tests (test suite) where every single one of them yields PASS. This can only mean two things: whether the code is immaculate or some scenarios were left out of the process. Usually, test suites are constructed from failed tests. When in a FAIL situation, i.e. fault detection or removal , a new test case is created, preventing this type of error to slip again in the future, incrementing the existing test suite, in a "never ending" process). So it is correct to say, failed tests are a measure of non-quality, meaning there is no recipe for assuring a software is delivered without flaws. [7]

1.4 Regression Testing

Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the working copy do not conflict with the existing features.

In a nutshell, whenever new features are added to an existing software system, not only the new features should be tested, but also the existing ones should be tested to ensure that their behaviours were not affected by the modifications. Usually this is done by applying test cases, to check if those features, in fact work or are still working. Therefore, this field encapsulates the task of managing a pool of tests, that are repeatedly applied across multiple platforms. [8]

The problem relies on the fact that tests do not run instantaneously and as software systems become more complex, test pools are ought to grow larger, increasing the cost of regression testing, to a point where it becomes infeasible, due to an elevated consumption of computer and time resources, so there is a high demand to search for automated heuristics that reduce this cost, improving regression testing. In this work, three solutions are proposed that can lead to possible substantial performance improvements:

- **Test Case Selection** - "do smarter" approach, selecting only relevant tests, given the type of change.
- **Test Suite Minimisation** - "do fewer" approach, removing possible redundancies.
- **Test Case Prioritisation** - also "do smarter" approach by running some tests first, increasing probability of early detection. [8]

In terms of notation, let us denote P as the current version of the program under test, P' as the next version of P. T is the test suite and individual tests are denoted by a lower case letter: t. Finally, $P(t)$ is the execution of test t in the system version P.

1.4.1 Test suite minimisation

Definition 1.1. Given a test suite T, a set of test requirements $R = r_1, ..., r_n$ that must be satisfied to yield the desired "adequate" testing, and subsets of T , $T_1, ..., T_n$, each one associated with the set of requirements, such that any test case t_j belonging to T_i can be used to achieve requirement r_i

The goal is to try to find a subset T' of T: $T' \subseteq T$, that satisfies all testing requirements in R. A requirement, r_i is attained by any test case, t_j belonging to T_i. So a possible solution might be the union of test cases t_j in T_i's that satisfy each r_i (*hitting set*). Moreover, the hitting set can be minimised, to avoid redundancies, this becoming a "minimal set cover problem" or, equivalently, "hitting set problem", that can be represented as a *Bipartite graph*[2]. The minimal hitting-set problem is a NP-complete problem (whose solutions can be verified in polynomial time). [9]

Another aspect to consider is that test suite minimisation is not a static process, it is temporary and it has to be *modification-aware*. - given the type of modification and version of the code, the hitting set is minimised again. [8]

1.4.2 Test Case Selection

Definition 1.2. Given a program P, the version of P that suffered a modification, P', and a test suite T, find a subset of T, named T' with which to test P'.

Ideally, the choice of T', should contain all the *fault-revealing* test cases in T, which are obtained by unveiling the modifications made from P to P'. Formally:

Definition 1.3. Modification-revealing test case A test case t is said to be modification-revealing for P and P' if and only if the output of $P(t) \neq P'(t)$. [10]

[2]s a graph whose vertices can be divided into two disjoint and independent sets U and V, such that every edge connects a vertex in U to one in V[9]

After finding the nature of the modifications, in simple terms, one has a starting point to select the more appropriate test cases. Both test case minimisation and selection rely on choosing an appropriate subset from the test suite, however they do so, often, with different criteria: in the first case the primal intent is to apply the minimal amount of tests without compromising code coverage in a single version, eliminating permanently the unnecessary tests from the test suite, as in the second the concern is related directly to the changes made from one previous version to the current one. [8]

1.4.3 Test case prioritisation

Definition 1.4. Given a test suite T, the set containing the permutations of T, PT, and a function from PT to real numbers $f : PT \rightarrow \mathbb{R}$, find a subset T' such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

Where f is a real function that evaluates the obtained subset in terms of a selected criteria: code coverage, early or late fault detection, etc [8]. For example, having five test cases (A-B-C-D-E), that detect a total of 10 faults, there are $5! = 120$ possible ordering, one possible way of choosing a permutation, is to compute the metric Average Percentage of Fault Detection (APFD) [11].

