Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Datateknik
2023 | LIU-IDA/LITH-EX-A--23/002--SE

Dynamic Test Scope Selection in Continuous Integration Loop

Gabraiel Bahnan

Supervisor : Martin Sjölund
Examiner : Kristian Sandahl

External supervisor : Fredrik Linderbäck, Ludvig Heinebäck

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Continuous integration (CI) is a critical step in big software projects, and the telecommunications industry relies on it in 5G network deployment. The testing infrastructure during the implementation is significantly challenging, and the developers need to execute fewer tests to get faster feedback about their changes in the code. There are many different test case selection methods where techniques based on test history are commonly used in test prioritization or decrease the test scope.

This thesis is about researching and implementing two dynamic models in Ericsson's environment to minimize integration test scope after code changes. One is based on dependencies between classes, and the second is based on the historical record of test executions. The TCS (Test Case Selector) is evaluated concerning time and fault detection effectiveness compared with the currently implemented technique. The results show that the model based on dependencies did not improve, while the model based on historical test executions shows impressive improvements in increasing the fault detection rate. That increases testing efficiency with less or the same time as the current technique, where only the most effective and essential test suites are being executed.

Contents

List of Figures

List of Tables

1 Introduction

Mobile networks are widespread over the globe and serve as the foundation for a connected society; they are spreading over the world at a quicker rate than any other communication technology [1]. Because communication is at the center of human activity in all aspects of life, the arrival of this technology, which allows multi-modal connection from everywhere to any place with adequate infrastructure, raises a slew of basic questions [2]. Ensuring network stability and operation is critical to today and future mobile networks. Continuous integration and deployment are essential for delivering high-quality new features on time. Choosing the proper test scope and balancing test coverage against time and resource allocation for accurate testing is a difficult task [3].

1.1 Motivation

Change occurs at an ever-increasing rate, at practically every angle, and in an endless cycle. Once the new code is pushed into an existing software platform, it will be thoroughly tested. Tests can be performed at various stages and in multiple environments to ensure that the code is fully operational. Continuous integration (CI) and continuous delivery (CD) systems often drive development teams to execute minor code changes and check them into a version control repository [4]. Most modern major large projects include developing software on several platforms and tools, necessitating a standardized method for integrating and validating changes. Because of the automated integration, these CI/CDs enable increased productivity, but the tests may take a lot of time and computational resources that could be utilized for anything else [4] [5].

Ericsson is an important information and communication technology company around the globe. Ericsson has around 14,000 employees in Sweden, representing over 100 nationalities [6], who work in various company sectors, including research, development, sales, production, and administration. The firm works with networks and digital services such as 5G and virtual reality.[1] Development teams use Continuous Integration (CI) technology to merge new code changes into the current codebase. Code changes often occur regularly, necessitating integration testing on various hardware. Since their code bases are enormous

[1] https://www.ericsson.com/en/about-us

and have a large number of tests associated with them, running and executing all of them in Ericsson's Multi-Standard Radio Networks (MS-RAN) could take a long time due to the large amount of computing power and resources required, making the development process too costly.

For this purpose, a company with a large project must make committing to the codebase feasible to merge often and comply with the rapid pace at which developers integrate their code. The software industry's continuous integration flow has arisen to address this demand. A CI flow is a computer-assisted tool that conducts many sorts of tests. The checking is often done on code changes and commits by constructing the project, running various tests, and statically parsing the code using a set of syntactic rules [5]. Various technologies have evolved to meet the demands of a CI flow, ranging in different ways to solutions or more complicated flows, all of which offer the individuals responsible for developing and maintaining flow flexibility [4].

The problem is that there is no method in the functionality testing department to decide which tests are the most critical to run in the CI flow for a particular code update in the Environment Delivery Check (EDC) testing level fig. 2.3, which runs a test scope on the latest released repo product to ensure its stability. The only options are to conduct all of the tests or none at all [7], or follow the current static option by running the same two test suites every time. For each update and commit, the EDC takes many hours, which is why there is a need to create or find a dynamic way to choose only relevant EDC tests when pushing new changes in the codebase.

1.2 Aim

Ericsson would gain immensely from using dynamic and accurate test scopes in the 5G traffic testing domain. The company currently operates a static test scope to safeguard its test code. This thesis aims to investigate existing methods and strategies for dynamically selecting a suitable test scope based on new code updates to determine and assess a better dynamic solution. The primary aim of the test case selection process would be to run fewer tests, which would save time and resources.

1.3 Research question

To fulfill the aim, the following research question has been defined:

> Which strategy fits for dynamically selecting a suitable test scope based on new code changes?

1.4 Delimitations

Software testing may be divided into numerous areas, such as performance testing and security testing. The goal of this thesis is testing in which the purpose is to verify the newly pushed code based on new changes. This thesis focuses exclusively on dynamic testing solutions, as opposed to static testing, which may be performed manually or automatically with the help of tools. Every code modification at Ericsson is subject to several phases (pre-merge, post-merge, EDC, MJE-JCAT-BUNDLE, NBC, and RBC) see fig. 2.3. This thesis is done in Ericsson's environment and aims to improve the Environment Delivery Check phase (EDC).

2 Theory

This chapter presents the background to the thesis topic by defining the concepts that are used in the project to provide a deeper understanding. Some of these concepts are related only to Ericsson, and others are external open sources that anyone could use.

2.1 Continuous integration

Continuous integration is a software development technique utilized at Ericsson that is used in cases where programmers should integrate code into a shared repository often, up to multiple times each day. The IT team traditionally creates and distributes new IT applications rapidly, while the Communication Network team provides more standard features. They seem to exist in parallel universes, but the standards for network applications, for example, are more strict than for other IT applications [8]. In a continuous improvement process, CI empowers IT and network teams to communicate new ways and benefit from one another's knowledge. For network developers, CI has the potential to alter the development process. Throughout the cycle, verification and validation are automated. The testing data, together with continuous feedback from developers who work with the network, results in software that is more focused on the demands of network developers and is built and published faster [9].

CD (Continuous Deployment) is the practice of automatically deploying code changes to a production environment as soon as they are committed to the repository. This allows for faster delivery of new features and bug fixes to customers, as well as reducing the risk of human error during manual deployment processes [10]. Ericsson began relying on CD in 2015 in some places, like the United States, to replace older versions of validation and verification. A 5G network is vast in terms of complexity and choices. Software development cycles are often 3–4 weeks rather than 6–12 months [10]. Communication network operators must use a very agile technique like CI/CD to stay current. The CI/CD automates the different phases of the lifecycle, see fig. 2.1.

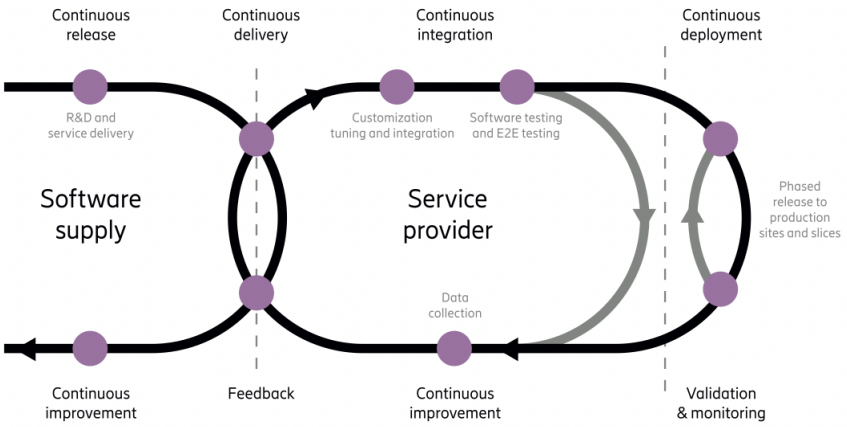


Figure 2.1: End-to-end software pipline [8].

2.2 Jenkins

Jenkins[1] is a Java-based open source automation server with plugins designed for continuous integration, and it provides support for version control tools such as Git to support any project. Jenkins facilitates continuous integration and delivery by automating software development elements linked to complete lifecycle processes such as building, testing, and deploying. Jenkins makes it simpler for developers to merge changes into the project and for users to get a new build. With the help of plugins, Jenkins can also execute Maven-based projects and projects based on other programming languages.

2.3 Gerrit

Gerrit[2] is a highly flexible collaboration tool for software developers that works with the Git version control system. Gerrit is a web-based version control system that focuses on providing a platform for developers' teams to review each other's source code modifications. It is an open-source project adopted from Rietveld, a similar code review tool.

After a developer pushes code to Gerrit, all changes are pushed to Pending Changes for other repository owners to evaluate and debate; see fig. 2.2. Gerrit is constructed to require a favorable evaluation of the code from at least one other developer before it can be submitted to the code base.