Definition 1.5. APFD Let T be a test suite containing n test cases and F the set of m faults revealed by T. Let TF_i be the order of the first test case that reveals the i^{th} fault.[8]

$$APFD = 1 - \frac{TF_1 + \cdots + TF_n}{nm} + \frac{1}{2n}$$

Simply, higher values of APFD, imply high fault detection rates, by running few tests, as can be seen in the following plot:

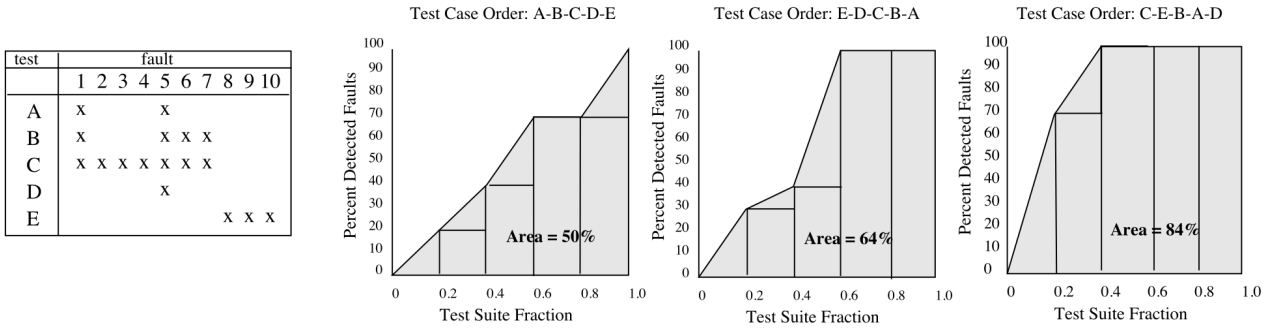

FIGURE 2: Average percentage of fault detection [11]

The area below the plotted line is interpreted as the percentage of detected faults against the number of executed test cases, maximizing APFD, demonstrating clear results in detecting early. The order of test cases that maximizes APFD, with $APFD = 84\%$ is **C-E-B-A-D**, being preferable over other permutations.

By detecting faults early, the risk of developing over faulty code is reduced, causing minimal breakage in productivity avoiding, for example, possible delayed feature releases.[1]

2 Objectives

The objectives delineated for this work are:

- Detect common usage patterns, in a controlled environment, by generating synthetic data.
  - Learn heuristics to automate fault detection process

- Optimize regression testing systems using real world data.
  - Analyse how different system configurations affect Continuous Integration
  - Reduce fault detection time
  - Provide a Live-Estimate of the Status of a Project
  - Given a commit, choose which chain of tests minimize pass/fail uncertainty

3 Planning

With the following Gantt Chart it is shown a rough estimate of the work timeline to be developed over the next months, containing the most relevant tasks and important milestones.