2.4 Maven

Maven[3] is a project management and comprehension tool for software projects. In other words, it facilitates and standardizes the project development process. It's built on the idea of a project object model (POM). Maven can handle the team's reporting, distribution, and documentation. It also helps the developers by automating the construction of the source folder structure, as well as the sorting and testing of the final outcome. The POM is an XML file that contains information about the project's build configuration and is normally found at the project's root. The POM contains the project's dependencies, their versions, and more detailed configurations of the tools used for build automation. Plugins that can be performed and build profiles are some examples of tools used for build automation.

[1] https://www.jenkins.io/doc/
[2] https://gerrit-review.googlesource.com/Documentation/intro-how-gerrit-works.html
[3] https://maven.apache.org/what-is-maven.html

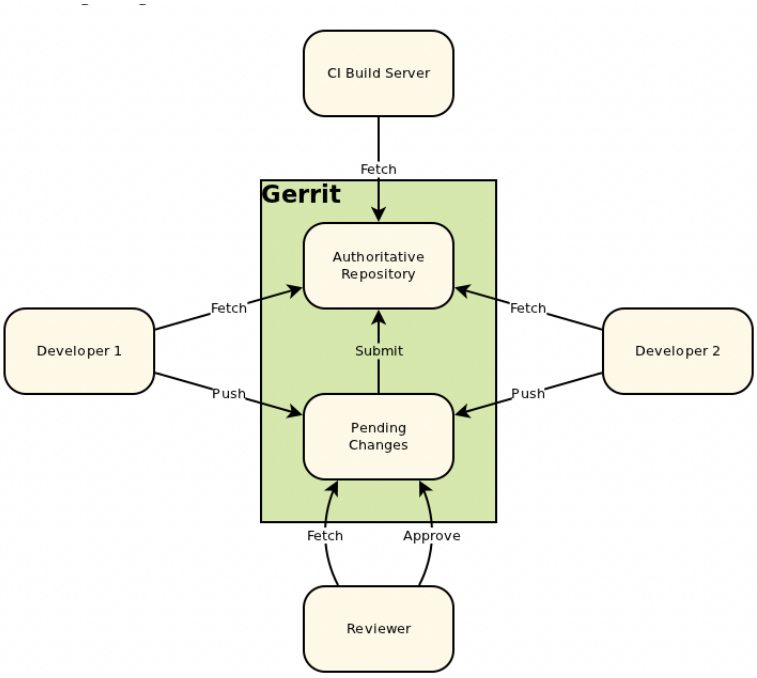


Figure 2.2: Gerrit as the central repository [11].

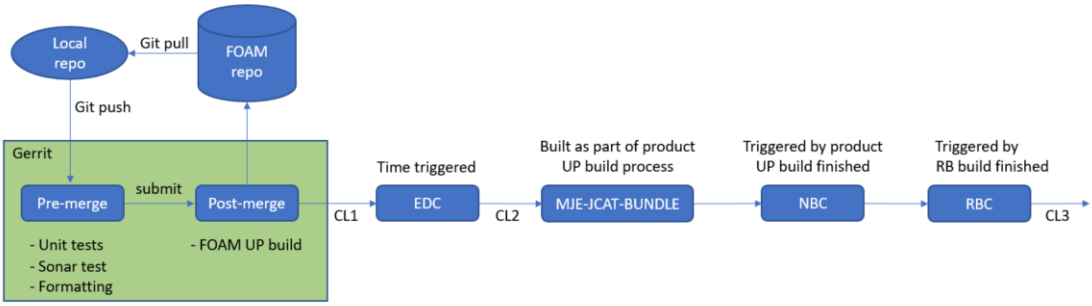


Figure 2.3: MJE CI flow.

2.5 MJE Continuous integration flow

MJE is Ericsson's internal concept, it is a test execution framework of Java Common Auto Tester (JCAT) that executes Java-based test methods (TestNG-style) that interact with nodes and test tool part of the multi-standard Radio Networks. MJE CI flow is an automated SDT (Software Development Test) and SDC (Source Delivery Check) solution for the MJE product, which performs different types of checking, see fig. 2.3. The SDT part, which is called the MJE PreMerge flow, is the CI for a developer where it helps to do automatic code review when pushing new code changes to Gerrit. PreMerge has unit tests that the framework's developers build, and this flow is implemented with Jenkins Pipeline. The SDC part has CI triggered automatically during delivery. Which contains three flows – the MJE PostMerge flow, the PostMerge downstream flow, and the MJE PostRelease flow. These flows are implemented with Jenkins Pipeline and include unit and integration tests.

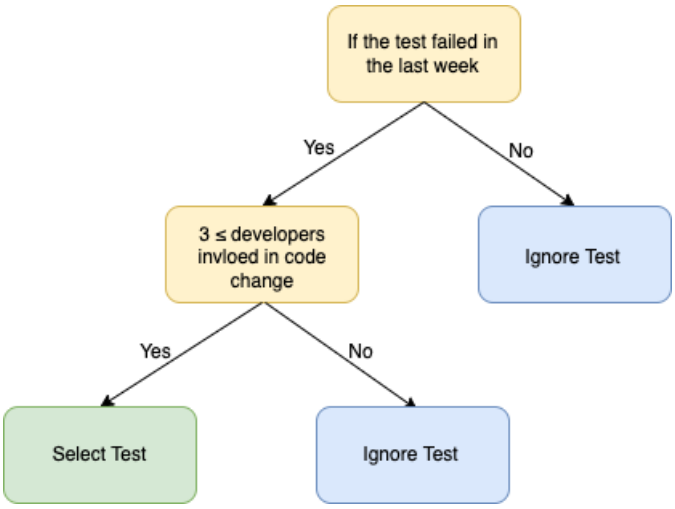


Figure 2.4: Decision tree.

2.6 Tree-based Algorithms

Navada et al. [12] introduced a decision tree algorithm, which is the primary form of tree-based algorithms. The algorithm is used to develop a training model in which test cases on input data patterns are represented as nodes, and the predicted classes of these patterns are defined as leaf nodes. To acquire the correct output for the input pattern, the algorithm classifies the tests by sorting them down through the tree. Figure 2.4 is an example of a decision tree showing the preference for a particular test. The above attribute of this tree is if the test failed in the previous week. If true, the number of developers is split into a range based on the number of requests.

Random forest is another tree-based algorithm that selects a subset of tests from the test set to train the decision tree [12]. In the case of regression, each tree provides a classification score for itself, which we call "voting". Assume the training set in fig. 2.5 has a certain number of cases. It generates several data subsets by choosing a randomly trained sample. For each sample chosen, it will create a decision tree. Then, each decision tree constructed will acquire a prediction result. Following that, voting will be conducted for each predicted outcome. Finally, Random Forest selects the classification with the highest votes and averages the regression results from different trees. Random Forest is useful in test suite selection because it is able to handle large amounts of data, handle missing values and outliers, and is less prone to overfitting than other algorithms.

2.7 Support Vector Machine

Stanislaw et al. [13] discuss how to select the optimal features using a genetic algorithm (GA) and a Support Vector Machine classifier (SVM). SVM is a powerful solution that can be used for both classification and regression issues. SVM's purpose is to find the optimum line or decision boundary that can be used to divide n-dimensional space into classes (where n is the number of features you have) so that we can simply print every data item as a point in n-dimensional space, see fig. 2.6. The optimal decision boundary is referred to as a hyperplane, and it aids in the classification of the data points. SVMs conduct classification by finding the extreme points/vectors that aid in the construction of the hyperplane. In addition, the number of features determines the size of the hyperplane's dimension.

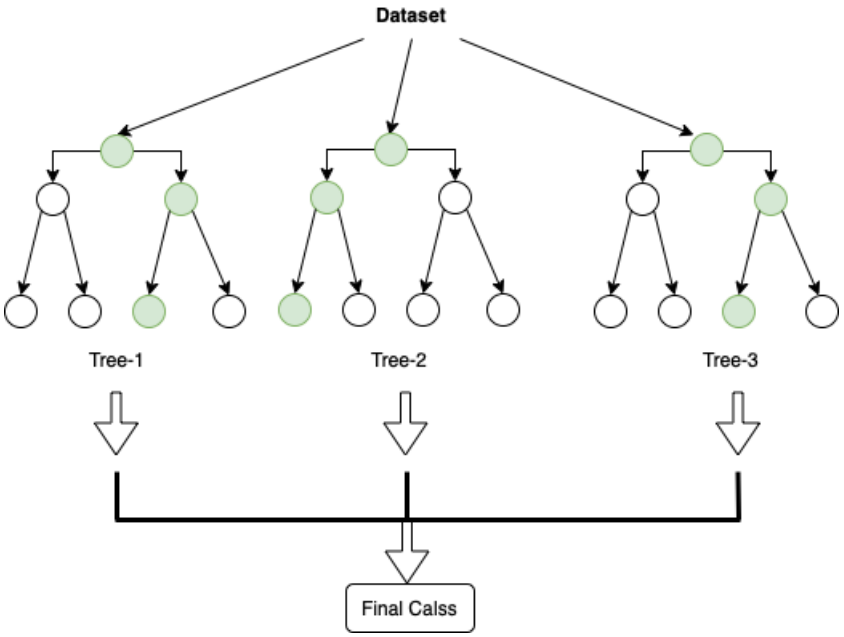


Figure 2.5: Random forest simplified.