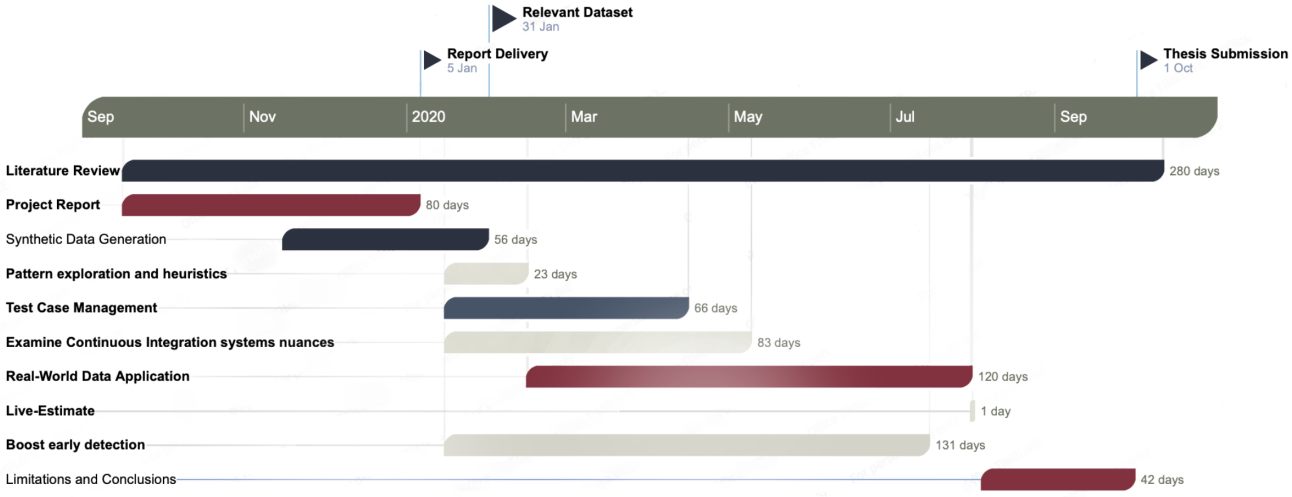

FIGURE 3: Master-Thesis Chronology

4 Background

In this section, the goal is to provide a solid comparison between the different strategies that can be adopted to make sure source-code repositories are a manageable choice.

4.1 Naive-approach

The simplest approach is to run every test for every single commit. As teams get bigger, both the number of commits and the numbers of tests grow linearly, so this solution does not scale. Also, as [12] stated, computer resources grow quadratically with two multiplicative linear factors, making continuous integration a very expensive and not eco-friendly process in terms of resources and time.

For example, Google's source code is of the order of 2 billion lines of code and, on average, 40000 code changes are committed daily to Google's repository, adding to 15 million lines of code, affecting 250000 files every week, having the need to come up with new and innovative strategies to handle the problematic.[2]

4.2 Cumulative Approach

One possible solution to contour the large amount of time spent on testing and waiting for results is to accumulate all the commits made during working hours and test the last version, periodically, lets say during the night. This enables teams to develop at a constant speed without having the concern of breaking the master branch, but rather worrying about it in the future. In this case, the lag between making a commit and knowing if it passed or fail is very large, running the risk of stacking multiple mistakes on top of each other.

Using this approach, two possible outcomes can occur: either the build passes every test and the change is integrated, or it fails, causing a regression. Here, we encounter another crossroads, as the batch was tested as a cumulative result, we have no information whatsoever of what commit, exactly, caused the regression. This problem can be solved by applying search algorithms:

- **Sequential Search** - re-testing every commit individually with time complexity is O(n), where n, in this case, is the number of individual parts (length of list to order).
- **Binary Search** - instead of running every option, divide the list in half and test the obtained subset, if pass, the mistake is located in the other half, if fail, split the subset and test again until narrowing it down to a specific change, this approach is O(log n) complex.

After finding out which commit caused the regression, it can be fixed or rolled back to the version before the change was introduced, meaning that possibly the progress made during that working day has to be scrapped and re-done, postponing deadlines and, at a human level, lowering down motivational levels, with developers being "afraid" of making changes that will propagate and will only be detected later on, so allowing a large turn-around time may results in critical hampering of productivity.

In comparison with naive-approach, cumulative approach is more scalable in the way that it avoids testing every single commit made, only trying to test the last of the day, spending more time trying to trace back the origin of the failure. So in conclusion, we forfeit absolute correctness to gain speed and pragmatism, although this solution's computer resources grow linearly with the number and time of tests.

4.3 Change-List Approach

In this case, changes are compiled and built in a serialized manner, each change is atomic and denoted as a Change List (CL). Clarifying, lets take a look at Google's example of workflow in Figure 4.

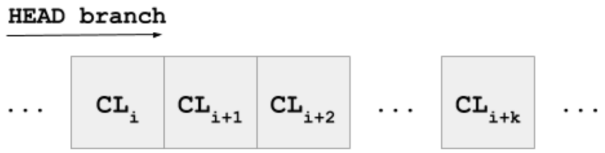

FIGURE 4: Code repository and workflow at Google. Each CL is submitted to master branch or HEAD [2]

Individually, tests validate each CL and if the outcome is pass, the change is incorporated into the master branch, resulting into a new version of the code. Otherwise, a regression is caused. For example, imagine that CL_i introduces a regression, causing failure on the consecutive system versions. Possible fixes and tips: investigate the modifications in CL_i, to obtain clues about the root cause of the regression; Debug the fail tests at CL_i, which is preferable to debug at later version CL_{i+k}, whereas after CL_i

code volume might increase or mutate, misleading conclusions; revert or roll back the repository, until the point where it was "green", in this case CL_{i-1}.[2]