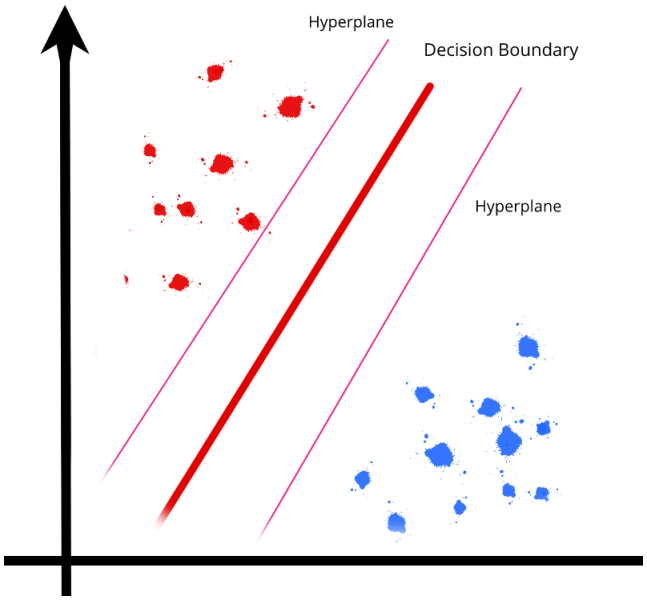


Figure 2.6: Support vector machine classifier [14].

3 Related Work

This chapter presents related work to the thesis topic by introducing some algorithms used in selecting a test scope. Some algorithms depend on machine learning, and others do not. Part of them could be used in the final design or to get inspiration. The represented algorithms are adopted from literature research and it had been selected those that can be reused in the project.

3.1 Regression Test Selection

Regression test selection (RTS), an approach discussed by Gligoric et al. [15], decreases regression testing time by selecting and executing a subset of the main suite of test cases. Finding dependencies between code and test cases is the first step in the classic RTS technique, followed by detecting the modification files. Some systems have incorporated RTS, like the Google TAP system. The approach has a dynamic technique called Ekstazi. It can be implemented in the Maven build system based on file and class dependencies as well as external resources. Unlike most other RTS techniques, Ekstazi does not need interaction with version-control systems and therefore does not specifically compare old, and new versions [16]. Instead, Ekstazi determines which files each test class depends on; in this case, files could be classes in Java, for example. If none of the test class' dependent files have changed in the current version, the test class does not need to be run.

Ekstazi dynamically saves dependencies in a simple structure similar to that used by build tools, followed by the names and checksums of these classes [15] [17]. The checksum hashes the content of the files, allowing Ekstazi to review changes without having open access to the previous version. For each test class, the information is saved in a separate file. Ekstazi verifies that the checksums of all dependent files are still the same for each test class, and if this is true, the test class is therefore not selected. Because there is no dependency information for just about any class on the first execution of Ekstazi, all of them would be selected, even freshly created test classes. Ekstazi first determines which test classes should not be executed, then runs the remaining test classes to see if they still pass or fail.

3.2 Test Prioritization

Busjaeger et al. present a novel test prioritizing approach that integrates many effective methods into a systematic machine learning framework to rank them [18]. Before committing code changes, test prioritization may be used to run a subset of the most likely failed tests. This provides engineers with immediate feedback before changing context and may help avoid significant regressions from reaching the baseline. Also, after committing code changes, test prioritization may be used to execute tests in the order of failure probability. The model uses the open-source Java Code Coverage Library (JaCoCo)[1] for measuring code coverage in a codebase, which avoids the need to rewrite source code [19]. The model has an inverted index that links each source file to the line numbers performed from every test to allow efficient measurement of coverage score. It also contains text similarity ratings for the location and content of the test file. For a specific test, the text score is represented by the formula cosine similarity among words in the test and words in the modified file. Two additional features unaffected by the change are the age of a test and its failure history.

Another method called Prioritization for Continuous Regression Testing (ROCKET) by Marijan et al [20], which based on test history data and time constraints, can choose and rank a subset of test cases that can discover the majority of regression faults. The ROCKET method does not need source code. It requires four inputs: a test suite (a collection of test cases), a failure status, the amount of time it took to execute each test case across numerous previous test runs, and the maximum amount of time allowed for testing. The prioritizing algorithm assigns each test case a weight based on the time used to complete the test, and the distance the test case's failures were from the present execution. In order to achieve the aim of a quick test runtime during regression testing, test cases that failed in earlier test runs should be prioritized effectively and given a short test execution time during regression testing. In addition, the findings demonstrate that compared to manually prioritized test cases, test cases prioritized simultaneously using ROCKET reduce test execution time by 40%, with 20% of the test suite being performed.

3.3 History-based test prioritization

Kim and Porter firmly believed in the history-based test prioritizing approach in their paper [21], which placed test cases in ascending order based on values determined from previous test case execution data. They contend that under severe time and resource restrictions, the number of modifications made between testing sessions significantly impacts how well specific techniques perform. They claim that it is essential to choose test cases that haven't been run in a long time. Additionally, they think using historical data may help keep regression testing techniques from becoming too expensive or ineffective.

In another study by Kim and Porter [22], they based selection on static source code analysis and not on test result history. They contend that while the RTS approach may result in cost savings, its efficacy varies greatly depending on work attributes, including program size, test suite size, and change size. They found that as the testing interval grows, so does the proportion of errors that the initial test suite is unable to identify, which may impact the performance of all RTS techniques. According to their findings, random selection is highly affordable and efficient. The difference in performance between minimization at low testing intervals and minimization at high testing intervals showed the most minimized selection executed the lowest performance of all their tested selection designs.

[1] https://www.jacoco.org

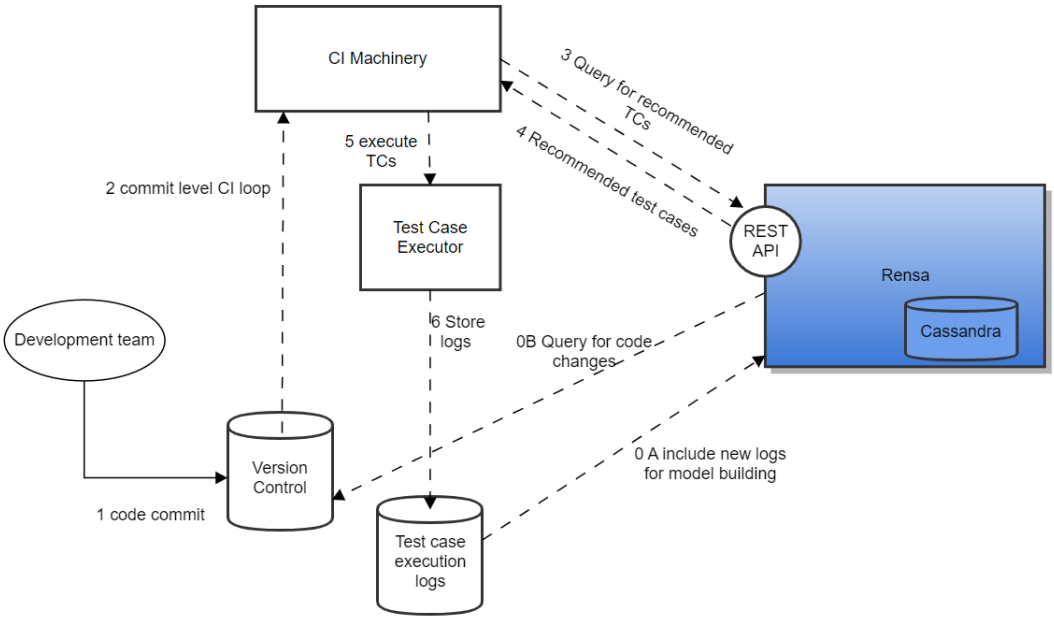


Figure 3.1: High level overview of RENSA's one logical place in the development flow.

3.4 Reflective Engine for Statistical Analysis (RENSA)

RENSA is a tool developed by Ericsson using machine learning to analyze historical test case results, correlate them with code changes, and provide a test case list based on the changed code fragment [23]. The benefit of using RENSA is mainly to have the most relevant test cases selected dynamically for the given code changes. This way, RENSA prioritizes the test cases most likely affected by the code change. RENSA should be installed on a machine with access to the code repository and the database storing the test case results. Having all the test case results stored in a database is a prerequisite. Once the developer pushes code changes, the code in a commit/module level scenario and the CI machinery is triggered. It turns to RENSA and queries, which are the most likely test cases (TCs) that will be affected by those code changes, see fig. 3.1. Based on its historical database, RENSA provides the most likely affected test cases ordered by their sensitivity (the more likely a TC changes its state for a given file change, the more sensitive that TC is considered). Based on this recommendation—which can be overwritten—the CI machinery executes the test cases and stores its log on the configured database. RENSA is used within Ericsson for regression testing of different software systems, and it is used by various teams across the company, including quality assurance (QA), testing, and development teams, but still can't be adopted in this testing section at Ericsson because of the lack of recorded data.

4 Method

The method chapter of this thesis will provide an overview of the research design and methods used to conduct the study. It describes the intended method, the chosen approach, the building approach, and the evaluation method. The study will also include in-depth interviews with expert engineers to discuss possible solutions.

4.1 Pre-study

A pre-study was conducted to further explore the research topic and identify potential areas for further investigation. The pre-study involved conducting interviews inside Ericsson with experts in the field from various departments, such as persons who have already researched test case selection, the RENSA development team, and database engineers. During the meetings with the RENSA development team, we discussed how RENSA works and which methods are used to see if we can get any benefits or inspiration from it. The departments addressed various problems, including handling and getting historical data for test case executions (TCEs). An expert engineer in Jenkins in the section was also involved in some meetings to provide more details about Jenkins' jobs and how to let Jenkins automatically run the selected tests. Another expert engineer in the database helped several times to get and filter the historical records.

As mentioned earlier, the project has been built in a maven environment. In order to apply any strategy in test case selection, it was possible to write an external script and get the benefits of mavens' plugins to execute it in the project. The *exec-maven-plugin* has a configuration to run scripts as Python scripts during the Maven build lifecycle, where we can execute programs in the same process or a separate one. After that, in its role, Jenkins has to start a new job on the selected tests. Therefore, a preliminary study was conducted to learn more about how Jenkins and Maven work and what features they have.

4.2 Building the First Approach Using Dependency Analyzer

The decision made to build the first approach based on the idea of Regression Test Selection EKSTAZI, which tracks dynamic dependencies of tests at the class level, the model is presented in the fig. 4.1. This approach doesn't require any data sources except access to the git

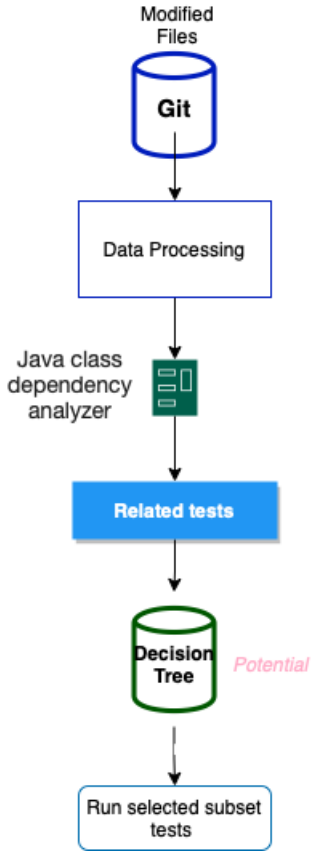


Figure 4.1: The first model for TCS.

repo. A Java library called Java Dependency Analysis Tool (JDeps)[1] was used to collect all dependencies. This tool takes a class or JARs containing them and statistically analyzes the declared dependencies between classes. The results can be filtered in various ways/options which affect the output and can be aggregated to the package or JAR level. Running the tool is simple in a new command-line tool introduced in JDK 8.

```
jdeps -recursive -verbose:class -cp class_path
```

As seen in the command line, it is written with three flags: *-recursive* recursively traverses all dependencies; *-verbose:class* prints class-level dependencies excluding dependencies within the same archive; and *-cp* specifies where to find class files.

In order to get the modified files/classes, the *git diff*[2] command is used. Diffing is a git function that tracks the differences between modifications to a file. It takes two data sets as input and produces the differences between them. *git diff* is a multi-purpose Git tool that performs a diff on Git data sources when performed. Commits, branches, and files may all be used as these data sources. The results from the dependency analyzer will be used as input to a decision tree algorithm 2.6 to minimize even more the test scope and finally the selected tests will be executed. Due to the significant number of integration tests that require more time to run, we decided to apply the model first on unit tests and then on the integration tests.

[1]https://docs.oracle.com/javase/9/tools/jdeps.htmJSWOR690
[2]https://git-scm.com/docs/git-diff

4.3 Data Collection

Due to the results from the first model, we figure out that we need to work on another model that requires more access to the database. In the beginning, we didn't have enough data history about test executions until we got help from another developer who works with the database. The data source is TGF (Test Governance Framework), which is a test execution engine. TGF was created in 2012 to tackle problems such as poor STP utilization, faults in everyday repetitive tasks, i.e., waste, a poor overview of bottlenecks, inefficiency, no test scope awareness, and redundancy. An STP is a simulated radio network used to test Ericsson's products since it's not possible to test the products on a live network, as it could interfere with the traffic that's going there. The TGF has some limitations, it can give a maximum of 1000 test results for one STP. The delimitation for this thesis has 34 STPs, which results in a maximum of 34 000 test results divided into 34 CSV lists, and each test suite is represented as a column in a list. There are many options and flags in the TGF framework, which allows filtering and deciding which test results you want to show like only failed tests. We chose to save the test executions in a list with the following parameters: test id, test suite, the result of the test, and run time.

4.4 Building the Second Approach

After studying the situation and considering the available resources, such as data and time, the decision was made to build the final model by prioritizing test suites based on failed history. The idea came from Test Prioritization algorithms and was not too complicated to be implemented. All it needs is historical and defect information of the code under test. Several studies [21] [24] [25] [26] have shown results improvement by using historical data and focused on improving the rate of detecting severe faults in regression testing. There is a critical limitation to this model, some records for test suites could go back a long time, maybe more than one or two years, and the test suite could be changed during this time. The built model is called Test Case Selector (TCS), which works as a script runnable from the command line. There was no mandatory language to write the code in because it was independent of the source code, so Python was used to write the script. The reason for choosing Python is that this language has libraries such as "pandas" which is a library for data analysis; it supports CSV lists and allows reading, writing, and manipulating any data stored in CSV files. Furthermore, Python is familiar to most programmers and easy to understand in case we want to make any upgrades to the feature. As shown in the fig. 4.2, a user can run the script from the command line and the script will begin to send commands to the terminal in order to collect the data from the database TGF by running this line:

```
tgr STP_name rbscipro nok 1000 --id,suite,finished | grep -v " ok"
| sed "s/\.\.\.\///g" > output.csv
```

We log in the user as rbscipro (Radio Base Station Node Continuous Integration Production FEM account) to ensure no private runs or dummy tests will be saved in our data, where it saves only the automatic tests that run every night. By running this line for each STP 1, the output is saved as CSV lists, and any previously stored lists are replaced. The script will read those 34 CSV lists and take them as input to go through them one by one and save the most failed test suites into a new file. In the second step, after handling all the lists, the script will choose the top seven failed tests from all STPs and send them to Jenkins. In its role, Jenkins will then start new jobs for the recommended tests to run those tests. The model doesn't return more than seven test suites because the current static model runs only two test suites and the requirements were to be a number of chosen suites around two. Even seven test suites can be a significant number for the department but it can be adjusted to be two, three, five, or any number.

Algorithm 1 Test Case Selector (TCS).

```
for All STPs do
    Collect data of test executions history for each STP using TGF
    Save the outcomes to .csv files
end for
for File in all .csv files do
    for Read all lines do
        Split the parameters and mark the similar test suite names
        Add the same test suite names to a new list
    end for
    for The suites in the new list do
        count the repetition
    end for
    Sort the five most failed tests in each STP
    Save five most failed tests from each STP to one file
    Select the seven most failed tests from all STPs
end for
```

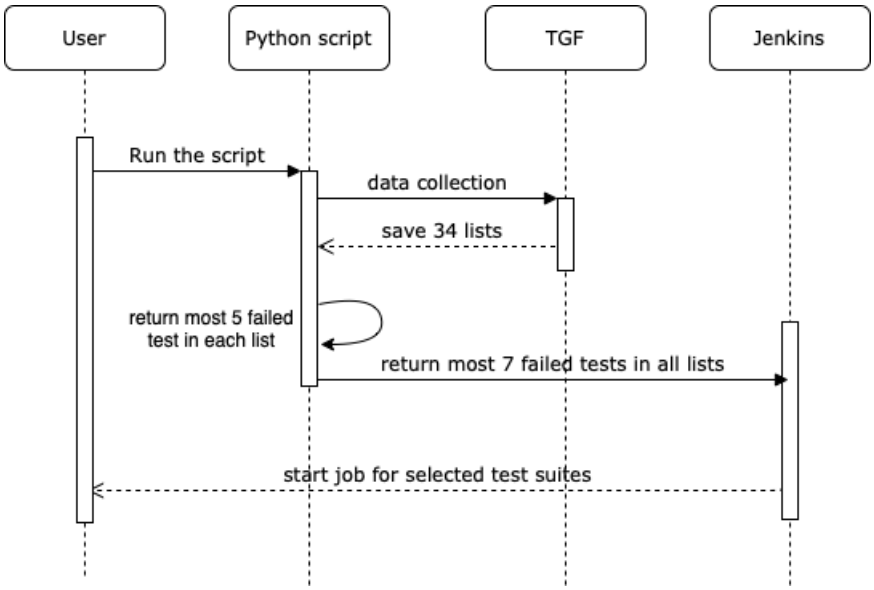


Figure 4.2: Final design of test case selector.