The novelty is introduced in test validation:

- **Presubmit tests** - Prior to CL submission, preliminary relevant small/medium tests are run, if all pass the submission proceeds, else it gets rejected. So instead of having to wait until all tests are complete, the developer has a quick estimate of the status of the project.
- **Postsubmit tests** - After CL submission, large/enormous tests are executed and if they indeed cause a regression, we are again in the situation where we have to find exactly what CL broke the build and, possibly, proceed in the same manner as Cumulative Approach to detect what version caused the regression. [2]

Nevertheless there are some major issues: while presubmit testing, the sole focus is to detect flaws on individual changes and not on concurrent one. *Concurrency* is the term referred to builds that run in parallel with each other, meaning that 2 developers can be committing changes of the same file, that are in conflict with each other, or with other files. Therefore, after precommit phase, there is the need to check all the affected dependencies of every commit, quickly identify were the conflict emerged and eventually roll back in the case of a red master, which can only be done in postcommit stage [1] and when a postcommit test fails: a 45 minute test, can potentially take 7.5 hours to pinpoint which CL is faulty, given a 1000 CL search window, so automation techniques, quickly putting them to work and returning to a green branch is pivotal. [2]

To better understand the concept of concurrency and the implications it brings, lets consider there are n concurrent changes in a batch, where all of them pass precommit phase: (1) committing changes 1 to $n-1$ do not lead to breakage, (2) applying the n^{th} change does not break the master, and (3) putting all the changes together breaks the mainline. Which might indicate that the n^{th} change is in conflict with the rest. Concurrency is practically unavoidable and it is more likely to happen when parts of the code that are interconnected are being altered. Frequent mainline synchronization is a possible path to diminish this effect, forcing developers to always work on the most updated version.[1]

4.4 Always Green Master Approach

After steering clear of analysing every single change with every single test, or accumulating all the changes and testing them , splitting it into smaller individualized chunks, cherry-picking an ideal and non-redundant subset of tests, prioritizing them by size making it possible to divide test phase into precommit and postcommit, and enabling a status estimate of the likelihood of regressions, [1] proposes that an always green mainline is the priority condition, so that productivity is not hampered. To guarantee an always green master, Uber designed a scalable tool called *SubmitQueue*, where every change is enqueued, tested and, only later on, integrated in the mainline branch if it is safe. The interesting aspect is which changes get tested first, while ensuring serializability. Uber resorts to a probabilistic model that can speculate on the likelihood of a given change to be integrated. - i.e. pass all build steps. Additionally, to agilize this process this service contains a *Conflict Analyzer* to trim down concurrent dependencies.

Probabilistic Speculation - Based on a probabilistic model, powered by logistic regression, it is possible to have an estimate on the outcome a given change will have and with that information select the ones that are more likely to succeed, although other criteria might be chosen. To select the more likely to succeed changes, this mechanism builds a *speculation tree*, which is a binary decision tree. Imagine a situation where two changes, C_1 and C_2, need to be integrated in the master branch M. C_1 is merged into M is denoted by $M \oplus C_1$ and the build steps for that change are denoted by B_1. B_{12} represents builds for $M \oplus C_1 \oplus C_2$. Speculating that B_1 has high probability of being integrated, it is useful to test

it in parallel with B_{12}, because if the assumption is correct, C_1 is integrated and now this result can be used to determine if C_2 also passes, without testing each one individually and then together. If the speculation is not correct, C_1 is not merged and C_2 has to build separately by running B_2. In the case of three pending changes C_1,C_2 and C_3, the speculation tree is of the form of Figure 5. One possible way of handling the result from this tree would be to speculate everything, assuming that the probability of merging a change is equal to the probability of rejecting it. Having 3 pending changes, one would have to perform 7 builds (number of nodes in the tree), under this assumption. But it is unnecessary to speculate all of the builds, if B_1 is successful, B_2 now becomes irrelevant and the tree is trimmed down considerably. Only n out of $2^n - 1$ builds are needed to commit n pending changes [1]. Additionally, the assumption that every change depends on one another is made, but this is not always the case, two changes can be totally distinct. By detecting independent changes, a lot of time can be saved by building them individually and committing them in parallel. This is done by the Conflict Analyzer. This approach combining the selection of most likely to pass builds, using machine learning models, parallel execution, the identification of independent changes, trimming down speculation tree size is able to scale thousands of daily commits to a monolithic repository. Finally, a comparison of all explored approaches is shown in the Table below.