4.5 Evaluation Method

The evaluation of the developed test case selection models is based on the performance of the proposed approaches by comparing them with the currently used technique, which always runs the same two test suites. The evaluation aims to measure the percentage of failed tests found by running outputs from TCSs and comparing them with the static executed tests [27]. Generally, a higher ratio of failed test suites is considered harmful because it means more bugs or issues in the software being tested. A lower ratio of failed test suites indicates that the software is more stable and reliable. But for our models, a higher ratio of failed test suites means the model works better and detects more errors and bugs than the current model. The required time for the models is also measured using the same data. The second model uses a set of features listed in table 4.1. Furthermore, the evaluation is performed in close collaboration with the guardian of the Git repository so that the models may be modified in response to his comments. The results were evaluated on a Linux environment on a single work com-

puter using a Citrix VDI remote[3] to access Ericsson's closed environment with resources of 8 CPU cores and 32GB of RAM.

Table 4.1: Features used by the model.

Name	Description
TP100	Number of times the test suite passes in the last 100 builds that execute test cases
TF100	Number of times the test suite fails in the last 100 builds
TFR100	Test suite failure rate for the last 100 builds
TPR100	Test suite passes rate for the last 100 builds
AT	Average time in minutes
TP1000	Number of times the test suite passes in the last 1000 builds that execute test cases
TF1000	Number of times the test suite fails in the last 1000 builds
TFR1000	Test suite failure rate for the last 1000 builds
TPR1000	Test suite passes rate for the last 1000 builds

[3]https://www.citrix.com/products/citrix-daas/citrix-virtual-apps-and-desktops.html

5 Results

This chapter presents all the results and findings from the thesis work.

5.1 Java Dependencies Analyzer Results

To get an idea of Jdeps' output ordering, we run this example on a random class,

```
user> jdeps -verbose TestSuite.class
```

and the results are as follows,

```
TestSuite.class -> java.base
    TestSuite   -> com.ericsson....ScriptSupportTest.class not found
    TestSuite   -> com.erics....ConfigurationansTest.class not found
    TestSuite   -> java.lang.Object     java.base
    TestSuite   -> java.lang.String     java.base
```

where the first line is showing module-level dependencies and the other 4 lines are showing class level dependencies. So the given class "TestSuite" needs 4 other classes to run.

We first use the dependency analyzer on the unit test level as discussed in the method chapter 4.2. The results are represented as follows:

```
UnitTestSuite.class -> Home:\Progam\Java\jdk1.8\lib\FOAM.jar
    UnitTestSuite   -> java.io.PrintStream
    UnitTestSuite   -> java.lang.Object
    UnitTestSuite   -> java.lang.String
    UnitTestSuite   -> java.util
    UnitTestSuite   -> java.util.function
    UnitTestSuite   -> java.util.stream
```

```
jdeps -recursive -verbose:class -cp FOAM.jar --class-path 'lib/*'
```

By calling this jdeps command in the integration test suites results in the dependencies of a specific class in the following output:

```
                                .
                                .
                                .
com.msran...ExampleGetEnm -> com...resourcemanager.Enm   FOAM.jar
com.msran...ExampleGetEnm -> com...interfaces.Resource   FOAM.jar
com.msran...ExampleGetEnm -> com...NodeActionBasee   FOAM.jar
com.msran...ExampleGetEnm -> com...internal.NodeResult   FOAM.jar
com.msran...ExampleGetEnm -> com...internal.NodeResult   FOAM.jar
com.msran...ExampleGetEnm -> com...internal.NodeResults   FOAM.jar
com.msran...ExampleGetEnm -> com...ExampelGetEnmResult   FOAM.jar
com.msran...ExampleGetEnm -> com...helpers.Helpers   FOAM.jar
com.msran...ExampleGetEnm -> com...internal.NodeResult  FOAM.jar
com.msran...ExampleGetEnm -> com...util.UltihelperFactory FOAM.jar
com.msran...ExampleGetEnm -> com...util.StringHelper   FOAM.jar
com.msran...ExampleGetEnm -> com...util.UtiHelperFAC   FOAM.jar
com.msran...ExampleGetEnm -> java.lang.Iterable   java.base
com.msran...ExampleGetEnm -> java.lang.Object   java.base
com.msran...ExampleGetEnm -> java.lang.String   java.base
com.msran...ExampleGetEnm -> java.lang.invoke.Callsite  java.base
com.msran...ExampleGetEnm -> java.lang.invoke.LambdaMeta java.base
com.msran...ExampleGetEnm -> java.lang.invoke.Methodhandle java.base
com.msran...ExampleGetEnm -> java...invoke.Methodhandles java.base
com.msran...ExampleGetEnm -> java...invoke.MethodType   java.base
com.msran...ExampleGetEnm -> java....StringConcatFactory java.base
com.msran...ExampleGetEnm -> java.util.List   java.base
com.msran...ExampleGetEnm -> java.util.function.Function   java.base
com.msran...ExampleGetEnm -> java.util.stream.Collector   java.base
com.msran...ExampleGetEnm -> java.util.stream.Collectors   java.base
com.msran...ExampleGetEnm -> java.util.stream.Stream   java.base
                                .
                                .
                                .
```

The list shows dependencies at the package level, which means which packages in the JAR depend on which other packages, by using the flag *-verbose* gets reports class-level dependencies rather than only package-level dependencies. The depends *java.base* tells as that the "com.ericsson.ch.common.cdf.CdfXmlExtractor" package requires a number of JDK packages. Where the *FOAM.jar* tells as that "com.ericsson.ch.common.cdf.CdfXmlExtractor" depends on another class.

The flag *-summary* is then used to give different perspectives on the dependencies, which only shows dependencies between classes in JAR as shown:

```
                                .
                                .
                                .
com...cdf.CdfXmlExtractor -> com...cdf.ArgumentChoice    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ArgumentIntArrayLabel FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ArgumentIntLabel    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ArgumentLabel    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ArgumentLabelScope   FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ArgumentStringLabel   FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ArgumentStruct    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf...SymbolicLabelArr    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf...SymbolicLabel    FOAM.jar
```

```
com...cdf.CdfXmlExtractor -> com...cdf.ArrayValidator    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf...LabelCollection    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.CommandLabel    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.EmbeddedArgument    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.IArgumentLabelColl    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.IArgumentVAllidat    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ICdDataExtractor    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.IntRangeValidator    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.IntSetValidator    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.IntegerArrayValid    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.ServerLabel    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.StringValidator    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.Symboliclabel    FOAM.jar
com...cdf.CdfXmlExtractor -> com...cdf.SymbolicLabelArr    FOAM.jar
                             .
                             .
                             .
```

5.2 Selecting Tests with TCS

The response from the TGF center depends on the number of jobs sorted in it, but it usually takes between 10 and 20 seconds to get the data history. The required time also depends on the number of called test cases, where the last 100 builds that execute test cases need less time than the previous 1000 builds. Regardless of the requested builds, we always call from all 34 STPs that we have, and the results are saved in CSV files visualized as shown:

```
STP#1:{TestSuite1.xml:274, TestSuite2.xml:203,...,TestSuite5.xml:98}
STP#2:{TestSuite1.xml:343, TestSuite2.xml:256,...,TestSuite5.xml:54}
   .                                  .
   .                                  .
   .                                  .
STP#34:{TestSuite1.xml:315, TestSuite2.xml:243,...,TestSuite5.xml:61}
```

Then we use those CSV files to find the most failed test suites, as shown

```
The most failed in all are:
    TestSuite1.xml      343     Failed
    TestSuite2.xml      315     Failed
    TestSuite3.xml      283     Failed
    TestSuite4.xml      282     Failed
    TestSuite5.xml      276     Failed
    TestSuite6.xml      274     Failed
    TestSuite7.xml      262     Failed
```

5.3 Evaluation of Most Failed Test Suites

As discussed in the method chapter (4), we call the TGF center to get the last 100 executed test cases for the static test suites, which are run every time any changes occur to the code. We measure the number of times the test suite fails and passes in the last builds, and we measure both the rate and the average time it needs to execute table 5.1. The total number of test execution is not precisely 100 in all cases because some executions have been stopped and didn't have any outcome.
After that, we compare the same numbers by running and presenting the results of the two 5.2, three 5.3, five 5.4, and seven 5.5 most likely test suites to fail that we got from TCS outputs.

We called the TGF center in the next experiment to get the last 1000 executed cases instead of 100. A few of the top test suites were different in this case from the previous one, and the results are presented in the table 5.7, table 5.6, and table 5.8. We use the abbreviations that have been introduced in the table 4.1

Table 5.1: Static test suites.

Test Suites	TF100	TP100	AT
Test Suite A	19	68	33
Test Suite B	0	96	8
TFR100 | TPR100 | AT	9.50%	82%	41

Table 5.2: Two most likely test suites to fail.

Test Suites	TF100	TP100	AT
Test Suite 1	48	41	14
Test Suite 2	35	65	23
TFR100 | TPR100 | AT	41.50%	53%	37

Table 5.3: Three most likely test suites to fail.

Test Suites	TF100	TP100	AT
Test Suite 1	48	41	14
Test Suite 2	35	65	23
Test Suite 3	43	51	63
TFR100 | TPR100 | AT	42%	52.30%	100

Table 5.4: Five most likely test suites to fail.

Test Suites	TF100	TP100	AT
Test Suite 1	48	41	14
Test Suite 2	35	65	23
Test Suite 3	43	51	63
Test Suite 4	29	25	55
Test Suite 5	17	72	13
TFR100 | TPR100 | AT	34.40%	50.80%	168

Table 5.5: Seven most likely test suites to fail.

Test Suites	TF100	TP100	AT
Test Suite 1	48	41	14
Test Suite 2	35	65	23
Test Suite 3	43	51	63
Test Suite 4	29	25	55
Test Suite 5	17	72	13
Test Suite 6	13	82	16
Test Suite 7	13	78	89
TFR100 | TPR100 | AT	28%	59%	237

Table 5.6: Two most likely test suites to fail in case 1000 builds.

Test Suites	TF1000	TP1000	AT
Test Suite 1	343	622	27
Test Suite 2	315	654	30
TFR1000 | TPR1000 | AT	32.90%	63.80%	57

Table 5.7: Three most likely test suites to fail in case 1000 builds.

Test Suites	TF1000	TP1000	AT
Test Suite 1	343	622	27
Test Suite 2	315	654	30
Test Suite 3	283	691	18
TFR1000 | TPR1000 | AT	31.40%	65.50%	75

Table 5.8: Static test suites in case 1000 builds.

Test Suites	TF1000	TP1000	AT
Test Suite A	188	789	33
Test Suite B	67	903	8
TFR1000 | TPR1000 | AT	12.75%	81.60%	41

6 Discussion

This chapter discusses the results from the previous chapter to get a deeper understanding of our findings and answer the research question of the thesis.

6.1 Java Dependencies Analyzer Performance

The ability to inspect the dependencies between classes as if they were a part of each other's code is a crucial feature of JDeps. Throwing the classes on the class path with the flag *-class:path* is a starting step toward achieving that aim. However, doing so merely allows JDeps to follow the paths into the JARs of our repo dependencies, so in order to actually analyze the dependencies as well as we wanted, we made JDeps recurse through the dependencies using the *-recursive* and *-verbose* options, which allow us to evaluate the dependencies. But the output becomes even more overwhelming as a result, so we needed to investigate how to interpret such a vast amount of information. The output of JDeps may be configured in a variety of ways. The flag option -summary, which displays inter-JAR dependencies as illustrated in the results, may be the best choice to utilize in a preliminary analysis of any project.

We tried this approach first with the unit tests, and it worked fine with them, but the aim of this thesis is the integration tests, not unit tests. The unit tests are running very fast, and there is no need to reduce them, unlike the integration tests, which require much more time to execute. The algorithms tree-based and Support Vector Machine described in 2 was planned to be used for the output of the JDeps in order to get a smaller scope. Then compare the results of those two algorithms and check the differences. When we tried the JDeps tool with integration tests, we got an unbelievably long list with too many dependencies more than 250 test suites. In other words, for each class, there was an inoperable list of dependencies on other classes, but the wanted results were much less where they should not be more than 20-30 classes, and then would minimize the test suites using a tree-based algorithm or support vector machine to get a number between two and seven test suites finally. After studying and analyzing our findings, we found that the code in integration tests in our repo is dependent on 80–90% of the source code, and the classes interact with lots of external dependencies so in every time we tried to find the dependencies for one class, we got a list of almost all other classes. The integration tests in this section at Ericsson are designed to test

the integration between different modules of the system. This means that the behavior of one module depends on the behavior of another module, and therefore the tests that verify the interaction between these modules may also be dependent on each other. In some cases, the integration tests involve also insetting up and tearing down test data and state, which can also create dependencies between tests. For example, if you have a series of tests that rely on a particular database state, each test may need to ensure that the correct state is present before running and clean up the state after running, which can create a dependency chain. That was theoretically unexpected based on our literature research, where there is much research about dependencies about class dependency and strategies depending on jDeps tool, but none of them mentioned these results. On the one hand, having dependencies between integration tests can ensure that the system is tested as a whole and that the different modules work together as intended. This can help catch issues that might not be caught by unit tests, which only test individual modules in isolation. On the other hand, if the dependencies between integration tests are too complex or poorly managed, it can make it difficult to maintain the tests and slow down the testing process. It also affects negatively the adoption of such test case selection models that depend on dependencies between tests. The paper doesn't include further details about implementing the tree-based algorithms and support vector machine because they would be implemented in later stages if the model worked fine.

The result we got from the java dependencies analyzer was unusable to help us get a smaller test scope or minimize the scope because the requirement from the start was to find a small scope at a maximum of between two and seven test suites. The output of this approach was more than 250 test suites, which uses too much time and resources to be executed. As we discussed before, this was the reason that let us change the way we were thinking and find another solution.

6.2 Evaluation Parameters Selection

Integration testing helps identify integration issues between the modules and find problems that are not immediately obvious by testing a specific unit's implementation. A key point in all this is that integration testing is designed foremost to detect bugs and faults, which means the tests that give us more bugs and faults are more sensible to run. For this reason, we chose to compare the ratios of failed test suites between the old static approach and the new dynamic one. Unlike unit testing, which runs very quickly, integration testing will cost more in terms of time. Therefore, most companies want to execute it as fast as possible. This makes the term time in focus in our evaluation.

6.3 Evaluate Results of TCS

The findings might be deceptive because of the understatement or misunderstanding of the risk involved in making decisions based on attribute testing. This is especially true when test suites are performed, and failure ratios are ascertained to be low. This shows that the faults in the system under test are harder to detect [27]. When failure is seen in a code that has been tested, the natural path to take is to proceed cautiously in forming inferences regarding the acceptance of the test code. Observing more significant failure rates results in an awareness of risk, which suggests that the executed test cases have found substantially more faults [26]. On the other hand, a low failure rate seen during testing often prompts a choice to go on with the code under examination. From the results table 5.1,table 5.2, we can see that the ratio of failure in the static approach is too low compared to the TCS model fig. 6.1, which means TCS has better performance than the static approach. Even one of the test cases in the static approach could not find any fault during the testing process table 5.1.

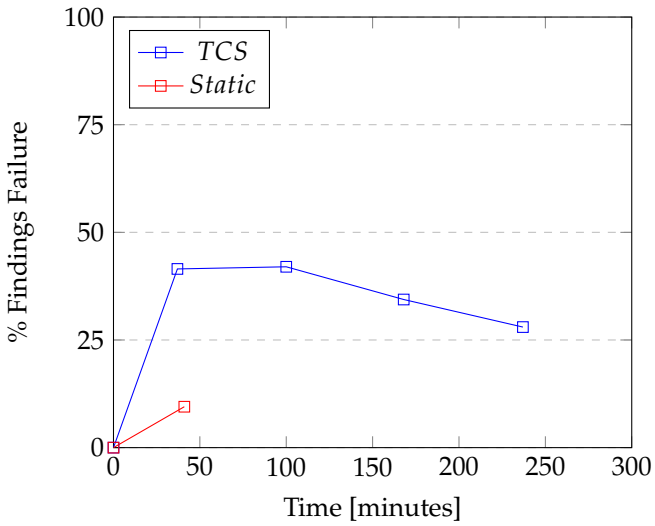


Figure 6.1: Failure detection using TCS.