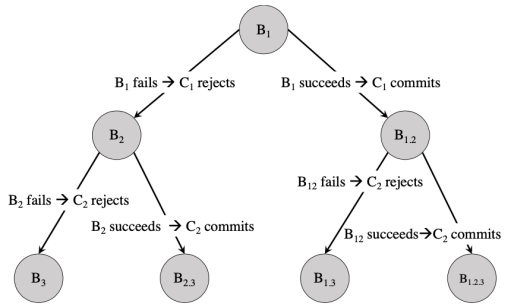

FIGURE 5: Speculation tree - builds and outcomes [1]

Approach	Correctness	Speed	Scales?
Naive	Very High	Very Low	No
Cumulative	Medium	Low	No
Change-List	Medium	Medium	Yes
Always Green Master	High	High	Yes

TABLE 1: Comparative analysis between strategies

5 Synthetic Data Generation

So far, after reviewing the state-of-the-art, the strategy is focused on trying to mimic a real life environment and try to find correlations between commits and regressions, by creating a data set that contains: (1) A list of commits, containing information of what/when files have been modified, and (2) a list of tests, with the associated duration and files covered. After that, we can experiment the different approaches explored above and combine some aspects to efficiently have a scalable Continuous Integration system. Firstly, the determination of automated techniques that facilitate an early detection, for example:

- **Commit List**

  - when someone makes a change that caused a regression, subsequent commits are prone to affect the same tests

  - some tests might be an indicator of whether other tests will fail

  - keeping a score board, based on past-history, to estimate the probability a given developer has of making a faulty committing.

- **Test List**

  - modifying a file might trigger off faults on files that depend on the first

  - some files affect many tests, but can be very stable and not cause failures often. - (for instance standard libraries)

  - other files, that affect few tests, are harder to cover and validate. - (like an innovative feature)

Finally, a test run list is generated to analyse the results of the approaches, this list is expected to contain a live-estimate of the status of the project as tests are being run. At the beginning, the value for the uncertainty that commit will pass has its maximum value and the goal is to choose what test to apply next, in order to minimize this uncertainty and give the estimate between a confidence value, being more accurate as more tests are applied. This way it is possible to keep up with the projects' progress as time evolves.

References

[1] Sundaram Ananthanarayanan et al. "Keeping Master Green at Scale". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. This source encompasses Uber's repository configuration and explains what's behind their fault detection process. ACM, 2019, 29:1–29:15.

[2] Celal Ziftci and Jim Reardon. "Who Broke the Build?: Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale". In: This source encompasses Google's repository configuration and explains what's behind their fault detection process. IEEE Press, 2017, pp. 113–122.

[3] Mark Santolucito et al. *Statically Verifying Continuous Integration Configurations*. Paper on technical issues regarding Continuous Integration systems. 2018. arXiv: 1805.04473.

[4] Michael Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., 2004.

[5] Rachel Potvin and Josh Levenberg. "Why Google Stores Billions of Lines of Code in a Single Repository". In: *Commun. ACM* (2016), pp. 78–87. ISSN: 0001-0782.

[6] Ciera Jaspan et al. "Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '18. This source compares and draws conclusion on whether it is more advantajous to prefer a monolithic repository over a multi-repository, by conducting a survey to Google's engineers. ACM, 2018, pp. 225–234.

[7] B. Meyer. "Seven Principles of Software Testing". In: *Computer* (2008). Introduction on the fundamental pilars of software testing in a general way, useful in defining concepts., pp. 99–101.

[8] Yoo Shin. "Extending the Boundaries in Regression Testing: Complexity, Latency, and Expertise". PhD thesis. King's College London, 2009.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[10] Gregg Rothermel and Mary Jean Harrold. "A Framework for Evaluating Regression Test Selection Techniques". In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. This source explores in detail the topic of test selection. IEEE Computer Society Press, 1994, pp. 201–210.

[11] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. "Prioritizing Test Cases For Regression Testing". In: *IEEE Trans. Softw. Eng.* (). This source explores in detail the topic of test prioritisation.

[12] Atif Memon et al. "Taming Google-scale Continuous Testing". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP '17. IEEE Press, 2017, pp. 233–242.