However, making assumptions about a code without testing failures carries a risk. A low failure rate creates an illusion of assurance that is often unfounded. It is easy to forget that even if the static tests passed without error, we still can't be sure that the other tests can't find faults much more effectively than those static tests would have, especially when the static approach tests only two test suites if the user or developer does not choose other suites. This may result in late detection of a code change that has been the cause of failure in test suites rather than it would if the code change were detected instantly and run through the test suite at once. This will make determining and repairing the core issue of a test suite failure more challenging, considering that developers typically forget the precise context of a code change after a while. Overconfidence may lead to modulating or adding a part of the code or moving on with the development process that will not perform as intended.

The model has performed higher test suite failure rates than the current model, which means detecting more bugs, as shown in the fig. 6.1, where we have 41.50% against only 9.50% for the static one. Running the selected tests from TCS will give us a better chance to detect failures. The developers can run the test suite at regular intervals and before releasing the software to ensure that all known test failures are found and fixed. We couldn't compare the other tables with the static model because it runs only two test suites. In other respects, the developers run the test suites that they think are suitable for the change in the code.

We derive more information about the most failed test suites from testing a build of 1000 than from testing a build of 10. We keep the user's or developer's ability to change the build size where the user can choose how much they want to go back with historical data, up to a maximum of 1000 executions. The experiments were based on two build sizes, 100 and 1000, and in both of them table 5.2 and table 5.6, the failure rate was higher than in the static model table 5.1 and table 5.6. When we choose to go back to 100 builds that execute test cases in 34 STPs, the response time improved by 5-10 seconds (out of 20 seconds). It is more important to know that zero failures occur in a build size n than to simply know that zero failure occurs.

The time cost of tests from the TCS is on average 37 min table 5.2 less than the 41 min average for the static model table 5.1 when we select the same number of test suites to be executed. In case we want to execute more test suites that absolutely require more time but that will cover more faults, we rely on the user or developer experience to choose how many

test suites they want to run depending on the situation and the available time. The lower percentage in tables 5.3, 5.4, and 5.5 compared with table 5.2 doesn't mean worse detected failures; it gets lower because we select first the most failed test suits and count the ratio by dividing the number of failures by the number of builds that execute test cases. In this case, when we add the second and the third most failed test suite, we will normally get a lower rate. We don't want to let the tests take too long to the point that the testing doesn't come down to cost vs. benefit because it doesn't matter how good the testing is if the test cost is too expensive.

By running the test cases from TCS results and after checking the results with the repo guardian, we find that the tests cover multiple different test classes, which increases the coverage of the main repo codebase. Compared to running test cases that use the same test class, this results in a positive gain in finding issues with recently merged commits. If the executed test cases use the same test class code, the test coverage will be inferior, and the risk of fault slip-through will be higher.

6.4 Research Methodology

To begin, a literature review was conducted to explore the problem scenario and identify alternative solutions to identical issues. Using Google Scholar and the Linköping University library, which contains several databases such as Scopus, IEEE Xplore, SpringerLink, and ACM, I started by searching on the web for publications about selecting test cases in various methods, such as machine learning, selecting cases based on similarity, and selecting cases based on dependencies. Then, I picked out the top and relevant studies that represent other solutions to similar problems from the actual search and looked inside them for more references. Internal Ericsson materials were looked at to get the technical knowledge needed, and old papers in relevant fields were read to give a basis for the choices that were made.

Because a significant amount of research had already been conducted to choose a subset of all the tests, I limited the scope of my search to only include sources that discussed solutions that have been used in continuous integration. Because of this, I ignored most of the publications that dealt with approaches that were based on historical data because we lacked an adequate quantity of historical records. In addition, I noted that many of the approaches suggested were very specialized for specific usage scenarios. This leaves the door open for the idea of building a code with the assistance of tools that are currently in existence. The purpose of the pre-study would thus be to concentrate first on finding a tool or algorithm that may assist in the construction of a code that does not deal with historical test records or at least does not heavily rely on them.

After building the first model, it was also necessary to address other solutions that dealt with the historical records. The same previous electronic sources were used to find articles. Many articles' research findings were produced due to several title searches in the databases. An initial choice was reached based on each publication's title, and from those publications, the final choices were based on each publication's abstract chapter. The papers were assumed to be relevant if their titles or abstracts included test case prioritization or selection, especially in the context of test case prioritization based on historical data. Articles specifically discussing previous solutions were disregarded since they had already been covered.

The results from the literature methodology for selecting papers are shown in the table 6.1, where the most used database was Scopus 6.2. The most covered area was test selection over all selected papers 6.3, which is very natural because the main topic for this thesis is test selec-

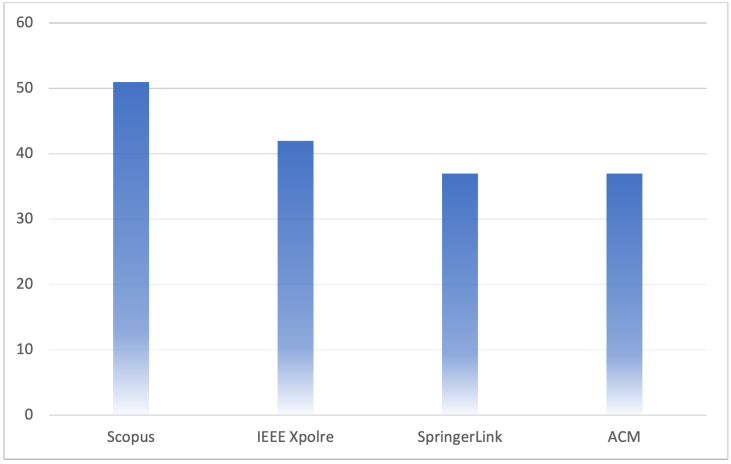


Figure 6.2: Literature from different databases.

tion regardless of the selected method. The lowest area was papers about test dependencies. The reason for that was choosing another approach that did not depend on dependencies.

Table 6.1: Table to show the searched topics on different databases.

Articles				
Topics	Scopus	IEEE Xpolre	Springer-Link	ACM
Continuous integration	6	5	8	3
Regression Test Selection	8	4	9	7
Test case selection	10	8	6	7
Regression Test dependencies	6	4	2	4
Test selection machine learning	9	5	5	8
Test Prioritization	7	8	4	3
Test selection historical data	5	8	3	5

6.5 Answering the Research Question

RQ: Which strategy fits for dynamically selecting a suitable test scope based on new code changes?
Based on our experiences and results, we conclude that the model based on dependency strategies are somewhat unsuitable for integration tests in such big projects. The model will include all dependencies to that file and run almost all tests, which will cost too much and not be a profitable solution. Even the static strategies could be helpful and take place in an early testing phase but not in the integration level of testing because it will run two or a maximum of three test suites to meet the time restriction. That may give a false sense that everything has been checked and addressed. The second model TCS was designed based on historical data, where it prioritizes the test suites according to the most likely tests to fail. This model was selected among other options for its ability to be modified depending on user or developer needs, where the model can be set to choose several test suites that the user or developer wants to test according to the available time and decide how far back through data history. Finally, the model which relies on historical data strategy is considered as a flexible and adaptable approach for effective test case selection.

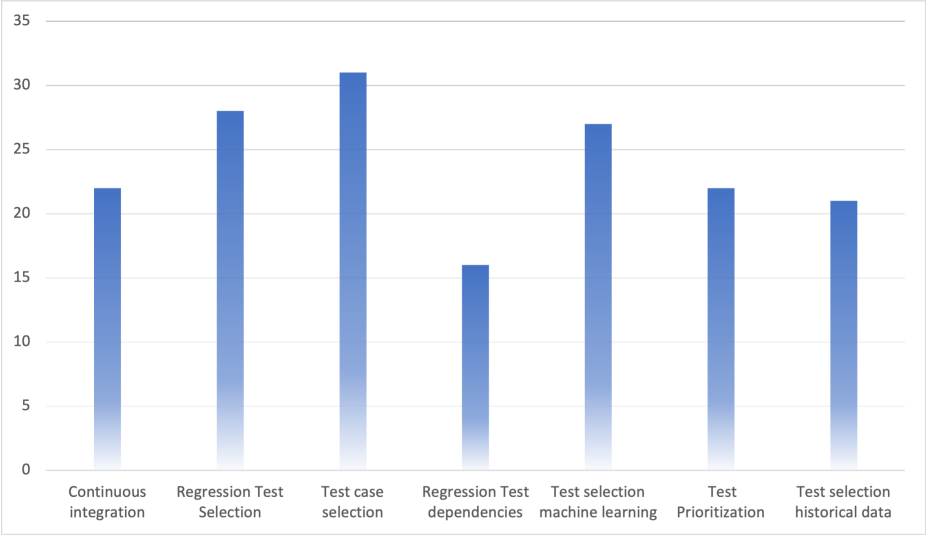


Figure 6.3: No. of articles for covered topics.

6.6 Threats to Validity

As we discussed before and showed that the model depends on historical record data and selects the most failed tests first, this increases the chance of choosing the test suites that already have a high ratio of failed tests if the test suites still fail. In other words, the user or developer will run the test suites with a low ratio of failures by selecting them or wait until they run by the automatic tests that run every night. Still, if the tests don't detect too many failures or are ignored by the user or developer, they will lose the chance to be selected by TCS.

Since the time aspect is critical and each test suite needs some time, we can not select too many test suites, which will be between 2 and 5 in a standard case. This number may not be enough to cover all risks relative to the number of test suites that we have, it might cost the company big losses if critical errors slip through.

Flaky tests threaten the model's validity since they are common in the software industry. Flaky tests can fail for no apparent reason, and they can pass sometimes and fail others for the same code [28], making test results unreliable because even when they succeed, they don't necessarily indicate the absence of a bug [29].

6.7 Ethical, Economic, and Environmental Considerations

The sources from which we gathered the data are accessible to all section workers. There are scenarios in which it could be handled unethically. For example, we compile information on which user made a particular modification. The information may be used to evaluate user working patterns in combination with other statistics we have obtained. To ensure that such things never happen, we hide the user id when we collect the historical data.

Since running tests for all commits requires a lot of resources, reducing the number of tests would be less expensive in both time and money since there would be fewer tests to run. That would even require less power, which would positively affect the environment.

Suppose the models do not choose appropriate tests. In that case, the consequences could be significant because of the large scale of Ericsson's systems and many users reliant on the systems' functionality. Errors might result in substantial financial losses for Ericsson and other partners.

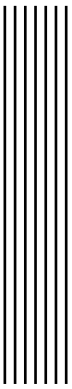
7 Conclusions and Future Work

This chapter contains a summarization of the purpose and the research question. Also, potential suggestions are presented to improve the model in the future.

This paper proposes two Test Case Selector (TCS) techniques and finally chooses one of them. The first technique is based on the dependencies between classes, and the second technique orders test suites according to historical test execution data. The second proposed technique was chosen where it prioritizes and selects test suites according to their fault rates without analyzing the source code. Compared with other traditional prioritization techniques, the proposed technique gives the user or developer a great ability to change the parameters to fit the case behaviors. Further, the proposed technique also avoids dummy and private runs to give better accuracy, improving prioritization performance.

The results highlight meaningful differences between TCS and static technique and have shown that the proposed technique can significantly improve its effectiveness compared to the current static technique. It provided higher efficiency because the detection of faults rate was much higher than the static test selection currently in use; it has also achieved a better reasonable coverage of function areas. Our experimental results support that the techniques based on dependencies between classes don't fit the integration test. They also support that historical information may help reduce costs and increase the effectiveness of long-running regression testing processes. Even with this technique, some test suites should be manually selected when it affects a specific function according to the user's or developer's expertise.

Future work includes conducting additional experiments to evaluate our technique as we want to use Jacoco code coverage 3.2 to obtain coverage result and compare it. Also, combining history-based techniques with other coverage-based techniques [25] seems to be a good potential attempt. Moreover, we aim to make the technique completely runs automatically, where the process starts automatically after each push without the need to run it by the user or developer.



Bibliography

[1] *Ericsson mobility report: 5g to top one billion subscriptions in 2022 and 4.4 billion in 2027*, Ericsson, 2022. [Online]. Available: https://news.cision.com/ericsson/r/ericsson-mobility-report--5g-to-top-one-billion-subscriptions-in-2022-and-4-4-billion-in-2027,c3588297 (visited on 04/21/2022).

[2] M. Castells, M. Fernandez-Ardevol, J. L. Qiu, and A. Sey, *Mobile communication and society: A global perspective*. Mit Press, 2009, ISBN: 9780262262309. [Online]. Available: https://books.google.se/books?id=cPIjf2DUxQAC.

[3] D. Ståhl and J. Bosch, "Industry application of continuous integration modeling: A multiple-case study," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2016, pp. 270–279. DOI: 10.1145/2889160.2889252.

[4] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017. DOI: 10.1109/ACCESS.2017.2685629.

[5] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007, ISBN: 9780321630148. [Online]. Available: https://books.google.se/books?id=PV9qfEdv9L0C.

[6] *Company facts*, Ericsson, 2021. [Online]. Available: https://www.ericsson.com/en/about-us/company-facts (visited on 04/21/2022).

[7] "ISO/IEC/IEEE: International standard - software and systems engineering —software testing —part 2:test processes," *ISO/IEC/IEEE 29119-2:2013(E)*, pp. 1–68, 2013. DOI: 10.1109/IEEESTD.2013.6588543.

[8] *CI/CD: It meets networks*, Ericsson, 2020. [Online]. Available: https://www.ericsson.com/48f6b1/assets/local/ci-cd/doc/cicd_it-meets-networks_2020.pdf (visited on 04/25/2022).

[9] H. Patil and G. Price, *CI/CD and CD/D: Continuous software delivery explained*, Sep. 2020. [Online]. Available: https://www.ericsson.com/en/blog/2020/9/cicd-and-cdd-continuous-software-delivery-explained (visited on 04/25/2022).

[10] M. Düsing, *Your guide to CI/CD: In telecom networks – for today and tomorrow*, Mar. 2021. [Online]. Available: https://www.ericsson.com/en/blog/2021/3/your-guide-to-cicd-in-telecom-networks--for-today-and-tomorrow (visited on 04/25/2022).

[11] *How gerrit works*. [Online]. Available: https://gerrit-review.googlesource.com/Documentation/intro-how-gerrit-works.html (visited on 04/25/2022).

[12] A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble, "Overview of use of decision tree algorithms in machine learning," in *2011 IEEE control and system graduate research colloquium*, IEEE, 2011, pp. 37–42. DOI: 10.1109/ICSGRC.2011.5991826.

[13] S. Osowski, R. Siroic, T. Markiewicz, and K. Siwek, "Application of support vector machine and genetic algorithm for improved blood cell recognition," *IEEE Transactions on Instrumentation and Measurement*, vol. 58, no. 7, pp. 2159–2168, 2008. DOI: 10.1109/TIM.2008.2006726.

[14] S. Mishra, *Breaking down the support vector machine (svm) algorithm*, Oct. 2020. [Online]. Available: https://towardsdatascience.com/breaking-down-the-support-vector-machine-svm-algorithm-d2c030d58d42 (visited on 04/25/2022).

[15] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 2, 2015, pp. 713–716. DOI: 10.1109/ICSE.2015.230.

[16] N. J. Wahl, "An overview of regression testing," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 1, pp. 69–73, 1999. DOI: 10.1145/308769.308790.

[17] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for. net," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 848–853, ISBN: 9781450351058. DOI: 10.1145/3106237.3117763.

[18] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering*, 2016, pp. 975–980. DOI: 10.1145/2950290.2983954.

[19] "Iso/iec/ieee international standard - software and systems engineering–software testing–part 4: Test techniques," *ISO/IEC/IEEE 29119-4:2015*, pp. 1–149, 2015. DOI: 10.1109/IEEESTD.2015.7346375.

[20] 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*, IEEE, 2013, pp. 540–543. DOI: 10.1109/ICSM.2013.91.

[21] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 2002, pp. 119–129. DOI: 10.1109/ICSE.2002.1007961.

[22] J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, 2000, pp. 126–135. DOI: 10.1145/337180.337196.

[23] M. Campbell, K. Martin, F. Bozóki, and M. Atkinson, "Dynamic test selection using source code changes," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, 2017, pp. 597–598. DOI: 10.1109/QRS-C.2017.111.

[24] Megala and K. Vivekanadan, "History based multi objective test suite prioritization in regression testing using genetic algorithm," 2017.

[25] A. Gupta, N. Mishra, A. Tripathi, M. Vardhan, and D. S. Kushwaha, "An improved history-based test prioritization technique technique using code coverage," in *Advanced Computer and Communication Engineering Technology*, Springer, 2015, pp. 437–448. DOI: 10.1007/978-3-319-07674-4_43.

[26] C.-T. Lin, C.-D. Chen, C.-S. Tsai, and G. M. Kapfhammer, "History-based test case prioritization with software version awareness," in *2013 18th International Conference on Engineering of Complex Computer Systems*, 2013, pp. 171–172. DOI: 10.1109/ICECCS.2013.33.

[27] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004. DOI: 10.1023/B:SQJO.0000034708.84524.22.

[28] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653, ISBN: 9781450330565. DOI: 10.1145/2635868.2635920.

[29] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840, ISBN: 9781450355728. DOI: 10.1145/3338906.3338945.